Semantics and applications of process and program algebra

Vu, T.D.

Publication date
2007
Document Version
Final published version

Semantics and Applications of Process and Program Algebra

Vu Thuy Duong

This book is typeset by the author using LaTeX2$_\varepsilon$.

Printing: Gildeprint Drukkerijen BV, Enschede, the Netherlands.

ISBN 978-90-9019794-4

Semantics and Applications of Process and Program Algebra

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus prof. dr. J.W. Zwemmer
ten overstaan van een door het college voor promoties ingestelde commissie,
in het openbaar te verdedigen in de Aula der Universiteit
op dinsdag 13 februari 2007 te 10.00 uur

door

Thuy Duong Vu

geboren te Hanoi, Vietnam

Promotiecommissie:

Promotor:	Prof. dr. J.A. Bergstra
Co-promotoren:	Dr. I. Bethke
	Dr. A. Ponse

Faculteit:	Natuurwetenschappen, Wiskunde & Informatica
	Kruislaan 403
	1098 SJ Amsterdam
	Nederland

UNIVERSITEIT VAN AMSTERDAM

The work described in this thesis has been carried out at the Sectie Software Engineering (SSE), part of the Computing, System Architecture and Programming Laboratory (CSP-Lab) of the University of Amsterdam, and under the auspices of the Institute for Programming Research and Algorithmics (IPA).

Contents

List of symbols

$\mathbb{P}$, $\mathbb{S}$, $\mathtt{states}$	set of processes or states
Σ	set of actions or instructions
$\mathtt{Act}$	set of actions
τ	silent (or internal) action
$\mathtt{tau}$	concrete internal action
$s \xrightarrow{a} s'$	transition from state s to state s' with label a
$\leftrightarroweq$	bisimulation equivalence
$\leftrightarroweq_o$	orthogonal bisimulation; pages 14, 21
$\leftrightarroweq_I$	bisimulation up to I; page 118
δ	deadlock; page 14
ϵ	termination; page 14
$\checkmark$	label of transition from ϵ to δ; page 14
$\leq$, $\sqsubseteq$	partial order
$\cong$	isomorphism relation; page 11
$\sim$	equivalence relation; page 11
$[e]_\sim$	equivalence class containing e; page 11
$E/_\sim$	set of equivalence classes on E; page 11
$\mathtt{run}(p)$	set of runs of a process p; pages 10, 14
$\mathtt{pre}(e)$	the pre-run of run e; pages 10, 15
$C(e)$	compression content; pages 10, 15
$\mathtt{last}(e)$	the last process of run e; page 14
$\tau\text{-}\mathtt{paths}(p)$	the set of all sequences $p_0 \dots p_n$ with $p_0 = p$, $n \geq 0$, and $p_i \xrightarrow{\tau} p_{i+1}$ for all $i < n$; pages 14, 21
$\lvert S \rvert$	number of elements of S; page 23
$\lvert \rightarrow \rvert$	number of transitions; page 23
$\lvert \theta \rvert$	length of the sequence θ; page 51
$\mathtt{pos}_a(B, B')$	the set of states in block B from which a state in block B' can be reached by action a; page 23
$\mathtt{pos}_\tau(B, B')$	the set of states in block B from which a state in block B' can be reached by a sequence of silent steps τ; page 23
$\mathtt{Ref}^a_P(B, B')$	the parition P where B is replaced by $\mathtt{pos}_a(B, B')$ and $B \setminus \mathtt{pos}_a(B, B')$; page 23
$r \sim_P s$	r and s are of the same block of the partition P; page 22
BTA_Σ, TA_Σ	set of finite threads
$\mathrm{BTA}^\infty_\Sigma$, $\mathrm{TA}^\infty_\Sigma$	set of (finite and infinite) threads

$\mathrm{BTA}_\Sigma^\omega$	set of Cauchy sequences; page 60
BTA_Σ^r	set of regular threads; page 66
S	successful termination
D	unsuccessful termination
$P \trianglelefteq a \trianglerighteq Q$	postconditional composition of processes P and Q; page 38
$a \circ P$	action prefix; page 38
$\mathtt{Pred}$	set of predicates; page 36
$\mathtt{ar}(f)$	the arity of function f; page 37
$\frac{H}{\pi}$	transition rule with H a set of premises, and π a premise; page 37
ρ	execution environment; page 40
$\mathtt{YIELD}$	the action to hand over control to another thread; page 43
$\|_{csi}$	cyclic interleaving operator; page 42
$\|_{csi}^{k,l}$	step counting; page 43
$\|_{ctp}$	current thread persistence operator; page 44
$\|_{csi}^{W2}$	allows threads to be performed simultaneously; page 45
$\|_{csi,f}$	thread creation (or forking); page 47
$\|_{csi,bff}$	blocked and failed fork actions; page 49
$\|_{csi,fn}^{\beta}$	terminating a named thread operator; page 49
$\|_{cip}$	the cyclic internal persistence operator; page 69
$\|_{csip}$	the cyclic interleaving with persistence operator; page 128
S_D	deadlock at termination operator; page 42
$\langle X \mid E \rangle$	solution for X in the guarded recursive specification E; pages 39, 65
π_n, $\pi_n^{\mathtt{tau}}$, π_n^I	approximation operators; pages 39, 70, 124
cpo	complete partial order; page 56
t_Φ	pre-abstraction operator; page 116
$\tau_{\mathtt{tau}}$, τ_t, τ_H	abstraction operators; pages 58, 117
$P \stackrel{I}{\Rightarrow} Q$	Q is a residual thread of P; page 118
$\mathtt{Var}_L$, $\mathtt{Var}_H$	sets of low and high variables; page 111
$=_{\mathbb{S}}^L$, $=_{\mathbb{S}}^H$	low and high equivalence between states; page 111
$\approx_{\mathbb{S}}^L$	low quasi-equivalence between threads; page 111
$\mathcal{L}$, $\mathcal{H}$	low and high security classes; page 112
$\mathtt{NI}$	NonInterference; page 111
$\mathtt{TINI}_I$	Termination-Insensitive NonInterference up to I; page 120
$\mathtt{TSNI}_I$	Termination-Sensitive NonInterference up to I; page 123
$\mathtt{TISNI}$	TIming-Sensitive NonInterference; page 125
Δ	disunification operator defined by $\Delta\mathbf{u}(Y) = Y$; page 82
$\sigma \oplus_X l$	the position obtained by jumping l steps forward from σ; page 83
$\sigma \ominus_X l$	the position obtained by jumping l steps backward from σ; page 83
$\mathtt{if} \pm a$	conditional instructions; page 85
$\mathtt{while} \pm a$	while-loops; page 90
$\mathtt{least\_jumps}(X, \sigma)$	the least jump that takes one outside X from σ ; page 86
$\mathtt{Upd}_2(X)$	updates the contents of jumps with two more steps; page 87
$£l$, $\#\#£l$	labels and gotos; pages 139
$P/_f B$	the use-operator to combines P and B; page 148
CoPA	the program notation representing Cobol in PGA; page 171
$[X]_\sigma$	the instruction of program X at position σ
$\mathtt{update}(i, j, \sigma, \chi)$	updates the content of χ in some cases; page 177

Chapter 1

Introduction

Process and program algebra are mathematical frameworks for the study of system and program behaviors. Process algebra is concerned with system behaviors (or processes) in general, while program algebra is a theory of computer programs. Both frameworks provide algebraic laws allowing process or program descriptions to be manipulated and analyzed mathematically. This thesis is about semantics and applications of process and program algebra.

1.1 Process algebra: Thread algebra and orthogonal bisimulation

Process algebra is defined over a set of *actions* together with a set of *operators*. Process algebras such as CCS (Calculus of Communicating Systems) [75] and ACP (Algebra of Communicating Processes) [19] have parallel operators to express concurrency. A system behavior as a process is an algebraic term satisfying the axioms of a process algebra.

In 2004, Bergstra and Middelburg introduced *thread algebra* (TA) [23], a process algebra for the semantics of object-oriented and multi-threaded programming languages such as C# and Java. It comprises *strategic interleaving operators* that turn a sequence of *threads* (or *polarized* processes) into a single thread capturing essential aspects of multi-threading. Here, strategic interleaving, in contrast to the *arbitrary* interleaving of other process algebras such as CCS and ACP, specifically determines the ordering of actions from the threads. TA is less general in dealing with parallelism. However, by considering a significant collection of different strategies, it is closer to a programmer's intuition. Recent results [28, 26] show that TA is a promising approach for the study of computer viruses and virtual machines. We study the semantics of TA. Chapter 4 presents a *structural operational semantics* (SOS) [81] for TA. Our SOS is less general but simpler than that of [23]. We show that the axiomatization of the strategic interleaving operators is sound and complete with respect to *bisimulation equivalence* as induced by the new SOS. This means that two processes are equal if

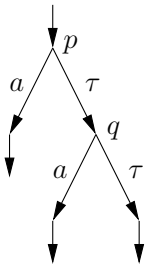

Figure 1.1: A labeled transition system.

and only if they are bisimilar.

Chapter 5 describes a *denotational semantics* [11] for TA. Turning the domains of TA into complete metric spaces, we show that the complete metric space consisting of projective sequences is an appropriate domain for TA. We also prove that the specification of a *regular* thread yields a unique solution. And finally, we propose a particular interleaving strategy for TA that deals with *abstraction* [21], an important operator that abstracts from *internal* actions in process algebra, in a natural way.

In giving semantics to process algebra, it is found that SOS is more flexible than denotational semantics [1]. A process is modeled by a *labeled transition system* generated by an SOS whose states are processes and whose transitions are labeled by actions. An example of a labeled transition system is given in Figure 1.1. In this example, the transition $p \xrightarrow{\tau} q$ means that process p can perform an internal action τ to process q. A *trace* of a process is obtained by putting in succession the actions of a run of that process. For instance, ϵ (the trace of an empty run), a, τ, τa and $\tau\tau$ are the traces of process p. Two process terms can be equated if their labeled transition systems are semantically equivalent. In 2003, Bergstra, Ponse and van der Zwaag proposed a semantic equivalence called *orthogonal bisimulation* [29] for process algebra. This equivalence is a refinement of *branching* bisimulation equivalence [52], the well-known semantics dealing with abstraction. A major difference between orthogonal bisimulation and other coarser semantics such as branching bisimulation, η-bisimulation equivalence [7], observation equivalence and delay bisimulation equivalence [73, 74] is that in orthogonal bisimilarity one may compress, but not completely discard internal activity. Therefore, a process without internal actions cannot be equivalent to one with internal actions. The advantage of orthogonal bisimulation is that it combines well with priority [6]. Also, orthogonal bisimulation has the following nice properties:

1. There is a modal logic based on Hennessy-Milner logic [57] which characterizes orthogonal bisimulation equivalence [29].

2. In the setting of ACP with abstraction, orthogonal bisimulation congruence is

completely axiomatized by three laws:

$$\begin{array}{rcll} x\tau\tau & = & x\tau & \\ x\tau(y+z) & = & x(y+z) & \text{if} \quad \tau y = \tau\tau y,\ \tau z = \tau\tau z \\ x(\tau(y+z)+z) & = & x(y+z) & \text{if} \quad \tau y = \tau\tau y \end{array}$$

Unlike in branching bisimulation equivalence, the axiom $x\tau = x$ is not sound in orthogonal bisimulation equivalence.

There are two open questions on orthogonal bisimulation raised in [29].

The first question is to define a trace characterization of orthogonal bisimilarity. The authors of [29] suggest a notion of *compression content* of traces to be the traces of a process from which all second and consecutive internal actions are removed. This might lead to a characterization of orthogonal bisimilarity as the trace characterization of branching bisimilarity given in [51]. That is, two processes are orthogonally bisimilar if and only if they have the same set of compression contents.

The second question is to determine the complexity for deciding orthogonal bisimulation in finite state transition systems.

In Chapter 2 we observe that there exist two orthogonally bisimilar processes that have different sets of compression content in the sense of [29]. We then define another notion of compression content of traces for a process, namely the traces of the process from which all internal actions are removed. Furthermore, if a trace ends in an internal action then the compression content of this trace is extended with the internal action symbol. Our trace characterization of orthogonal bisimulation equivalence depends on this new notion of compression content and is called the *compression structure of a process*. This compression structure characterizes orthogonal bisimilarity in the same way as in [51] the *branching structure* characterizes branching bisimilarity.

In Chapter 3 we study the complexity of deciding orthogonal bisimulation in finite state transition systems. Unlike in branching bisimulation, in orthogonal bisimulation, cycles of internal actions cannot be eliminated. Hence, the algorithm for deciding branching bisimulation of Groote and Vaandrager [54] cannot be adapted easily. However, we show that it is possible to decide orthogonal bisimulation with the same complexity as that of Groote and Vaandrager's algorithm. Thus if n is the number of states, and m the number of transitions then it takes $O(n(m+n))$ time to decide orthogonal bisimilarity on finite labeled transition systems, using $O(m+n)$ space.

1.2 Program algebra

Program algebra (PGA) [22], proposed by Bergstra and Loots in 2002, is an algebraic theory of sequential programming languages. The starting point of this research as stated in [14] is that the theoretical literature pays no real attention to the question of what constitutes a computer program. Bergstra and Loots define programs (or program descriptions) as expressions in PGA.

PGA is defined over a set of *primitive instructions* generated from a set of *basic instructions*, and two operators *concatenation* and *repetition*. The concatenation $X;Y$

of two programs X and Y in PGA is in PGA. The repetition Z^ω of a program Z in PGA is also in PGA. A basic instruction is viewed as a request to the environment, and it is assumed that upon its execution a boolean value (`true` or `false`) is returned that may be used for subsequent program control. The following is an example of a program in PGA.

$$X ::= +a; !; b^\omega.$$

In this example, the instructions $+a$, b and ! are primitive instructions. The primitive instructions $+a$ and b are obtained from basic instructions a and b. The instruction ! yields termination of the program. Program X first performs a. If `true` is returned after the execution of a, the program proceeds with the subsequent instruction and terminates. If `false` is returned, the program skips the termination instruction ! and performs b repeatedly.

Unlike in process algebra, the semantics of a program in PGA is given in a separated setting. More precisely, a regular thread in TA is assigned to a program in PGA by means of *behavior extraction equations*. Based on PGA, more complex programming languages can be developed and studied by providing other general constructs. In particular, a programming language is defined as a pair (L, ψ) where L is some collection of textual objects and ψ a *program algebra projection*. The program algebra projection ψ can be seen as a formal semantics that assigns to objects in L programs in PGA, possibly with the use of state machines [27]. This definition of a programming language is simplistic, but it covers the principles of programming languages such as Cobol, Java, C# and other programming paradigms such as ASF+SDF [56, 18]. A steady development of the core theory of PGA has been created and results on Maurer computers [26] and risk assessment [28] were achieved.

In Chapter 6 we study PGA itself and explore the expressiveness of extensions of PGA with conditional statements and while-loops with respect to a lazy projection semantics proposed in [22]. The advantage of this semantics, in comparison with the full projection semantics of conditional statements and while-loops of [22], is its simplicity. We show that PGA with while-loops yields non-regular behaviors in certain cases. Under a restriction that avoids non-regular behaviors, we present two projections from PGAuc (PGA extended with units [82] and conditional statements) and PGAucw (PGAuc extended with while-loops) to PGA. The existence of these projections shows that conditional statements and while-loops, while allowing for a flexible style of programming, are not needed as primitive instructions in terms of expressiveness.

1.3 Applications of thread and program algebra

In Chapter 7 we claim that the semantics TA of PGA creates a process-algebraic framework for formalisation and analysis of security properties in language-based security [88]. We quote from [88]: "There is a little assurance that current computing systems protect data confidentiality and integrity; existing theoretical frameworks for expressing these security properties are inadequate, and practical techniques for enforcing these properties are unsatisfactory". The current approaches based on *type*

systems (see [88] for an overview) are confusing when parallelism is introduced in the languages, and they cannot be applied to unstructured programs. Our approach accepts all secure programs that are accepted by the type systems defined in [99, 91]. The advantage of our work is that it is suitable for considering security properties in multi-threaded and unstructured programming languages. Furthermore, we can use existing tools such as [48, 49, 31] for checking process-equivalence to develop our security checkers.

Finally, in Chapter 8 we show that PGA provides a mathematical and systematic framework for reasoning about the correctness and equivalence of algorithms and transformation rules for goto removal [45]. Goto removal eliminates goto statements from a program by replacing them with other structured constructs. Although goto removal has been studied for several decades, it is still important because of maintenance and redevelopment of legacy software systems. A particular application of goto removal is to extract business knowledge embedded in these systems [92]. Once the business logic from a legacy system has been extracted, it is ready for integration and migration. Existing techniques indicate that goto removal can be applied, but their correctness and equivalence have not been discussed formally. We explain goto removal with mathematical rigor. We also provide formal correctness proofs for some transformation rules to restructure Cobol programs in a real-life application [96].

1.4 The structure of this thesis

This thesis consists of three parts. The first part contains four chapters (from Chapter 2 to Chapter 5) presenting our results on process algebra, in particular on orthogonal bisimulation and thread algebra. The second part consists of Chapter 6 and presents our results on program algebra. The third part consists of the last two chapters and is a connection between these two topics. All chapters are self-contained, and can be read separately. Chapter 2 was published as [103]. Chapter 3 is to be published as [100]. The chapters are available from `http://www.science.uva.nl/~tdvu`.

Part I

Process algebra: Thread algebra and orthogonal bisimulation

Chapter 2

The compression structure of a process

2.1 Introduction

In [29], Bergstra, Ponse and van der Zwaag introduce *orthogonal* bisimulation equivalence as a refinement of *branching* bisimulation equivalence [52]. A major difference with branching bisimulation equivalence and other coarser semantics dealing with abstraction [73, 74, 7] is that in orthogonal bisimilarity one may compress, but not completely discard *internal* activity (the performances of τ-steps). Therefore, a process without τ-steps cannot be equivalent to one with τ-steps.

Moreover, it is well-known that the priority operator [6] is not fully compatible with any known semantics that deals with abstraction. Several solutions for this problem have been proposed [95, 34], but none of these are totally satisfactory and generally accepted. The main advantage of orthogonal bisimulation equivalence is that it is a congruence for ACP (Algebra of Communicating Processes) [19] with abstraction and priority operators [6].

These features of orthogonal bisimulation represent another perspective on abstraction in process algebra. Hence, there is a need to reconsider results on previously defined semantics.

In the case of branching time semantics, van Glabbeek has proposed a definition called *the branching structure* of processes [51]. This definition depends on a notion of *observable content* of the *traces* of a process, that is, the traces from which all internal actions are removed. For instance, $\{\epsilon, a\}$ is the set of observable contents of process p in Figure 2.1. Here, ϵ denotes the trace of an *empty run* of a process. Two processes have the same branching structure if every trace of one process has a correspondence in the other with the same observable content. It is shown in [51] that branching bisimulation preserves the branching structure of processes, unlike the standard *observation equivalence* of Milner [73]. As a consequence, a behavioral equivalence on labeled transition systems respects the branching structure of processes

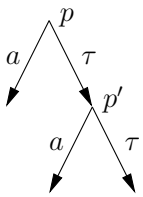

Figure 2.1: An example of a process.

if and only if it is finer than or equal to branching bisimulation equivalence (for processes containing τ-steps). Thus orthogonal bisimulation equivalence respects the branching structure of processes.

This chapter presents a trace characterization of orthogonal bisimilarity which is a solution to the open question on orthogonal bisimulation raised in [29]. The authors of [29] suggested a notion of *compression content* of the traces to be the traces of a process from which all second and consecutive internal actions are removed. This notion might lead to a characterization of orthogonal bisimilarity as the trace characterization of branching bisimilarity given in [51]. According to this notion, $\{\epsilon, a, \tau, \tau a\}$ is the set of compression contents of process p (see Figure 2.1). However, we observe that there exist two orthogonally bisimilar processes that have different sets of compression content in the sense of [29]. We then define another notion of compression content of the traces of a process, which is the traces of the process from which all internal actions are removed. Furthermore, if a trace ends in a τ-action then the compression content of this trace is extended with τ. The set of compression contents of process p in Figure 2.1 is now given by $\{a, \tau\}$. Our trace characterization of orthogonal bisimulation equivalence depends on this notion and is called the *compression structure of a process*. We show that two processes have the same compression structure if and only if they are orthogonally bisimilar.

The structure of this chapter is as follows. Section 2.2 defines the compression structure of processes. In Section 2.3, we instantiate the general definition for the case of labeled transition systems. The chapter is ended with some comments in Section 2.4.

2.2 The compression structure of a process

In this section we define the compression structure of a process. Let $\mathbb{P}$ be a class of processes. We assume that for each process p a set $\mathtt{run}(p)$ of partial runs is defined, equipped with a prefix ordering $\leq$, and that each run $e \in \mathtt{run}(p)$ is associated with its compression content $C(e)$ and pre-run $\mathtt{pre}(e)$.

Definition 2.1 *A* **labeled partial order set** *with a* **pre-run** *function (elpop) is a tuple* $(E, \leq, C, \mathtt{pre})$ *with* E *a set,* $\leq$ *a partial order on* E*,* C *a function with domain* E*, and* $\mathtt{pre} : E \to E$ *a function. Two elpops* $(E, \leq_E, C_E, \mathtt{pre}_E)$ *and* $(F, \leq_F, C_F, \mathtt{pre}_F)$

are **isomorphic** *if there exists a bijective function* $i : E \to F$ *such that for* $e, e' \in E$, $e \leq_E e' \Leftrightarrow i(e) \leq_F i(e')$, $C_F(i(e)) = C_E(e)$, *and* $e = \mathtt{pre}_E(e') \Leftrightarrow i(e) = \mathtt{pre}_F(i(e'))$. *An isomorphism* ($\cong$) *is an equivalence relation, and an equivalence class of isomorphic elpops is called a* **partial ordered multi set** *or* **pomset** *with pre-runs.*

Definition 2.2 *(***Compression congruence***). If* $\sim$ *is an equivalence relation on a set* E, *then* $E/_\sim$ *denotes the set of equivalence classes of* $\sim$ *and* $[e]_\sim$ *denotes the equivalence class containing* $e \in E$.

A **compression congruence relation** *on an elpop* $(E, \leq, C, \mathtt{pre})$ *is an equivalence relation* $\sim$ *on* E, *such that*

$$\begin{array}{rcl} e \sim f & \Rightarrow & C(e) = C(f), \\ \exists f(e \sim f \leq f') & \Leftrightarrow & \exists e'(e \leq e' \sim f'), \\ \mathtt{pre}(f') = f \wedge e \sim f & \Rightarrow & \exists e'(\mathtt{pre}(e') = e \wedge e' \sim f'). \end{array}$$

If $\sim$ *is a compression congruence relation on an elpop* $(E, \leq, C, \mathtt{pre})$, *then* $\leq_\sim$, $C_\sim$ *and* $\mathtt{pre}_\sim$ *are defined on* $E/_\sim$ *by*

$$\begin{array}{rcl} [e]_\sim \leq_\sim [f']_\sim & \Leftrightarrow & \exists e' \in [f']_\sim (e \leq e') \quad (\Leftrightarrow \exists f \in [e]_\sim (f \leq f')), \\ C_\sim([e]_\sim) & = & C(e), \\ \mathtt{pre}_\sim([e]_\sim) & = & [\mathtt{pre}(e)]_\sim. \end{array}$$

Definition 2.3 *The* **compression structure** *of a process* p *is the isomorphism class of* $(\mathtt{run}(p)/_\sim, \leq_\sim, C_\sim, \mathtt{pre}_\sim)$, *where* $\sim$ *is the coarsest compression congruence on* $(\mathtt{run}(p), \leq, C, \mathtt{pre})$. *An equivalence* $\equiv$ *on* $\mathbb{P}$ *respects the compression structure of a process if*

$$p \equiv q \Rightarrow (\mathtt{run}(p)/_\sim, \leq_\sim, C_\sim, \mathtt{pre}_\sim) \cong (\mathtt{run}(q)/_\sim, \leq_\sim, C_\sim, \mathtt{pre}_\sim).$$

Definition 2.4 *Let* $p, q \in \mathbb{P}$. *A relation* $R \subseteq \mathtt{run}(p) \times \mathtt{run}(q)$ *with domain* $\mathtt{run}(p)$ *(so,* $\mathtt{run}(p) = \{x | \exists y \in \mathtt{run}(q)\, R(x, y)\}$*) and range* $\mathtt{run}(q)$ *(so,* $\mathtt{run}(q) = \{y | \exists x \in \mathtt{run}(p)\, R(x, y)\}$*) is called a* **compression relation** *between* p *and* q *if there exists an elpop* $(E, \leq, C, \mathtt{pre})$ *with* $\mathtt{run}(p) \cup \mathtt{run}(q) \subseteq E$ *such that* $R \cup R^{-1}$ *is a compression congruence relation on* E.

Lemma 2.5 *Two processes* $p, q \in \mathbb{P}$ *have the same compression structure if and only if there exist compression congruences* $\sim_p$ *and* $\sim_q$ *such that*

$$(\mathtt{run}(p)/_{\sim_p}, \leq_{\sim_p}, C_{\sim_p}, \mathtt{pre}_{\sim_p}) \cong (\mathtt{run}(q)/_{\sim_q}, \leq_{\sim_q}, C_{\sim_q}, \mathtt{pre}_{\sim_q}).$$

Proof: (Similar to the proof of Lemma 1 in [51].)

1. Only if: This follows from two facts:

(a) If $\sim$ is the coarsest and $\sim_1$ any congruence on an elpop $(E, \leq, C, \mathtt{pre})$, and $\sim_2$ is the coarsest congruence on $(E/_{\sim_1}, \leq_{\sim_1}, C_{\sim_1}, \mathtt{pre}_{\sim_1})$, then

$$(E/_{\sim_1}/_{\sim_2}, \leq_{\sim_1\sim_2}, C_{\sim_1\sim_2}, \mathtt{pre}_{\sim_1\sim_2}) \cong (E/_{\sim}, \leq_{\sim}, C_{\sim}, \mathtt{pre}_{\sim}).$$

(b) If $(E, \leq, C, \mathtt{pre}) \cong (E', \leq, C', \mathtt{pre}')$ and $\sim$ is a congruence on $(E, \leq, C, \mathtt{pre})$, then there is a congruence $\sim'$ on $(E', \leq, C', \mathtt{pre}')$ such that

$$(E/_{\sim}, \leq_{\sim}, C_{\sim}, \mathtt{pre}_{\sim}) \cong (E'/_{\sim'}, \leq_{\sim'}, C'_{\sim'}, \mathtt{pre}'_{\sim'}).$$

Moreover, $\sim'$ is the coarsest if and only if $\sim$ is.

2. If: Straightforward.

□

Proposition 2.6 *Two processes $p, q \in \mathbb{P}$ have the same compression structure if and only if there exists a compression relation between p and q.*

Proof: Sketch based on the proof of Proposition 1 in [51].

1. Only if: Let R be a compression relation between p and q. We define $\sim_p$ (and similarly $\sim_q$) as follows.

$$\begin{array}{rcl} e \sim_p e' & \Leftrightarrow & \exists n \geq 0\, \exists e_0, \ldots, e_n, f_1 \ldots, f_n : \\ & & e = e_0 R f_1 R^{-1} e_1 R f_2 R^{-1} \ldots R f_n R^{-1} e_n = e'. \end{array}$$

Since R is a compression relation between p and q, there exists an elpop $(E, \leq, C, \mathtt{pre})$ with $\mathtt{run}(p) \cup \mathtt{run}(q) \subseteq E$ such that $R \cup R^{-1}$ is a compression congruence relation on E. This follows that $\sim_p$ and $\sim_q$ are compression congruence relations on $(\mathtt{run}(p), \leq, C, \mathtt{pre})$ and $(\mathtt{run}(q), \leq, C, \mathtt{pre})$, respectively. Let $i : \mathtt{run}(p)/_{\sim_p} \to \mathtt{run}(q)/_{\sim_q}$ be defined by $i([e]_{\sim_p}) = [f]_{\sim_q}$ for some $f \in \mathtt{run}(q)$ with eRf. It is not hard to see that i is a bijective function. We show that i is an isomorphism by proving that it satisfies the following properties:

(a) $[e]_{\sim_p} \leq_{\sim_p} [e']_{\sim_p} \Leftrightarrow i([e]_{\sim_p}) \leq_{\sim_q} i([e']_{\sim_p})$;
(b) $C_{\sim_p}([e]_{\sim_p}) = C_{\sim_q}(i(e_{\sim_p}))$;
(c) $[e]_{\sim_p} = \mathtt{pre}_{\sim_p}([e']_{\sim_p}) \Leftrightarrow i([e]_{\sim_p}) = \mathtt{pre}_{\sim_q}(i([e']_{\sim_p}))$.

Properties (a) and (b) are trivial. Property (c) follows from:

- $[e]_{\sim_p} = \mathtt{pre}_{\sim_p}([e']_{\sim_p}) \Rightarrow i([e]_{\sim_p}) = \mathtt{pre}_{\sim_q}(i([e']_{\sim_p}))$: This is the case because

$$\begin{array}{rcl} [e]_{\sim_p} = \mathtt{pre}_{\sim_p}([e']_{\sim_p}) & \Rightarrow & e \sim_p \mathtt{pre}(e') \\ & \Rightarrow & \exists e''(\mathtt{pre}(e'') = e \wedge e'' \sim_p e' \\ & \Rightarrow & eRf \wedge e = \mathtt{pre}(e'') \wedge e'' \sim_p e' \\ & \Rightarrow & \exists f''(\mathtt{pre}(f'') = f \wedge e'' R f'') \wedge e'' \sim_p e' \\ & \Rightarrow & i([e]_{\sim_p}) = [f]_{\sim_q} = [\mathtt{pre}(f'')]_{\sim_q} \\ & & = \mathtt{pre}_{\sim_q}([f'']_{\sim_q}) = \mathtt{pre}_{\sim_q}(i([e'']_{\sim_p})) \\ & & = \mathtt{pre}_{\sim_q}(i([e']_{\sim_p})), \end{array}$$

- $[e]_{\sim_p} = \mathtt{pre}_{\sim_p}([e']_{\sim_p}) \Leftarrow i([e]_{\sim_p}) = \mathtt{pre}_{\sim_q}(i([e']_{\sim_p}))$: Similar to the previous case by replacing p, i with q, i^{-1}.

By Lemma 2.5, p and q have the same compression structure.

2. If: Let $i : \mathtt{run}(p)/_\sim \to \mathtt{run}(q)/_\sim$ be an isomorphism. We define a relation $R \subseteq \mathtt{run}(p) \times \mathtt{run}(q)$ with domain $\mathtt{run}(p)$ and range $\mathtt{run}(q)$ by $eRf \Leftrightarrow i([e]_\sim) = [f]_\sim$. Then R (similarly R^{-1}) has the required properties as we can see from:
(a) $eRf \Rightarrow C(e) = C(f)$:

$$C(e) = C_\sim([e]_\sim) = C_\sim(i([e]_\sim)) = C_\sim([f]_\sim) = C(f),$$

(b) $eRf \leq f' \Rightarrow \exists e'(e \leq e'Rf')$:

$$\begin{array}{rcl} eRf \leq f' & \Leftrightarrow & i([e]_\sim) = [f]_\sim \wedge f \leq f' \\ & \Rightarrow & \exists e''(i([e]_\sim) = [f]_\sim \leq [f']_\sim = i([e'']_\sim)) \\ & \Rightarrow & [e]_\sim \leq [e'']_\sim \wedge i([e'']_\sim) = [f']_\sim \\ & \Rightarrow & \exists e' \sim e''(e \leq e' \wedge i([e']_\sim) = [f']_\sim) \\ & \Rightarrow & \exists e'(e \leq e'Rf'), \end{array}$$

(c) $\exists f(eRf \leq f') \Leftarrow e \leq e'Rf'$:

$$\begin{array}{rcl} e \leq e'Rf' & \Leftrightarrow & e \leq e' \wedge i([e']_\sim) = [f']_\sim \\ & \Rightarrow & \exists f''(i([e]_\sim) = [f'']_\sim \leq [f']_\sim = i([e']_\sim)) \\ & \Rightarrow & \exists f \sim f''(f \leq f' \wedge i([e]_\sim) = [f'']_\sim) \\ & \Rightarrow & \exists f(eRf \leq f'), \end{array}$$

(d)$\mathtt{pre}(f') = f \wedge eRf \Rightarrow \exists e'(\mathtt{pre}(e') = e \wedge e'Rf')$:

$$\begin{array}{rcl} \mathtt{pre}(f') = f \wedge eRf & \Leftrightarrow & \mathtt{pre}(f') = f \wedge i([e]_\sim) = [f]_\sim \\ & \Rightarrow & i([e]_\sim) = [\mathtt{pre}(f')]_\sim \wedge [f']_\sim = i([e'']_\sim) \\ & \Rightarrow & i([e]_\sim) = \mathtt{pre}_\sim(i([e'']_\sim)) \wedge [f']_\sim = i([e'']_\sim) \\ & \Rightarrow & [e]_\sim = \mathtt{pre}_\sim([e'']_\sim) \wedge [f']_\sim = i([e'']_\sim) \\ & \Rightarrow & e \sim \mathtt{pre}(e'') \wedge [f']_\sim = i([e'']_\sim) \\ & \Rightarrow & \exists e'(\mathtt{pre}(e') = e \wedge e' \sim e'') \wedge [f']_\sim = i([e'']_\sim) \\ & \Rightarrow & \exists e'(\mathtt{pre}(e') = e \wedge e'Rf'), \end{array}$$

□

2.3 The case of labeled transition systems with and without τ

In this section, we instantiate our definition for the case of labeled transition systems. We define the compression content of a labeled transition system and show that the resulting compression structure characterizes orthogonal bisimilarity.

Definition 2.7 *A* **labeled transition system** *(LTS) with* **termination** *and* **deadlock** *is a pair* $(\mathbb{P}, \rightarrow)$ *with* $\mathbb{P}$ *a set of* **processes** *(or* **states***), and* $\rightarrow \subseteq \mathbb{P} \times \texttt{Act} \times \mathbb{P}$ *for* `Act` *a set of* **actions** *(or* **labels***). A triple* $(p, a, q) \in \rightarrow$ *is called a* **transition***. A process that has no outgoing transitions is called* **deadlock***, denoted by* δ*. Let* $\checkmark$ *be a label in* `Act`*. A process that has only one outgoing transition with label* $\checkmark$ *to* δ *is called* **termination***, denoted by* ϵ*. Furthermore, if* $(p, \checkmark, q) \in \rightarrow$ *then* $p = \epsilon$ *and* $q = \delta$*.*

We write $p \xrightarrow{a} q$ for $(p, a, q) \in \rightarrow$ and $p \xrightarrow{a}$ for $\exists q \in \mathbb{P} : p \xrightarrow{a} q$.

The elements of $\mathbb{P}$ represent the processes we are interested in, and $p \xrightarrow{a} q$ means that process p can evolve into process q by performing an action a. We note that the set of actions `Act` may contain an internal action, denoted by τ.

Next, we recall the definition of orthogonal bisimulation from [29]. Let $\tau\texttt{-paths}(p)$ be the set that consists of all sequences $p_0 \cdots p_n$ of states with $p_0 = p$, $n \geq 0$, and $p_i \xrightarrow{\tau} p_{i+1}$ for all $i < n$.

Definition 2.8 *(***Orthogonal bisimulation***). Let* $(\mathbb{P}, \rightarrow)$ *be an LTS. Two processes* $p, q \in \mathbb{P}$ *are* **orthogonally bisimilar***, denoted by* $p \leftrightarrows_o q$*, if there exists a binary symmetric relation* $\mathcal{B} \subseteq \mathbb{P} \times \mathbb{P}$*, called an* **orthogonal bisimulation***, satisfying*

1. *if* $p \xrightarrow{a} p'$ *for some* p' *and* $a \neq \tau$*, then* $q \xrightarrow{a} q'$ *for some* q' *with* $p' \mathcal{B} q'$*; or*
2. *if* $p \xrightarrow{\tau} p'$ *for some* p'*, then* $q \xrightarrow{\tau}$*, and there is a path* $q_0 \cdots q_n \in \tau\texttt{-paths}(q)$ *with* $n \geq 0$ *such that* $p' \mathcal{B} q_n$ *and* $p \mathcal{B} q_i$ *for all* $i < n$*.*

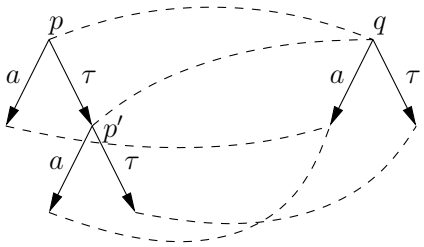

Figure 2.2: An example of orthogonal bisimilarity. The dashed lines represent orthogonal bisimulation equivalence between processes.

Definition 2.8 is illustrated in Figure 2.2. In this figure, two processes p and q are orthogonally bisimilar. However, the sets of compression contents $\{\epsilon, a, \tau, \tau a\}$ and $\{\epsilon, a, \tau\}$ of p and q, according to the notion suggested in [29], are different. We now present another notion of compression content of the traces of a process as follows.

Definition 2.9 *(***Run***). A sequence of (connected) transitions* $e = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots p_{n-1} \xrightarrow{a_n} p_n$ *for* $n \in \mathbb{N}$ *in an LTS* $(\mathbb{P}, \rightarrow)$ *is called a* **run** *of* p_0*. We define that* $\texttt{last}(e) = p_n$*. Let* $\texttt{run}(p)$ *be the set of runs of process* $p \in \mathbb{P}$*, and let* $\leq$ *be the prefix ordering on runs.*

Definition 2.10 *(**Pre-run**). Let e be a run of a process p. Then the* **pre-run function** $\mathtt{pre} : \mathtt{run}(p) \to \mathtt{run}(p)$ *is defined as* $\mathtt{pre}(e) = e'$ *if* $e = e' \xrightarrow{a} q$ *for some process q with $a \neq \tau$, and* $\mathtt{pre}(e) = e$ *otherwise.*

Definition 2.11 *(**Compression content**). Let e be a run and p be a state. The compression content $C(e)$ is a string in $(\mathtt{Act} \setminus \{\tau\})^\star \cup (\mathtt{Act} \setminus \{\tau\})^\star\tau$ and is defined as follows (we use ϵ for the empty string, and $\leq$ for the prefix ordering on strings).*

$$\begin{array}{lcl} C(p) & = & \tau \text{ if } p \xrightarrow{\tau}, \\ C(p) & = & \epsilon \text{ otherwise}, \\ C(p \xrightarrow{\tau} e) & = & C(e), \\ C(p \xrightarrow{a} e) & = & aC(e) \text{ for } a \neq \tau. \end{array}$$

By Definition 2.11, processes p and q in Figure 2.2 have the same set $\{a, \tau\}$ of compression contents.

Theorem 2.12 *Let $(\mathbb{P}, \to)$ be an LTS. Two processes $p, q \in \mathbb{P}$ have the same compression structure if and only if they are orthogonally bisimilar.*

This theorem follows from Proposition 2.6, Proposition 2.14 and Proposition 2.16. To prove Proposition 2.14, we use the following lemma.

Lemma 2.13 *Let $(\mathbb{P}, \to)$ be an LTS, and $p, q \in \mathbb{P}$. Let R be a compression relation between p and q. Then for all $e \in \mathtt{run}(p)$ and $f, f_1, f_2 \in \mathtt{run}(q)$:*

$$eRf_1 \leq f \leq f_2R^{-1}e \Rightarrow eRf.$$

Proof: Since $f \leq f_2R^{-1}e$, it follows from Definition 2.2 that there exist $e' \leq e$ and $e'Rf$. Suppose that $g \leq e$ and gRf, and there does not exist $g' \neq g$, $g' \leq g$ and $g'Rf$. Since $g \leq eRf_1$, there exists $h \leq f_1$ such that gRh. Since $f_1 \leq fR^{-1}g$, there exists $g_1 \leq g$ and g_1Rf_1. Since $h \leq f_1R^{-1}g_1$, there exists $g_2 \leq g_1$ and g_2Rh. R is an equivalence, hence, g_2Rf. Moreover, $g_2 \leq g_1 \leq g$. This implies that $g_2 = g_1 = g$. Therefore, g_1Rf. Since $eRf_1R^{-1}g_1$, eRf. □

Proposition 2.14 *Let $(\mathbb{P}, \to)$ be an LTS, and $p, q \in \mathbb{P}$. If there exists a compression relation between p and q then $p \leftrightarrows_o q$.*

Proof: Let R be a compression relation between p and q. We define a binary relation $\mathcal{B}$ on processes as follows. For all runs e, f, if eRf then $\mathtt{last}(e)\mathcal{B}\mathtt{last}(f)$. It is not hard to see that $p\mathcal{B}q$. We will show that $\mathcal{B}$ is an orthogonal bisimulation. Let $s = \mathtt{last}(e)$, $t = \mathtt{last}(f)$. We prove that

1. if $s \xrightarrow{a} s'$ with $a \neq \tau$ then $t \xrightarrow{a} t'$ and $s' \mathcal{B} t'$. Let $e' = e \xrightarrow{a} s'$. Then e is the pre-run of e'. Therefore, there exists $f' R^{-1} e'$, $f = \texttt{pre}(f')$. Since $C(f) = C(e)$ and $C(f') = C(e')$, $t \xrightarrow{a} t'$. Moreover, $s' \mathcal{B} t'$.

2. if $s \xrightarrow{\tau} s'$ for some s', then $t \xrightarrow{\tau}$, and there is a path $t_0 \cdots t_n \in \tau\text{-}\texttt{paths}(t)$ with $n \geq 0$ such that $s' \mathcal{B} t_n$ and $s \mathcal{B} t_i$ for all $i < n$. Since $s \xrightarrow{\tau} s'$, $C(f) = C(e) = \alpha\tau$ for some string α. Therefore, $t \xrightarrow{\tau}$.

   Now, let $e' = e \xrightarrow{\tau} s'$. Since eRf and $e \leq e'$, it follows from Definition 2.2 that there exists h such that $f \leq h$ and $e' R h$. Let f' be a run of q such that $e' R f'$, $f \leq f'$ and there does not exist $h \neq f'$ such that $f \leq h \leq f'$ and $e' R h'$. We prove that for all $h \neq f'$ such that $f \leq h \leq f'$, eRh. Since $h \leq f' R^{-1} e'$, there exists $g \leq e'$ such that gRh. If $g = e'$ then $e' R h$. This contradicts the assumption of f'. Thus, $g \leq e$. Since eRf, there exists $h' \leq f$ such that gRh'. By Lemma 2.13, gRf. This implies that eRh. Let $f_0 = f, f_1, \ldots, f_{n-1}$ be all the runs h such that $f_i \leq f_{i+1}$ for all $i < n$, and $f_n = f'$. Let $t_i = \texttt{last}(f_i)$ for all $0 \leq i \leq n$. Then for all $0 \leq i < n$, $s \mathcal{B} t_i$ and $s' \mathcal{B} t_n$. Furthermore, the path $t_0 \cdots t_n \in \tau\text{-}\texttt{paths}(t)$ because of the fact that $C(f') = C(e') \leq C(e) = C(f)$.

Similarly, if $t \xrightarrow{a} t'$ with $a \neq \tau$ then $s \xrightarrow{a} s'$ and $s' \mathcal{B} t'$. If $t \xrightarrow{\tau} t'$ for some t', then $s \xrightarrow{\tau}$, and there is a path $s_0 \cdots s_n \in \tau\text{-}\texttt{paths}(s)$ with $n \geq 0$ such that $s_n \mathcal{B} t'$ and $s_i \mathcal{B} t$ for all $i < n$. □

In order to prove Proposition 2.16, we provide the following lemma.

Lemma 2.15 *If $\mathcal{B}$ is an orthogonal bisimulation with $p \mathcal{B} q$, and there is a path $p_0 \cdots p_n$ with $p = p_0$ for some $n \geq 0$, then there is, for every $i \leq n$, an $m_i \geq 0$, such that q has a path with $q_0^0 = q$ and $m_n = 0$ and*

1. *for all $i < n$, if $p_i \xrightarrow{\tau} p_{i+1}$ then $q_i^0 \cdots q_i^{m_i} \in \tau\text{-}\texttt{paths}(q_i^0)$ and $q_i^{m_i} = q_{i+1}^0$,*
2. *for all $i < n$, if $p_i \xrightarrow{a} p_{i+1}$ then $q_i^0 \xrightarrow{a} q_{i+1}^0$, $m_i = 1$ and $q_i^1 = q_{i+1}^0$,*
3. *for all $i \leq n$, if $j < m_i$ or $j = 0$, then $q_i^j \mathcal{B} p_i$.*

Proof: Straightforward by induction on n. □

Proposition 2.16 *Let $(\mathbb{P}, \rightarrow)$ be an LTS. If two processes $p, q \in \mathbb{P}$ are orthogonally bisimilar then there exists a compression relation between p and q.*

Proof: Let $\mathcal{B}$ be an orthogonal bisimulation between p and q. Let R be a binary relation between runs of p and q defined as follows. For all runs $e \in \texttt{run}(p)$ and $f \in \texttt{run}(q)$ formed as $p_0 \cdots p_n$ and $q_0^0 \cdots q_i^j \cdots q_n^0$ in Lemma 2.15, $(e, f) \in R$. By Lemma 2.15, for each $e \in \texttt{run}(p)$ there is a run $f \in \texttt{run}(q)$, and vice versa. Thus, the domain of R is $\texttt{run}(p)$ and the range of R is $\texttt{run}(q)$. We prove that:

1. $eRf \Rightarrow C(e) = C(f)$;

2. $\exists f(eRf \leq f') \Leftarrow e \leq e'Rf'$;
3. $eRf \leq f' \Rightarrow e \leq e'Rf'$;
4. $\mathtt{pre}(f') = f \wedge eRf \Rightarrow \exists e'(\mathtt{pre}(e') = e \wedge e'Rf')$.

Property (1) and Property (2) follow from the definition of R. Property (3) and Property (4) follow from Lemma 2.15 and the definition of R. It follows from Definition 2.4 that R is a compression relation. □

2.4 Conclusion

In this chapter, we have defined a compression structure of processes and a compression relation that identifies processes having the same compression structure. Furthermore, we have proven that in a labeled transition system, two processes have the same compression structure if and only if they are orthogonally bisimilar. Our definition of the compression structure of a process is thereby a solution to the open question in [29].

We note that our definition of compression content (which is a part of compression structure) is different from the one suggested in [29]. Moreover, we use an extra function `pre` to define the compression structure of a process. It remains an open question whether an alternative definition of the compression structure exists (can be found).

Chapter 3

Deciding orthogonal bisimulation

3.1 Introduction

Branching bisimulation equivalence proposed by van Glabbeek and Weijland [52] is a well-known and elegant equivalence in concurrency theory. This equivalence resembles, but is finer than the standard *observation equivalence* of Milner [73].

In 2003, Bergstra, Ponse and van der Zwaag [29] introduced the notion of *orthogonal bisimulation equivalence* on labeled transition systems. Orthogonal bisimulation is a refinement of branching bisimulation in which consecutive τ-actions (silent steps) can be compressed into one (but not zero) τ-action. This is a major difference with branching bisimulation equivalence and other coarser semantics dealing with *abstraction* such as observation equivalence, delay bisimulation equivalence [73, 74] and η-bisimulation equivalence [7].

The main advantage of orthogonal bisimulation, compared to branching bisimulation, is that it combines well with priorities [29]. Moreover, it has the following nice properties:

1. There is a modal logic based on Hennessy-Milner logic [57] which characterizes orthogonal bisimulation equivalence [29].

2. On closed terms in the setting of ACP (Algebra of Communicating Processes) with abstraction, orthogonal bisimulation congruence is completely axiomatized by three laws:

$$\begin{array}{rcll} x\tau\tau & = & x\tau & \\ x\tau(y+z) & = & x(y+z) & \text{if} \quad \tau y = \tau\tau y,\ \tau z = \tau\tau z \\ x(\tau(y+z)+z) & = & x(y+z) & \text{if} \quad \tau y = \tau\tau y \end{array}$$

   We note that unlike in branching bisimulation equivalence, the axiom $x\tau = x$ is not sound in orthogonal bisimulation equivalence.

3. A trace characterization of orthogonal bisimulation equivalence, called the *compression structure of a process*, is provided in [103]. The compression structure characterizes orthogonal bisimilarity in the same way as the *branching structure* characterizes branching bisimilarity in [51].

A commonly used algorithm to analyze the complexity of branching bisimilarity on finite labeled transition systems is presented by Groote and Vaandrager [54]. This algorithm solves the *Relation Coarsest Partition with Stuttering problem* (RCPS) which is closely related to the *Relational Coarsest Partition problem* (RCP) [78]. It is shown in [54] that the algorithm for RCPS can be easily transformed to an $O(n(m+n))$ algorithm for deciding stuttering equivalence on finite Kripke structures [38] and deciding branching bisimulation equivalence on finite labeled transition systems.

In this chapter we take a step towards a theoretical foundation of orthogonal bisimulation by presenting an algorithm for deciding orthogonal bisimulation equivalence on finite labeled transition systems. This problem has been raised in [29]. Our approach is based on the work of Groote and Vaandrager. More precisely, the algorithm in this chapter solves a generalization of the RCPS problem called *Relational Coarsest Partition with Stuttering problem characterizing Orthogonal bisimulation* (RCPSO), and therefore, can be used to decide orthogonal bisimulation. We note that in the Groote-Vaandrager algorithm, the authors perform a preprocessing step by eliminating the presence of cycles of silent steps. This is possible since if two states of a labeled transition system are strongly connected by silent steps, they are branching bisimilar. In the case of orthogonal bisimulation, we cannot eliminate the presence of cycles of silent steps. However, we show that the complexity of our algorithm remains the same as that of Groote and Vaandrager's algorithm. Thus, if n is the number of states and m the number of transitions, it takes $O(n(m+n))$ time and $O(n+m)$ space for deciding orthogonal bisimulation.

The structure of this chapter is as follows. Section 3.2 recalls from [29] the definition of orthogonal bisimulation equivalence. Section 3.3 presents the RCPSO problem, and an algorithm to solve it. We show that this algorithm can be used to decide orthogonal bisimilarity. The chapter is concluded with some remarks in Section 3.4.

3.2 Labeled transition systems and orthogonal bisimulation

In this section, we recall the definitions of labeled transition systems and orthogonal bisimulation from [29].

Definition 3.1 *A* **labeled transition system** *(LTS) is a pair $(S, \rightarrow)$ with S a set of* **processes** *(or* **states***), and $\rightarrow \subseteq S \times A \times S$ for a set A of* **actions** *(or* **labels***) containing the* **silent step** *τ. A triple $(s,a,r) \in \rightarrow$ is called a* **transition**.

An LTS is called **finite** *if both S and A are finite.*

We write $s \xrightarrow{a} r$ for $(s,a,r) \in \rightarrow$, $s \xrightarrow{a}$ for $\exists r \in S : s \xrightarrow{a} r$, and $s \stackrel{\tau}{\Rightarrow} s'$ if there is a sequence $s_0 \ldots s_n$ of states with $s_0 = s$, $s_n = s'$, $n \geq 0$, and $s_i \xrightarrow{\tau} s_{i+1}$ for all $i < n$.

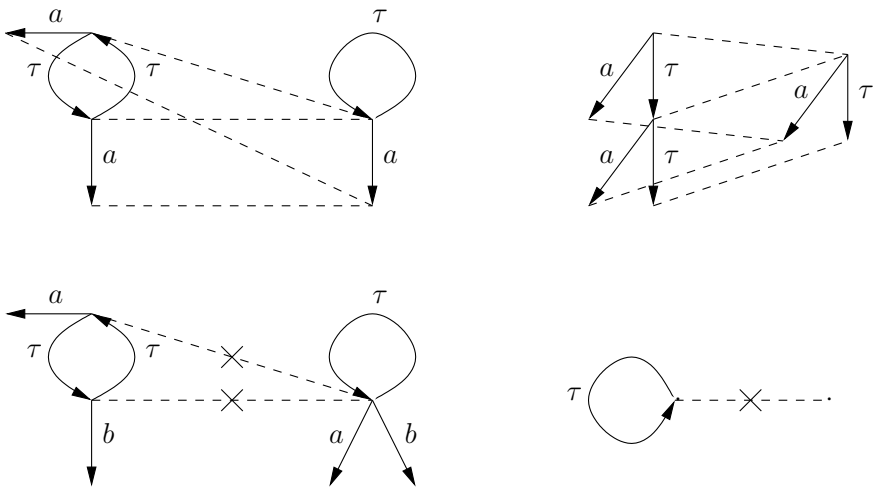

Figure 3.1: Examples of orthogonal bisimulation. Here the dashed lines represent orthogonal bisimulation between processes.

We note that $s \stackrel{\tau}{\Rightarrow} s$ for all states $s \in S$. This is the case when $n = 0$.

Let τ-$\mathtt{paths}(s)$ be the set that consists of all sequences $s_0 \dots s_n$ of states with $s_0 = s$, $n \geq 0$, and $s_i \stackrel{\tau}{\rightarrow} s_{i+1}$ for all $i < n$.

Definition 3.2 *(***Orthogonal bisimulation***). Let $(S, \rightarrow)$ be an LTS. An* **orthogonal bisimulation** *is a binary symmetric relation $\mathcal{B} \subseteq S \times S$ satisfying that for all states $s, r \in S$:*

1. *if $s\mathcal{B}r$ and $s \stackrel{a}{\rightarrow} s'$ for some s' and $a \neq \tau$, then $r \stackrel{a}{\rightarrow} r'$ for some r' with $s'\mathcal{B}r'$; and*

2. *if $s\mathcal{B}r$ and $s \stackrel{\tau}{\rightarrow} s'$ for some s', then $r \stackrel{\tau}{\rightarrow}$, and there is a path $r_0 \cdots r_n \in \tau$-$\mathtt{paths}(r)$ with $n \geq 0$ such that $s'\mathcal{B}r_n$ and $s\mathcal{B}r_i$ for all $i < n$.*

Two states $s, r \in S$ are **orthogonally bisimilar***, denoted by $s \leftrightarrows_o r$, if there exists an orthogonal bisimulation $\mathcal{B}$ such that $s\mathcal{B}r$.*

According to Definition 3.2, a state with a τ-outgoing transition will never be orthogonally bisimilar to a state without τ-outgoing transitions. Furthermore, the states of a cycle of silent steps are not orthogonally bisimilar in most cases. This is the reason why we cannot perform a preprocessing by eliminating cycles of silent steps as in [54]. Examples of orthogonal bisimulation are illustrated in Figure 3.1.

3.3 An efficient algorithm for deciding orthogonal bisimulation

In this section, we generalize the RCPS problem to the RCPSO problem that characterizes orthogonal bisimulation. Next, we will present an algorithm based on the algorithm in [54] to solve RCPSO. We also show that this algorithm can be used for deciding orthogonal bisimulation.

3.3.1 The RCPSO problem

We recall the definition of *partition* from [78, 54] to describe RCPSO.

Definition 3.3 *Let S be a set. A collection $\{B_i | i \in I\}$ of nonempty subsets of S is called a* **partition** *of S if $\cup_{i \in I} B_i = S$ and for $i \neq j : B_i \cap B_j = \emptyset$. The elements of a partition are called* **blocks**. *If P and P' are partitions of S then P'* **refines** *P (P is* **coarser** *than P') if any block of P' is included in a block of P. The* **equivalence** *$\sim_P$ on S induced by a partition P is defined by: $r \sim_P s$ if and only if $\exists B \in P$: $r \in B$ and $s \in B$.*

The *Relational Coarsest Partition with Stuttering problem characterizing Orthogonal bisimulation* (RCPSO) can be specified as follows:
Given: a nonempty, finite set S of states, a relation $\rightarrow \subseteq S \times A \times S$ of *transitions* and an *initial partition* P_0 of S.
Find: the coarsest partition P_f satisfying:

1. P_f refines P_0;
2. if $s \sim_{P_f} r$ and $s \xrightarrow{a} s'$ with $a \neq \tau$, then there exists $r' \in S$ such that $r \xrightarrow{a} r'$ and $s' \sim_{P_f} r'$;
3. if $s \sim_{P_f} r$ and $s \xrightarrow{\tau} s'$, then there is an $n \geq 0$ and there are $r_0, \ldots, r_n \in S$ such that:
   (a) $r_0 = r$;
   (b) for all $0 \leq i < n$: $s \sim_{P_f} r_i$ and $r_i \xrightarrow{\tau} r_{i+1}$;
   (c) $s' \sim_{P_f} r_n$.

To decide orthogonal bisimulation, it is essential to start with a partition P_0 in which states with an outgoing τ-transition have been separated from states without an outgoing τ-transition. This agrees with orthogonal bisimulation equivalence.

3.3.2 The algorithm

This section describes an algorithm to solve the RCPSO problem. The algorithm is based on the algorithm for deciding branching bisimulation of Groote and Vaandrager [54], where transition systems might contain cycles of silent steps.

Let $|S| = n$ and $|\rightarrow| = m$. For blocks $B, B' \subseteq S$ we define $\mathtt{pos}_a(B, B')$ with $a \neq \tau$ as the set of states in B from which a state in B' can be reached by an *observable* action a. Furthermore, $\mathtt{pos}_\tau(B, B')$ is the set of states in B from which a state in B' can be reached by a sequence of silent steps τ.

$$\begin{aligned}\mathtt{pos}_a(B, B') &= \{s \in B | \exists s' \in B' : s \xrightarrow{a} s'\} \text{ for } a \neq \tau,\\ \mathtt{pos}_\tau(B, B') &= \{s \in B | \exists n \geq 0 \exists s_0, \ldots, s_n : s_0 = s,\\ &\qquad \forall i < n : s_i \in B \wedge s_i \xrightarrow{\tau} s_{i+1} \text{ and } s_n \in B'\}.\end{aligned}$$

Definition 3.4 *We say that a block B' is a* **splitter** *of a block B* **with respect to** *a if and only if:*

1. *$B \neq B'$ or $a \neq \tau$, and*
2. *$\emptyset \neq \mathtt{pos}_a(B, B') \neq B$.*

We note that Clause 1 in Definition 3.4 implies that in case $a = \tau$, a block B cannot be a splitter of itself.

If P is a partition of S and a block B' is a splitter of a block B with respect to a, then $\mathtt{Ref}_P^a(B, B')$ is the partition P where B is replaced by $\mathtt{pos}_a(B, B')$ and $B \setminus \mathtt{pos}_a(B, B')$.

Definition 3.5 *A partition P is* **stable with respect to a block** *B' if for no block B of P and for no action a, B' is a splitter of B. P is* **stable** *if it is stable with respect to all its blocks.*

The algorithm maintains a partition P that is initially P_0. It repeats the following steps until P is stable:

1. find blocks B, B' of P and a label $a \in A$ such that B' is a splitter of B with respect to a;
2. $P := \mathtt{Ref}_P^a(B, B')$.

Theorem 3.6 *The above algorithm for the RCPSO problem terminates after at most $n - |P_0|$ refinement steps. The resulting partition P_f is the coarsest stable partition refining P_0.*

Proof: Sketch based on the proof of Theorem 3.1 in [54]. At each iteration of the refinement step, if we cannot find blocks B, B' of the current partition P and a label $a \in A$ such that B' is a splitter of B with respect to a then we know that the current partition is stable, and that the algorithm terminates. Otherwise, the number of blocks increases by one. Thus, termination will occur after at most $n - |P_0|$ iterations. Next, we show that the resulting partition P_f is the coarsest stable partition refining P_0. We prove by induction on the number of refinement steps that any stable partition refining P_0 is also a refinement of the current partition P. Clearly the statement holds

initially. Let R be a stable refinement of P_0. By the induction hypothesis, R is a refinement of P. Let Q be a refinement of P after a refinement step, using a splitting pair (B, B') with respect to a. We show that R is also a refinement of Q. Let C be a block in R. Then C is included in a block D of P. We prove that C is included in a block of Q. If $D \neq B$ then we are done. In the case $D = B$, we show that either $C \subseteq \mathtt{pos}_a(B, B')$ or $C \subseteq B \setminus \mathtt{pos}_a(B, B')$. Suppose that there are $r, s \in C$ with $s \in \mathtt{pos}_a(B, B')$ and $r \notin \mathtt{pos}_a(B, B')$. There are two cases:

1. $a \neq \tau$. There exists $s' \in B'$ such that $s \xrightarrow{a} s'$. Let C' be a block in R such that $s' \in C'$. Thus, $C' \subseteq B'$. Since R is a stable refinement of P_0 and $r, s \in C$, there exists $r' \in C' \subseteq B'$ such that $r \xrightarrow{a} r'$. This contradicts $r \notin \mathtt{pos}_a(B, B')$.

2. $a = \tau$. There are $s_0 = s, \ldots, s_n$ such that for all $i < n$: $s_i \in B$, $s_i \xrightarrow{\tau} s_{i+1}$ and $s_n \in B'$. Let $C_0 = C, \ldots, C_n$ be the blocks of R such that $s_i \in C_i$. Since R is a refinement of P and $s_n \in C_n \cap B'$ and for all $i < n$: $s_i \in C_i \cap B$, $C_i \subseteq B$ and $C_n \subseteq B'$. Since $s, r \in C$, there is a sequence $r_0, \ldots, r_m$ with $r_0 = r$, for all $i < m$, $r_i \in B$ and $r_i \xrightarrow{\tau} r_{i+1}$, and $r_m \in B'$. This contradicts $r \notin \mathtt{pos}_\tau(B, B')$.

Therefore, P_f is the coarsest stable partition refining P_0. □

We now describe how one can find in $O(m)$ time a splitter of the current partition, or find in $O(m)$ time that no such splitter exists. Furthermore, if a splitter has been found, it takes $O(m+n)$ time to refine the current partition. We will use the following definitions and lemmas.

Definition 3.7 *Let P be a partition of S. A transition $s \xrightarrow{a} s'$ is called (P-)***inert** *if $s \sim_P s'$ and $a = \tau$. A transition is* **non-inert** *if it is not an inert transition.*

Definition 3.8 *Let P be a partition of S. A (P-)***inert component** *is a maximal subset $C \subseteq S$ such that for arbitrary states $s, s' \in C$ where $s \neq s'$ there is a path of inert transitions from s to s', and vice versa. Let B be a block of P such that $C \subseteq B \subseteq S$. We say that C is an inert component of B. An inert component C of a block B is a* **terminal component** *of B if there is no inert component C' of B with $C' \neq C$ such that $s \xrightarrow{\tau} s'$ for some $s \in C$ and $s' \in C'$. An inert component of a block is called a* **non-terminal component** *if it is not a terminal component of that block.*

Note that an inert component can contain only one state (for example a state that is not connected by a τ-transition).

Example 3.9 Let B be a block consisting of states $s_0, s_1, s_2, s_3, s_4, s_5$, and a state $s_6 \notin B$ as illustrated in Figure 3.2. Then the sets $C_1 = \{s_0, s_1, s_2, s_3\}$ and $C_2 = \{s_4, s_5\}$ are two inert components of B. More precisely, C_1 is a non-terminal component, while C_2 is a terminal component of B.

For a state s, an inert component C, blocks B, B' and an action a, we write $s \xrightarrow{a} B$ if there is a state $s' \in B$ such that $s \xrightarrow{a} s'$ otherwise $s \xrightarrow{a}\!\!\!\!\!\!\rightarrow B$. Moreover, we denote $C \xrightarrow{a} B$ if there are states $s \in C$ and $s' \in B$ such that $s \xrightarrow{a} s'$ otherwise $C \xrightarrow{a}\!\!\!\!\!\!\rightarrow B$, and $B \xrightarrow{a} B'$ if there are states $s \in B$ and $s' \in B'$ such that $s \xrightarrow{a} s'$ otherwise $B \xrightarrow{a}\!\!\!\!\!\!\rightarrow B'$.

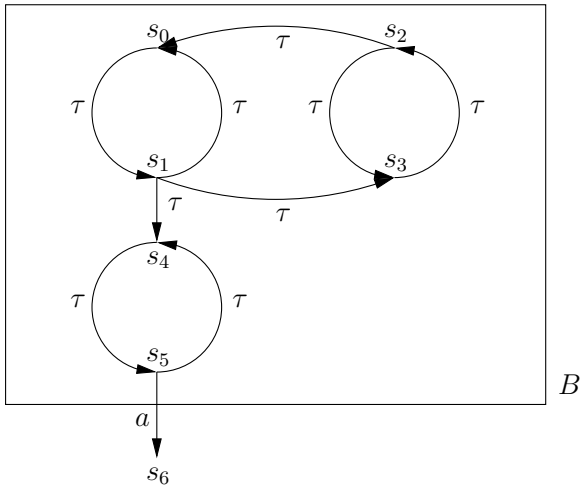

Figure 3.2: An example of inert components.

Lemma 3.10 *Let P be a refinement of P_0 and let $B, B' \in P$ and $a \in A$. Then B' is a splitter of B with respect to a if and only if:*

1. *$B \neq B'$ or $a \neq \tau$;*
2. *$B \xrightarrow{a} B'$;*
3. *if $a \neq \tau$ then there exists $s \in B$ such that $s \stackrel{a}{\nrightarrow} B'$;*
4. *if $a = \tau$ then there exists a terminal component C of B such that $C \stackrel{\tau}{\nrightarrow} B'$.*

Proof:

1. $\Rightarrow$: Suppose B' is a splitter of B. Clause 1 follows from Definition 3.4. Since $\mathtt{pos}_a(B, B') \neq \emptyset$ and $B \neq B'$ if $a = \tau$, $B \xrightarrow{a} B'$. Clause 3 follows from the fact that $\mathtt{pos}_a(B, B') \neq B$. To prove Clause 4, we assume that for every terminal component C of B there is a state $s' \in B'$ such that $s \xrightarrow{\tau} s'$ for some $s \in C$. It follows from Definition 3.8 that every state in B can lead to a state in B' by a sequence of τ transitions. Thus, $\mathtt{pos}_a(B, B') = B$. This is a contradiction to the fact that B' is a splitter of B.
2. $\Leftarrow$: Suppose that B and B' satisfy Clause 1, 2, 3 and 4. Then B' is a splitter of B because:
   (a) $\emptyset \neq \mathtt{pos}_a(B, B')$ because of Clause 2.
   (b) $\mathtt{pos}_a(B, B') \neq B$ because of Clause 3 and Clause 4.

□

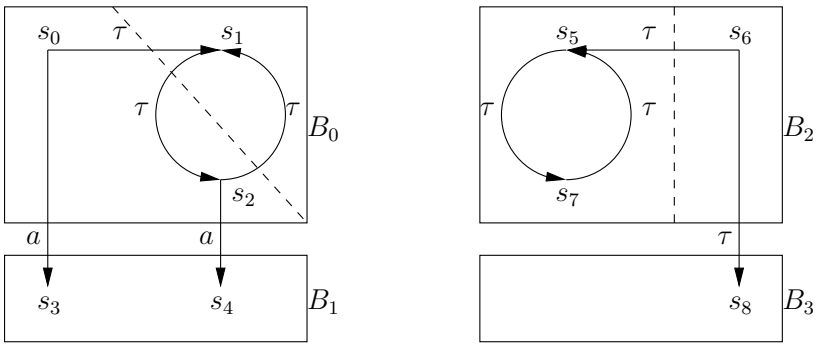

Figure 3.3: An example of splitting.

Example 3.11 Let B_0, B_1, B_2 and B_3 be the blocks given in Figure 3.3. Then B_1 is a splitter of B_0 with respect to a, and B_3 is a splitter of B_2 with respect to τ. More precisely, $\mathtt{pos}_a(B_0, B_1) = \{s_0, s_2\}$ and $\mathtt{pos}_\tau(B_2, B_3) = \{s_6\}$.

We note that in the case of branching bisimulation, each state is an inert component since the initial P_0 does not have cycles of τ-transitions. Therefore, instead of dealing with terminal-components, one has to deal with *bottom-states* only. This is the main difference between the Groote-Vaandrager algorithm and our algorithm. Moreover, while the initial partition of Groote-Vaandrager consists of a single block containing all states, our initial partition will consist of two blocks: one block of states that can perform a τ-transition, and one block of states that cannot perform a τ-transition.

Lemma 3.12 *Let P and R be partitions such that R refines P, and P and R have the same inert transitions. Let B' be a block of both P and R such that P is stable with respect to B'. Then R is stable with respect to B'.*

Proof: We prove this lemma by contradiction. Suppose that there exists a block B of R and an action a such that B' is a splitter of B with respect to a. There are two cases:

1. $a \neq \tau$. By Lemma 3.10, there exists a transition $r \xrightarrow{a} r'$ with $r \in B$, $r' \in B'$, and a state $s \in B$ such that for no $s' \in B'$: $s \xrightarrow{a} s'$. Since R refines P, B is included in a block B'' of P. Thus, $r, s \in B''$. By Lemma 3.10, B' is a splitter of B''. This contradicts the fact that P is stable with respect to B'.

2. $a = \tau$. Then $B \neq B'$. By Lemma 3.10, there is a terminal component C of B such that for no $s' \in B'$: $s \xrightarrow{\tau} s'$ for some $s \in C$. Since R refines P, B is included in a block B'' of P. Thus C is included in an inert component C'' of B''. We prove that C'' is also a terminal component of B''. Suppose that C'' is not a terminal component of B''. Then there exists an inert component K with $K \neq C''$, and an inert transition $r \xrightarrow{\tau} r'$ of B'' with $r \in C''$ and $r' \in K$.

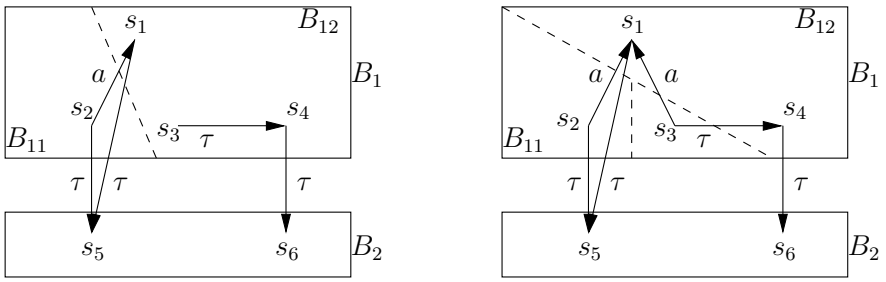

Figure 3.4: An example of stability (a) and unstability (b).

Let $p \in C$. Then $p \in C''$. Thus there is a path $p_0 \dots p_n$ of states such that $p_0 = p$ and $p_n = r$ with $p_i \sim_P p_{i+1}$ for all $i < n$. Since P and R have the same inert transitions, $p_i \sim_R p_{i+1}$ for all $i < n$, and $r \sim_R r'$. It follows from Definition 3.3 and $p \in B$ that $r' \in B$. Since C is a terminal component of B and Definition 3.8, $r' \in C$. Thus, $r' \in C''$. This contradicts the fact that $r' \in K$.

□

Example 3.13 Let $(S, \rightarrow)$ be the LTS illustrated in Figure 3.4(a).

$$\begin{array}{lcl} S & = & \{s_1, s_2, s_3, s_4, s_5, s_6\} \text{ and} \\ \rightarrow & = & \{s_1 \xrightarrow{\tau} s_5, s_2 \xrightarrow{a} s_1, s_2 \xrightarrow{\tau} s_5, s_3 \xrightarrow{\tau} s_4, s_4 \xrightarrow{\tau} s_6\}. \end{array}$$

Let $B_1 = \{s_1, s_2, s_3, s_4\}$, $B_2 = \{s_5, s_6\}$ and $P = \{B_1, B_2\}$. It is obvious that P is stable with respect to B_2. Moreover, B_1 is a splitter of itself with respect to a. We split B_1 into $B_{11} = \{s_2\}$ and $B_{12} = \{s_1, s_3, s_4\}$. Let R be the refinement, $R = \{B_{11}, B_{12}, B_2\}$. Then P and R have the same inert transition $s_3 \xrightarrow{\tau} s_4$. Therefore, R is also stable with respect to B_2.

We now extend $(S, \rightarrow)$ with a transition $s_3 \xrightarrow{a} s_1$ (see Figure 3.4(b)). The partition $P = \{B_1, B_2\}$ is still stable with respect to B_2. Furthermore, B_1 is also a splitter of itself with respect to a. However, after the splitting of B_1 into $B_{11} = \{s_2, s_3\}$ and $B_{12} = \{s_1, s_4\}$, the inert transition $s_3 \xrightarrow{\tau} s_4$ becomes non-inert. The refinement R is no longer stable with respect to B_2 as B_2 can be used as a splitter of B_{11} with respect to τ ($\mathtt{pos}_\tau(B_{11}, B_2) = \{s_2\}$).

Given an LTS, the data structure for an implementation for solving the RCPSO problem is initialized as follows, where we identify a block, a component and a state with a record representing it (transitions are represented indirectly):

- There are two lists of blocks `tobeprocessed` and `stable`. A block B' is in `stable` if the current partition is stable with respect to B', otherwise B' is in `tobeprocessed`. Initially, all blocks in P_0 are in the list `tobeprocessed`.

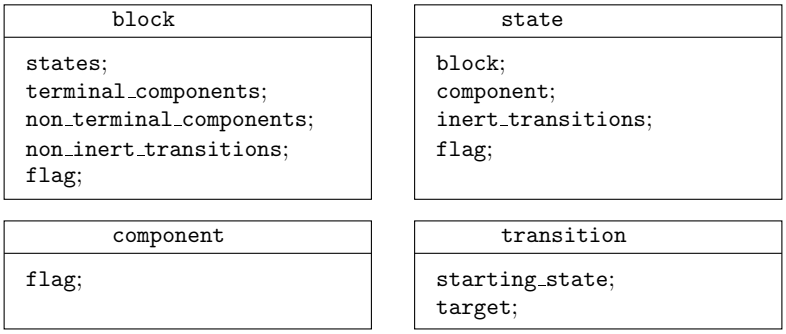

Figure 3.5: Data structures for solving RCPSO.

- Each state contains two pointers `block` and `component` to the block and the inert component of which it is an element, and a list `inert_transitions` of inert transitions ending in this state (see Figure 3.5).
- Each block B contains a list `states` of states in B.

  Furthermore, it has a list `terminal_components` of terminal components in B and a list `non_terminal_components` of non-terminal components in B. Finally, it points to a list `non_inert_transitions` of groups of non-inert transitions that end in B. More precisely, all transitions with the same label are in subsequent records in the list. To compute the lists of terminal and non-terminal components of all blocks in the initial partition, one can apply a variant of the standard depth first search algorithm [2] using $O(m + n)$ time and space. In addition, grouping of the non-inert transitions has a complexity $O(|A|+m)$ (bucket sort).
- Each transition contains two pointers `starting_state` and `target`: one to its starting state, and one to its target.
- Each state, each component and each block has an auxiliary field `flag` of type boolean, which is 0 (standing for `false`) initially.
- Moreover, there are two auxiliary booleans `found_a_splitter` and `inert_becomes_non_inert`, and an auxiliary list `BL`. Initially, `found_a_splitter` = `false`, `inert_becomes_non_inert` = `false` and `BL` = $\emptyset$. We note that given a block B', the block list `BL` contains all blocks B having a non-inert transition from B to B'.

Note that the transitions of the LTS are either represented in the blocks (the non-inert ones), or in the states (the inert ones).

The implementation of the algorithm for deciding the RCPSO problem is presented in Table 3.1 and Table 3.2.

With reference to Table 3.1, we first explain how to compute in $O(m)$ time whether we can find a splitter of the current partition or decide that no such splitter exists, meaning that the current partition is stable. Let $m^a_{\mathtt{BB}}$, denote the number

```
(1) tobeprocessed = P_0; stable = ∅;
(2) repeat
(3) Let B' = head(tobeprocessed);
(4) // Scan the list L of all non-inert transitions that end in B'
(5)  L = B'.non_inert_transitions;
(6) if  L ≠ ∅ then repeat
(7)    Let T_a = head(L); L = tail(L);
(8)    BL = ∅;
(9)    for all transitions s -a-> s' ∈ T_a do
(10)       if s.block.flag = 0 then s.block.flag = 1; insert(s.block, BL); end if;
(11)       case a ≠ τ: s.flag = 1;
(12)       case a = τ: s.component.flag = 1;
(13)       end case;
(14)   end for;
(15)   repeat
(16)       Let B = head(BL); BL = tail(BL);
(17)       case a ≠ τ:
(18)           if there is a state s ∈ B such that s.flag = 0 then
(19)               found_a_splitter = true;
(20)       case a = τ:
(21)           if there is a terminal-component C ∈ B such that C.flag = 0 then
(22)               found_a_splitter = true;
(23)       end case;
(24)   until found_a_splitter or BL = ∅;
(25)   if not found_a_splitter then Reset all flags;
(26) until found_a_splitter or L = ∅;
(27) end if ;
(28) if found_a_splitter then
(29)   B_1, B_2 = split(B, a);
(30)   tobeprocessed = remove(B, tobeprocessed);
(31)   tobeprocessed = insert(B_1, insert(B_2, tobeprocessed));
(32)   if  inert_becomes_non_inert then
(33)       tobeprocessed = tobeprocessed ∪ stable; stable = ∅;
(34)   end if;
(35)   Reset all flags;
(36)   found_a_splitter = false; inert_becomes_non_inert = false;
(37) else
(38)   tobeprocessed = remove(B', tobeprocessed); stable = insert(B', stable);
(39) end if;
(40) until tobeprocessed = ∅
```

Table 3.1: An algorithm for solving the RCPSO problem. The functions `insert`, `head` and `tail` are standard functions on lists. The function `remove` denotes removal of an element from a list, and the function ∪ denotes concatenation of two lists.

```
split(B, a)
(1)   if a = τ then
(2)       Raise the flag of all states in B that can lead to a state in a
(3)       terminal-component with a raised flag by a path of inert transitions;
(4)   end if;
(5)   B1 = new; B2 = new;
(6)   // Assign the states to B1 and B2
(7)   for all states s ∈ B.states do
(8)         if s.flag = 1 then insert(s, B1.states)
(9)                       else  insert(s, B2.states); end if;
(10)  end for;
(11)  //Compute the list of non-inert transitions of B1 and B2
(12)  for all transitions t ∈ B.non_inert_transitions do
(13)        if t.target ∈ B1 then insert(t, B1.non_inert_transitions);
(14)        else  insert(t, B2.non_inert_transitions);end if;
(15)  end for;
(16)  //Compute the list of inert transitions ending in each state of B1
(17)  for all states s ∈ B1.states do
(18)        for all transitions t ∈ s.inert_transitions do
(19)            if t.starting_state ∉ B1.states then
(20)                inert_becomes_non_inert = true;
(21)                remove(t, s.inert_transitions);
(22)                insert(t, B1.non_inert_transitions);
(23)            end if;
(24)        end for;
(25)  end for;
(26)  //Compute the list of inert transitions ending in each state of B2
(27)  for all states s ∈ B2.states do
(28)        for all transitions t ∈ s.inert_transitions do
(29)            if t.starting_state ∉ B2.states then
(30)                inert_becomes_non_inert = true;
(31)                remove(t, s.inert_transitions);
(32)                insert(t, B2.non_inert_transitions);
(33)        end if;
(34)        end for;
(35)  end for;
(36)  Group non-inert transitions of B1 and B2;
(37)  Compute the lists of terminal and non-terminal components for B1 and B2;
```

Table 3.2: Constructing B_1 and B_2.

of transitions from a block B to a block B' with label a. Let B' be a block in `tobeprocessed`. Scan the list L of groups of non-inert transitions that end in B' (initially, $L = B'$.`non_inert_transitions`). Consider subsequently all groups T_a of non-inert transitions with a label a in L. We set the flag field of the blocks of the starting states of all transitions in T_a, and add these blocks to the list BL. Further-

more, if a is an observable action then we raise the flag of the starting states of all transitions in T_a. In this case, to find out whether B' is a splitter of a block B in `BL`, we only have to check whether the flag of some state in B is not raised. In case $a = \tau$, we raise the flag of the components of the starting states of all transitions in T_τ. To find out whether B' is a splitter of a block B in `BL` with respect to τ, we only have to check whether the flag of some terminal-component in B is not raised. The complexity to find out that B' is a splitter of B with respect to an action a or not is $O(m^a_{\mathtt{BB}'})$. Therefore, the complexity to find a splitter of the current partition (if it exists) is $\Sigma O(m^a_{\mathtt{BB}'})$ or $O(m)$.

In the case we have found that B' is a splitter of a block B in the current partition, we split B into B_1 and B_2 and insert these blocks to the list `tobeprocessed`. By Lemma 3.12, if some inert transition of the current partition becomes a non-inert transition in the new partition then we append the list `stable` to the list `tobeprocessed` and make `stable` empty. If B' is not a splitter in the current partition, then we move B' from the list `tobeprocessed` to the list `stable`, and repeat the same procedure for the next block in `tobeprocessed`. If `tobeprocessed` is empty then we know that the current partition is stable.

With reference to Table 3.2, we now explain how to split B by B' into B_1 and B_2 in $O(m_B + n_B)$ time, where m_B is the number of transitions and n_B the number of states in B. In case a is a τ-action, we raise the flag of all states in B that can lead to a state in a terminal-component with a raised flag by a path of inert transitions. To do this, one can apply a standard depth first search algorithm using $O(m_B + n_B)$ time and space. (Here we use the list of inert transitions ending in each state of B). We now refine B into B_1 and B_2. All states with a raised flag are inserted into B_1, the others are placed in B_2. It is easy to compute the list of non-inert transitions ending in B_1 and B_2, and the lists of inert transitions ending in each state of B_1 and B_2 (line 11-35 of Table 3.2). Since the set of actions is finite, one can apply the bucket sort algorithm [2] to group the non-inert transitions of B_1 and B_2 in $O(m_B)$ time. Finally, one can apply the well-known algorithm for finding strongly connected components in a directed graph [2] using $O(m_B + n_B)$ time and space to compute the lists of terminal and non-terminal components for B_1 and B_2.

Therefore, we can find in $O(m)$ time a splitter of the current partition or find in $O(m)$ time that the current partition is stable. If a splitter is found, it takes $O(m+n)$ time to construct the new partition. Moreover, it is not hard to check that the space complexity of the algorithm above is $O(m + n)$. Thus we have the following theorem:

Theorem 3.14 *The RCPSO problem can be decided in $O(n(m + n))$ time, using $O(m + n)$ space.*

Example 3.15 Let $(S, \rightarrow)$ be the LTS given in Figure 3.6.

$$\begin{aligned} S &= \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\} \text{ and} \\ \rightarrow &= \{s_0 \xrightarrow{\tau} s_1, s_0 \xrightarrow{a} s_4, s_1 \xrightarrow{\tau} s_0, s_1 \xrightarrow{a} s_5, s_2 \xrightarrow{\tau} s_1, s_2 \xrightarrow{b} s_6, s_3 \xrightarrow{\tau} s_2, s_3 \xrightarrow{a} s_7\}. \end{aligned}$$

At the beginning, let $B_0 = \{s_4, s_5, s_6, s_7\}$, $B_1 = \{s_0, s_1, s_2, s_3\}$, and $P_0 = \{B_0, B_1\}$. We find the coarsest partition P_f of P_0 as follows. The block B_0 is a splitter of B_1

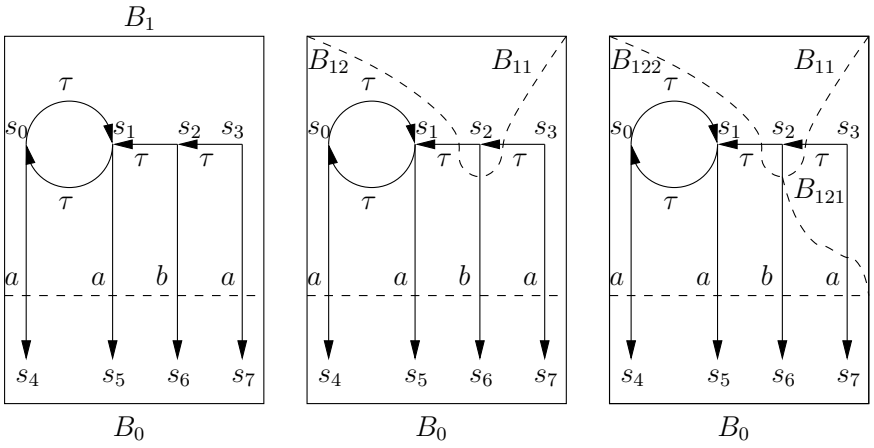

Figure 3.6: An example for solving RCPSO.

with respect to b. Thus, B_1 is split into $B_{11} = \{s_2\}$ and $B_{12} = \{s_0, s_1, s_3\}$ by (B_0, a). Let $P_1 = \{B_0, B_{11}, B_{12}\}$. Then it is easy to see that B_{11} is a splitter of B_{12} with respect to τ. We split B_{12} to $B_{121} = \{s_3\}$ and $B_{122} = \{s_0, s_1\}$. The refinement P_2 of P_1 is stable with respect to all the blocks, and therefore, $P_f = P_2$.

3.3.3 The RCPSO problem can be used to decide orthogonal bisimulation on finite LTS's

To decide orthogonal bisimulation of two states in a finite LTS, we can check whether they are in the same block of the coarsest stable partition P_f in the RCPSO problem with the initial partition P_0 consisting of two blocks: the first block contains all states that have no outgoing τ transitions and the second block contains the remaining states in this LTS. It takes $O(m+n)$ time to construct P_0, using $O(m+n)$ space.

Theorem 3.16 *Let $(S, \rightarrow)$ be a finite LTS, and let P_f be the final partition obtained after applying the RCPSO algorithm on an initial partition P_0 containing two blocks B_1 and B_2, where B_1 consists of all states in S that have no outgoing τ transitions and $B_2 = S \setminus B_1$. Then $\sim_{P_f} = \leftrightarrows_o$.*

Proof: This follows from the following two facts:

1. $\sim_{P_f} \subseteq \leftrightarrows_o$. It follows from Theorem 3.6 that P_f exists. We show that if $s \sim_{P_f} r$ then $s \leftrightarrows_o r$. By the definition of P_f, if $s \xrightarrow{a} s'$ then there exists r' such that $r \xrightarrow{a} r'$ and $s' \sim_{P_f} r'$. In the case that $s \xrightarrow{\tau} s'$, since P_f refines P_0, $r \xrightarrow{\tau} r'$. Moreover, there is an $n \geq 0$ and there are $r_0, \ldots, r_n \in S$ such that $r_0 = r$, for all $0 \leq i < n : s \sim_{P_f} r_i$ and $r_i \xrightarrow{\tau} r_{i+1}$, and $s' \sim_{P_f} r_n$. This implies that P_f is an orthogonal bisimulation equivalence.

2. $P_f \supseteq \leftrightarrows_o$. Orthogonal bisimulation equivalence $\leftrightarrows_o$ induces a stable partition that refines P_0 on S. As P_f is the coarsest stable partition that refines P_0, $\leftrightarrows_o \subseteq P_f$.

□

The complexity for deciding orthogonal bisimulation is $O(n(m+n))$ time, using $O(m+n)$ space.

3.4 Concluding remarks

In this chapter, we have presented an algorithm for deciding orthogonal bisimulation. Our algorithm is based on the well-known algorithm for deciding branching bisimulation given by Groote and Vaandrager in [54]. The difference between the two algorithms is that in our algorithm, transition systems may have cycles of silent steps. This makes the problem addressed in this chapter more complicated. For instance, instead of dealing with states, we have to deal with sets of states called inert components. Nevertheless, we have shown that the complexity of our algorithm remains the same as that of [54]. Thus, it takes $O(n(m+n))$ time to decide orthogonal bisimilarity in finite state transition systems using $O(m+n)$ space. This thereby answers the open question in [29].

Chapter 4

Structural operational semantics for thread algebra

4.1 Introduction

Thread algebra (TA) is a framework for the description and analysis of multi-threaded systems proposed by Bergstra and Middelburg [23, 24, 25]. This semantics is based on *basic thread algebra* (BTA) first introduced under the name *basic polarized process algebra* (BPPA) in [22], an algebraic theory about sequential program behaviors. Thread algebra specifies a collection of strategic interleaving operators that turn a sequence of threads into a single thread capturing the essential aspects of multi-threading. Here strategic interleaving, in contrast with the arbitrary interleaving in other process algebras such as CCS [75] or ACP [19], deterministically determines the ordering of performed actions from various threads. Thread algebra is promising for the study of security related phenomena involving concurrent systems [28, 101]. Therefore, there is a need to study carefully the semantics for thread algebra.

In [104], a metric denotational semantics for BTA is provided. This chapter focuses on *structural operational semantics* (SOS) [81] (see [1] for an overview), a formal semantics of programming and specification languages, for thread algebra. We present a SOS for thread algebra which is less general than the SOS introduced in [23, 25]. However, it is simpler. We show that bisimulation equivalence defined by our SOS characterizes the equality induced by the axioms of thread algebra.

The structure of this chapter is as follows. Section 4.2 recalls the basic concepts of SOS, BTA and TA. Section 4.3 presents transition rules for TA. Section 4.4 shows that bisimulation is a congruence with respect to TA. Hence, it characterizes the equality induced by the axioms of TA. The chapter is ended with some concluding remarks in Section 4.5.

4.2 Preliminaries

This section recalls from [1, 4, 22, 23] the basic concepts of structural operational semantics (SOS), basic thread algebra (BTA) and thread algebra (TA).

4.2.1 Structural operational semantics (SOS)

Structural operational semantics (SOS) generates a *labeled transition system* (LTS) whose *states* are *closed terms* over an algebraic signature, and whose transitions between states are obtained inductively from a collection of *transition rules*, called *transition system specification* (TSS).

Labeled transition systems

Definition 4.1 *A* **labeled transition system** *(LTS) is a quadruple* $(\mathbb{S}, A, \{\xrightarrow{a} | a \in A\}, \mathtt{Pred})$*, satisfying:*

- $\mathbb{S}$ *is a set of states (or threads);*
- A *is a set of actions;*
- $\xrightarrow{a} \subseteq (\mathbb{S} \times \mathbb{S})$ *for every* $a \in A$*;*
- $P \subseteq \mathbb{S}$ *for every* $P \in \mathtt{Pred}$*. We write* pP *if state* p *satisfies predicate* P*.*

Binary relations $s \xrightarrow{a} s'$ *and unary predicates* sP *in an LTS are called transitions. We write* $s \xrightarrow{a} s'$ *for* $(s, s') \in \xrightarrow{a}$*.*

Bisimulation [79] is an important equivalence in process algebras that classifies processes (or threads) behaving identically.

Definition 4.2 *Given an LTS* $(\mathbb{S}, A, \{\xrightarrow{a} | a \in A\}, \mathtt{Pred})$*, a symmetric relation* $\mathcal{B} \subseteq \mathbb{S} \times \mathbb{S}$ *is a* **bisimulation** *if it satisfies:*

1. *If* $(p, q) \in \mathcal{B}$ *and* pP *then* qP *for all* $P \in \mathtt{Pred}$*.*
2. *If* $(p, q) \in \mathcal{B}$ *and* $p \xrightarrow{a} p$ *then there exists* q' *such that* $q \xrightarrow{a} q'$ *and* $(p', q') \in \mathcal{B}$*.*

Two threads p *and* q *are* **bisimilar***, denoted by* $(p \leftrightarroweq q)$*, if there is a bisimulation relation* $\mathcal{B}$ *such that* $(p, q) \in \mathcal{B}$*.*

Term algebras

The states of an LTS can be given as closed terms over some signature. In the following, we will present some basic notions of term algebras.

Let $\mathtt{Var}$ be an infinite set of variables, with typical elements x, y, z. A *signature* is a set $\mathtt{Sig}$ of function symbols f with arity $\mathtt{ar}(f)$. The set $\mathbb{T}(\mathtt{Sig})$ of *terms* is defined as usual. A term is *closed* if it does not contain any variables. Let t, u denote terms and p, q closed terms. A *substitution* is a mapping $\sigma : \mathtt{Var} \to \mathbb{T}(\mathtt{Sig})$. A substitution is *closed* if it maps each variable to a closed term in $\mathbb{T}(\mathtt{Sig})$.

Definition 4.3 *(***Congruence***) Assume a signature* $\mathtt{Sig}$*. An equivalence relation* $\sim$ *over closed terms in* $\mathbb{T}(\mathtt{Sig})$ *is a* **congruence** *if, for all* $f \in \mathtt{Sig}$*,*

$$p_i \sim q_i \text{ for } i = 1, \ldots, \mathtt{ar}(f) \text{ implies } f(p_1, \ldots, p_{\mathtt{ar}(f)}) \sim f(q_1, \ldots, q_{\mathtt{ar}(f)}).$$

Transition system specifications

The transitions between states in an LTS can be generated inductively from a collection of transition rules, called transition system specification (TSS). Given a term algebra, we define a TSS as follows.

Definition 4.4 *(***Transition system specification***) A* **literal** *is an expression* $t \xrightarrow{a} t'$ *or* tP*. A* **transition rule** *is of the form* $\frac{H}{\pi}$*, where* H *is a set of literals called the* **premises***, and* π *is a literal. A rule* $\frac{\emptyset}{\pi}$ *is also written* π*. A* **transition system specification** *(TSS) is a set of transition rules. A transition rule is* **closed** *if it contains only closed terms.*

We note that in the premises in the previous definition are *positive*. We do not consider *negative* premises in this chapter. To define the LTS generated by a TSS, we use the notion of a *proof* of a closed transition rule from a TSS.

Definition 4.5 *A* **proof** *from a TSS* T *of a closed transition rule* $\frac{H}{\pi}$ *consists of an upwardly branching tree in which all upward paths are finite, where the nodes of the tree are labeled by transitions such that:*

- *the root has label* π*, and*
- *if some node has label* l*, and* K *is the set of labels of nodes directly above this node then*
  1. *either* K *is the empty set and* $l \in H$*,*
  2. *or* $\frac{K}{l}$ *is a closed substitution instance of a transition rule in* T*.*

Definition 4.6 *(***Generated LTS***) The LTS generated by a TSS* T *consists of the transitions* π *such that* π *can be proven from* T*.*

In order to guarantee congruence for strong bisimulation between the states of the LTS generated by a TSS, the *path format* [4] has been introduced for TSS's. If a TSS is in path format, then bisimulation is a congruence for that TSS, meaning that each function symbol respects this equivalence.

Definition 4.7 *(***Path format***)A transition rule is in* **path format** *if it is of the forms*

$$\frac{\{t_i \xrightarrow{a_i} y_i | i \in I\} \cup \{u_j P_j | j \in J\}}{f(x_1, \ldots, x_{\mathtt{ar}(f)}) \xrightarrow{a} t'}$$

or

$$\frac{\{t_i \xrightarrow{a_i} y_i | i \in I\} \cup \{u_j P_j | j \in J\}}{f(x_1, \ldots, x_{\mathtt{ar}(f)})P}$$

where I, J are sets of indices, P_j are predicates, $a, a_i \in A$, $f \in \mathtt{Sig}$ is a function symbol, the variables $x_1, \ldots, x_{\mathtt{ar}(f)}$ and y_i are all distinct, and, $t', t_i, u_j \in \mathbb{T}(\mathtt{Sig})$ for $i \in I, j \in J$.

A transition system specification is in path format if all its transition rules are in path format.

Theorem 4.8 *(see [4]). If a TSS is in path format, then bisimulation is a congruence with respect to the LTS generated by that TSS.*

4.2.2 Basic thread algebra (BTA)

Basic thread algebra (BTA) was introduced as *basic polarized process algebra* (BPPA) in [22], a theory about sequential programming languages. The semantics of a deterministic sequential program is supposed to be a *polarized process* or a *thread* in BTA. We assume the existence of a set Σ of *basic actions* in BTA. Each basic action performed by a thread is taken as a command to be processed by the execution environment of the thread. At completion of the processing of the command, the execution environment produces a reply value. This reply is either T (`true`) or F (`false`) and is returned to the thread concerned.

Definition 4.9 *The set BTA_Σ of finite threads is defined inductively the following operators:*

- **Successful termination***: $S \in \mathrm{BTA}_\Sigma$ yields successful terminating behavior.*
- **Unsuccessful termination** *or* **deadlock***: $D \in \mathrm{BTA}_\Sigma$ represents inactive behavior.*
- **Postconditional composition***: $(-) \trianglelefteq a \trianglerighteq (-)$ with $a \in \Sigma$. The thread $P \trianglelefteq a \trianglerighteq Q$, where $P, Q \in \mathrm{BTA}_\Sigma$, first performs a and then proceeds with P if T was returned and with Q otherwise. In case $P = Q$ we abbreviate this thread by the* **action prefix** *operator: $a \circ (-)$. In particular, $a \circ P = P \trianglelefteq a \trianglerighteq P$.*

We note that S and D are similar to the termination ϵ and the deadlock δ used in other process algebras such as CCS [75] and ACP [20].

Threads can be *infinite*. An infinite thread in thread algebra is represented by a *projective sequence* consisting of its finite approximations. These finite approximations are defined inductively by means of the approximation operators $\pi_n(-)$ of depth n of threads with $n \in \mathbb{N}$ whose axioms on finite threads are given as P0-P3 in Table 4.1.

Definition 4.10 *A projective sequence is a sequence $(P_n)_{n \in \mathbb{N}}$ such that for all $n \in \mathbb{N}$, $\pi_n(P_{n+1}) = P_n$.*

$\pi_0(x)=D$	P0
$\pi_{n+1}(S)=S$	P1
$\pi_{n+1}(D)=D$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y)=\pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3
If $\pi_n(x) = \pi_n(y)$ for all $n \in \mathbb{N}$ then $x = y$	AIP

Table 4.1: Axioms for approximation operators and induction principle.

$\langle X_i\|E\rangle = t_i(\langle X_1\|E\rangle, \ldots, \langle X_n\|E\rangle)$ $(i \in [1..n])$	RDP
If $y_i = t_i(y_1, \ldots, y_n)$ for $i \in [1..n]$ then $y_i = \langle X_i\|E\rangle$ $(i \in [1..n])$	RSP

Table 4.2: Axioms for the constants $\langle X|E\rangle$.

The axiom AIP (Approximation Induction Principle) in Table 4.1 states that two threads are considered identical if their finite approximations at every depth are identical.

The notion of *regular* threads in thread algebra is given as in other process algebras. They are considered as solutions of *guarded recursive specifications* defined formally as follows.

Definition 4.11 *A* **finite recursive specification** *E is a finite set of recursive equations*

$$\begin{aligned} X_1 &= t_1(X_1, \ldots, X_n) \\ &\vdots \\ X_n &= t_n(X_1, \ldots, X_n) \end{aligned}$$

where X_i are recursive variables, and $t_i(X_1, \ldots, X_n)$ are terms in BTA*. A finite recursive specification E is* **guarded** *if for all i, $t_i(X_1, \ldots, X_n) = S$ or $t_i(X_1, \ldots, X_n) = D$ or $t_i(X_1, \ldots, X_n) = X_{il} \trianglelefteq a \trianglerighteq X_{ir}$ with $a \in \Sigma$, and $1 \leq il, ir \leq n$.*

If E is a guarded recursive specification and X a recursive variable in E, then $\langle X|E\rangle$ denotes the thread that has to be substituted for X in the solution for E.

Theorem 4.12 *A guarded recursive specification E determines a unique solution.*

Proof: See Theorem 5.41. □

The threads determined by guarded recursive specifications are called **regular**. The axioms for regular threads are given in Table 4.2.

$$x \trianglelefteq \mathtt{tau} \trianglerighteq y = \mathtt{tau} \circ x$$

Table 4.3: Axioms for the concrete internal action.

The **concrete internal action** $\mathtt{tau} \in \Sigma$ given in [23] plays a special role. Its execution will never change any state and alway produces a positive reply. The axiom for this action is given in Table 4.3.

4.2.3 Thread algebra (TA)

Thread algebra is an extension of BTA and is designed for strategic interleaving of parallel threads. Thread algebra is a collection of strategic interleaving operators, capturing essential aspects of multi-threading. A sequence of threads to be interleaved is called a *thread vector*. *Strategic interleaving operators* turn a thread vector of arbitrary length into a single thread. This single thread obtained via a strategic interleaving operator is called a *multi-thread*. Formally, both threads and multi-threads are polarized processes.

4.2.4 SOS depending on an execution environment for thread algebra

This section recalls from [23, 25] the SOS for thread algebra. In this SOS, each action of a thread is taken as a command to be processed by the execution environment. This command can be accepted or rejected by the execution environment, depending on the execution history of the thread and external conditions. For example, the execution environment will not accept a command to write a file to a diskette if the diskette is write-protected. Let $\rho : (\Sigma \times \{T, F\})^* \to \mathcal{P}(\Sigma \times \{T, F\})$ be a function representing an execution environment that satisfies the condition: $(a, \kappa) \notin \rho(\alpha) \Rightarrow \rho(\alpha \frown \langle (a, \kappa) \rangle) = \emptyset$ for all $a \in \Sigma$, $\kappa \in \{T, F\}$ and $\alpha \in (\Sigma \times \{T, F\})^*$. Here we write $\langle\rangle$ for the empty sequence, $\langle d \rangle$ for the sequence having d as sole element and $\alpha \frown \beta$ for concatenation of sequences α and β. We assume that $\alpha \frown \langle\rangle = \langle\rangle \frown \alpha = \alpha$. Let $\mathcal{E}$ be the set of execution environments. Given an execution environment $\rho \in \mathcal{E}$ and a basic action $a \in \Sigma$, the derived execution environment of ρ after processing a with a positive reply, written $\frac{\partial^+}{\partial a}\rho$, is defined by $\frac{\partial^+}{\partial a}\rho(\alpha) = \rho(\langle (a, T) \rangle \frown \alpha)$ and the derived execution environment of ρ after processing a with a negative reply, written $\frac{\partial^-}{\partial a}\rho$, is defined by $\frac{\partial^-}{\partial a}\rho(\alpha) = \rho(\langle (a, F) \rangle \frown \alpha)$. The following transition relations on threads are used in the structural operational semantics of [23]:

- the *action step* $\langle p, \rho \rangle \xrightarrow{a} \langle p', \rho' \rangle$ for each $a \in \Sigma$ and $\rho, \rho' \in \mathcal{E}$;
- the *termination* $p \downarrow$; and
- the *deadlock* $q \uparrow$;

$S\downarrow$	$\langle x \trianglelefteq a \trianglerighteq y, \rho\rangle \xrightarrow{a} \langle x, \frac{\partial^+}{\partial a}\rho\rangle$	$((a,T) \in \rho(\langle\rangle))$
$D\uparrow$	$\langle x \trianglelefteq a \trianglerighteq y, \rho\rangle \xrightarrow{a} \langle y, \frac{\partial^-}{\partial a}\rho\rangle$	$((a,F) \in \rho(\langle\rangle))$
	$\langle x \trianglelefteq \mathtt{tau} \trianglerighteq y, \rho\rangle \xrightarrow{\mathtt{tau}} \langle x, \rho\rangle$	

Table 4.4: Transition rules for BTA using execution environments. Here $a \in \Sigma\backslash\{\mathtt{tau}\}$.

The structural operational semantics for BTA is described in Table 4.4. In this SOS, a thread p in the environment ρ can perform an action a depending on a condition $(a,T) \in \rho(\langle\rangle)$ or $(a,F) \in \rho(\langle\rangle)$.

4.3 SOS for thread algebra

In this section, we present another SOS for thread algebra in which the execution of an action in a thread depends only on its execution history. We do not consider threads with blocking actions and thread-service compositions as in [23]. Our SOS is less general than the SOS of [23, 24], but it is simpler. We will show in Section 4.4 that bisimulation between the states of the LTS generated by our TSS characterizes the equality induced by the axioms of thread algebra. We recall from [23] the axioms for the strategic interleaving operators.

4.3.1 Labeled transition systems for thread algebra

We use the following transition relations on threads.

- The *action step* $p \xrightarrow{a,\kappa} p'$ which essentially says that a thread p is capable of first performing a basic action a, and proceeding with thread p', where $\kappa \in \{T,F\}$ denotes the returned boolean value after the execution of a ($\kappa = T$ if `true` is returned after the execution of a and $\kappa = F$ otherwise). This transition can also be written as $p \xrightarrow{a} p'$ if $p \xrightarrow{a,\kappa} p'$ for both $\kappa = T$ and $\kappa = F$, or κ is always T;
- The *concrete internal action step* $p \xrightarrow{\mathtt{tau}} p'$ which essentially says that a thread p is capable of first performing an internal action `tau`, and proceeding with thread p';
- The *termination* $p\downarrow$ means that thread p is capable of termination successfully;
- The *deadlock* $q\uparrow$ means that thread q is neither capable of performing an action nor capable of termination successfully;

Let $A = (\Sigma \setminus \{\mathtt{tau}\}) \times \{T,F\} \cup \{\mathtt{tau}\}$. A labeled transition system for TA is an LTS whose states are threads, whose actions are from the set A, whose transitions are

$$x \trianglelefteq a \trianglerighteq y \xrightarrow{a,T} x \qquad x \trianglelefteq a \trianglerighteq y \xrightarrow{a,F} y \qquad x \trianglelefteq \mathtt{tau} \trianglerighteq y \xrightarrow{\mathtt{tau}} x \qquad S\downarrow \qquad D\uparrow$$

Table 4.5: Transition rules for BTA. Here $a \in \Sigma \setminus \{\mathtt{tau}\}$.

$$\frac{x \xrightarrow{\alpha} x'}{\pi_{n+1}(x) \xrightarrow{\alpha} \pi_n(x')} \qquad \frac{x\downarrow}{\pi_{n+1}(x)\downarrow} \qquad \frac{x\uparrow}{\pi_{n+1}(x)\uparrow} \qquad \frac{}{\pi_0(x)\uparrow}$$

$$\frac{\langle t_X|E\rangle \xrightarrow{\alpha} x'}{\langle X|E\rangle \xrightarrow{\alpha} x'} X = t_X \in E \qquad \frac{\langle t_X|E\rangle\downarrow}{\langle X|E\rangle\downarrow} X = t_X \in E \qquad \frac{\langle t_X|E\rangle\uparrow}{\langle X|E\rangle\uparrow} X = t_X \in E$$

Table 4.6: Transition rules for $\pi_n(x)$ and $\langle X|E\rangle$. Here $\alpha \in A$.

$$\begin{array}{ll}
\|_{csi}(\langle\rangle) = S & \text{CSI1} \\
\|_{csi}(\langle S\rangle \frown \gamma) = \|_{csi}(\gamma) & \text{CSI2} \\
\|_{csi}(\langle D\rangle \frown \gamma) = S_D(\|_{csi}(\gamma)) & \text{CSI3} \\
\|_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma) = \|_{csi}(\gamma \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csi}(\gamma \frown \langle y\rangle) & \text{CSI4} \\
 & \\
S_D(S) = D & \text{S2D1} \\
S_D(D) = D & \text{S2D2} \\
S_D(x \trianglelefteq a \trianglerighteq y) = S_D(x) \trianglelefteq a \trianglerighteq S_D(y) & \text{S2D3}
\end{array}$$

Table 4.7: Axioms for cyclic rotation. Here $a \in \Sigma$.

described as above, and whose predicates are $\uparrow$ and $\downarrow$. For a thread p, we write $p\updownarrow$ if $p\uparrow$ or $p\downarrow$.

4.3.2 Transition rules for BTA

The transition rules for BTA are given in Table 4.5. Transition rules for the approximation operator, and for the regular threads $\langle X|E\rangle$ are given in Table 4.6.

4.3.3 Transition rules for cyclic rotation

This section presents transition rules for the cyclic rotation or the cyclic interleaving operator, a basic interleaving strategy of [23]. The axioms for this strategy are given in Table 4.7.

The cyclic interleaving operator $\|_{csi}$: $\mathrm{BTA}_\Sigma{}^* \to \mathrm{BTA}_\Sigma$ works in a round-robin fashion which invokes rotation of the thread vector after every step. Let $\langle\rangle$ denote the empty sequence, $\langle x\rangle$ stand for a sequence of length one, and $\gamma \frown \beta$ the concatenation of two sequences. We assume that the following identity holds: $\gamma \frown \langle\rangle = \langle\rangle \frown \gamma = \gamma$.

$$\frac{x_1\downarrow,\ldots,x_n\downarrow,x\xrightarrow{\alpha}x'}{\|_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\rangle\frown\gamma)\xrightarrow{\alpha}\|_{csi}\ (\gamma\frown\langle x'\rangle)}\ n\geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow,x\xrightarrow{\alpha}x'}{\|_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\rangle\frown\gamma)\xrightarrow{\alpha}S_D(\|_{csi}\ (\gamma\frown\langle x'\rangle))}\ 0<m\leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle)\downarrow}\ n\geq 0 \qquad \frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{\|_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle)\uparrow}0<m\leq n$$

$$\frac{x\xrightarrow{\alpha}x'}{S_D(x)\xrightarrow{\alpha}S_D(x')} \qquad \frac{x\downarrow}{S_D(x)\uparrow} \qquad \frac{x\uparrow}{S_D(x)\uparrow} \qquad \frac{x\uparrow}{x\updownarrow} \qquad \frac{x\downarrow}{x\updownarrow}$$

Table 4.8: Transition rules for the cyclic interleaving operator $\|_{csi}\ (-)$ and $S_D(-)$. Here $\alpha\in A$.

The axioms for the cyclic interleaving operator are the axioms CSI1-CSI4 in Table 4.7, where a ranges over Σ. In axiom CSI3, an auxiliary operator S_D is used. This operator turns termination of a thread to deadlock which means that if one thread in a thread vector deadlocks, the whole does not deadlock till all other have terminated or deadlocked. The axioms for S_D are the axioms S2D1-S2D3 in Table 4.7.

Transition rules for cyclic rotation $\|_{csi}\ (-)$ and $S_D(-)$ are given in Table 4.8.

4.3.4 Transition rules for step counting

A simple variation of the cyclic interleaving operator $\|_{csi}$ is $\|^{k,l}_{csi}$: $\mathrm{BTA}_\Sigma{}^*\to\mathrm{BTA}_\Sigma$ which is equipped with counters and gives each thread a fixed number k of consecutive steps. The superscript l indicates that $l-1$ of the k steps have already been performed. The axioms for the strategic interleaving operator $\|^{k,l}_{csi}$ are given in Table 4.9 (CSIsc0-CSIsc5). Here a ranges over Σ. Clearly, for all γ, $\|_{csi}\ (\gamma)=\|^{1,1}_{csi}\ (\gamma)$. We note that the conditional operator $-\lhd-\rhd-$ is defined as follows: $x\lhd\mathtt{true}\rhd y=x$ and $x\lhd\mathtt{false}\rhd y=y$.

The action YIELD

A thread in thread algebra can have an action $\mathtt{YIELD}\in\Sigma$ which means handing over control to another thread. This action becomes meaningful in the step counting strategy when $k>1$. The axiom CSIsc6 in Table 4.9 is defined for this action.

Transition rules for the step counting strategy $\|^{k,l}_{csi}$ are given as in Table 4.8 (for $k=l=1$) and in Table 4.10 (for $k>1$). Transition rules for the strategy $\|^{k}_{csi}$ are similar to the transition rules for the strategy $\|^{k,1}_{csi}$. Note that in the transition rules for the action YIELD, the counter k must be greater than 1.

$\|^k_{csi}(\gamma) = \|^{k,1}_{csi}(\gamma)$	CSIsc0
$\|^{k,l}_{csi}(\langle\rangle) = S$	CSIsc1
$\|^{k,l}_{csi}(\langle S\rangle \frown \gamma) = \|^{k,1}_{csi}(\gamma)$	CSIsc2
$\|^{k,l}_{csi}(\langle D\rangle \frown \gamma) = S_D(\|^{k,1}_{csi}(\gamma))$	CSIsc3
$\|^{k,l}_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma) =$ $\|^{k,1}_{csi}(\gamma \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{k,1}_{csi}(\gamma \frown \langle y\rangle)$ $\lhd k = l \rhd$ $\|^{k,l+1}_{csi}(\langle x\rangle \frown \gamma) \trianglelefteq a \trianglerighteq \|^{k,l+1}_{csi}(\langle y\rangle \frown \gamma)$	CSIsc4
$\|^{k,l}_{csi}(\langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle \frown \gamma) =$ $\mathtt{tau} \circ (\|^{k,1}_{csi}(\gamma \frown \langle x\rangle) \lhd \gamma \neq \langle\rangle \rhd \|^{k,l+1}_{csi}(\langle y\rangle))$	CSIscY

Table 4.9: Axioms for the strategic interleaving operator $\|^{k,l}_{csi}(-)$. Here $a \in \Sigma$.

4.3.5 Transition rules for the current thread persistence operator

The cyclic interleaving operator switches the execution of the current thread to another thread automatically after every single step. In contrast, the *current thread persistence* operator $\|_{ctp}\colon \mathrm{BTA}_\Sigma{}^* \to \mathrm{BTA}_\Sigma$ invokes rotation of the thread vector only when asked for by the thread via an action `YIELD`. Table 4.11 provides axioms for the current thread persistent strategy, and Table 4.12 presents its transition rules. Here a ranges over Σ and α ranges over A.

4.3.6 Transition rules for the strategic interleaving operator $\|^{W2}_{csi}$

In thread algebra, basic actions from different threads can be performed simultaneously. The number of basic actions that can be performed simultaneously is called the *basic action width*.

Table 4.13 provides axioms for the strategic interleaving operator $\|^{W2}_{csi}\colon \mathrm{BTA}_\Sigma{}^* \to \mathrm{BTA}_\Sigma$ in which the basic action width is two. Actions a and b are *independent*, written as $a\#b$, if both can be performed simultaneously with an effect that equals the effect of performing them in any of the two possible orderings. We assume that independence is known as a relation given on actions, and basic actions independent of other basic actions always returns `true`. The result of performing independent actions a and b simultaneously is considered to be a basic action, denoted by $a|b$.

The axiomatization given in Table 4.13 provides transition rules for the strategic interleaving operator $\|^{W2}_{csi}$ as in Table 4.14.

4.3.7 Transition rules for thread creation

This section provides transition rules for thread creation (or forking off) [23], an important feature in multi-threading. The axioms for thread creation are given in

$$\frac{x_1\downarrow,\ldots,x_n\downarrow,x\xrightarrow{\alpha}x'}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\rangle\frown\gamma)\xrightarrow{\alpha}\|^{k,2}_{csi}\ (\langle x'\rangle\frown\gamma)}n>0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow,x\xrightarrow{\alpha}x'}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\rangle\frown\gamma)\xrightarrow{\alpha}S_D(\|^{k,2}_{csi}\ (\langle x'\rangle\frown\gamma))}0<n,0<m\leq n$$

$$\frac{x\xrightarrow{\alpha}x'}{\|^{k,l}_{csi}\ (\langle x\rangle\frown\gamma)\xrightarrow{\alpha}\|^{k,l+1}_{csi}\ (\langle x'\rangle\frown\gamma)}l<k$$

$$\frac{x\xrightarrow{\alpha}x'}{\|^{k,l}_{csi}\ (\langle x\rangle\frown\gamma)\xrightarrow{\alpha}\|^{k,1}_{csi}\ (\gamma\frown\langle x'\rangle)}\ l=k$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle)\downarrow}$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle)\uparrow}\ 0<m\leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle\frown\gamma)\xrightarrow{\texttt{tau}}\|^{k,1}_{csi}\ (\gamma\frown\langle x\rangle)}\gamma\neq\langle\rangle$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle\frown\gamma)\xrightarrow{\texttt{tau}}S_D(\|^{k,1}_{csi}\ (\gamma\frown\langle x\rangle))}0<m\leq n,\gamma\neq\langle\rangle$$

$$\frac{}{\|^{k,l}_{csi}\ (\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle)\xrightarrow{\texttt{tau}}\|^{k,l+1}_{csi}\ (\langle y\rangle)}\ l<k$$

$$\frac{}{\|^{k,l}_{csi}\ (\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle)\xrightarrow{\texttt{tau}}\|^{k,1}_{csi}\ (\langle y\rangle)}\ l=k$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle)\xrightarrow{\texttt{tau}}\|^{k,2}_{csi}\ (\langle y\rangle)}n\geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{\|^{k,l}_{csi}\ (\langle x_1\rangle\frown\cdots\frown\langle x_n\rangle\frown\langle x\trianglelefteq\texttt{YIELD}\trianglerighteq y\rangle)\xrightarrow{\texttt{tau}}S_D(\|^{k,2}_{csi}\ (\langle y\rangle))}\ 0<m\leq n$$

Table 4.10: Transition rules for the step counting strategy with $k>1$. Here $\alpha\in A$.

Table 4.15. The axioms CSIf5, CSIbf5 and CSIbff5 are alternative choices of the cyclic interleaving strategy when dealing with thread forking. More precisely, the axiom CSIf5 defines *thread creation* in general, while the axiom CSIbf5 considers the case that the current thread forking is temporarily blocked. Finally, the axiom CSIbff5 deals with thread creation by separating blocked thread forking from failed thread forking. These notions are described below:

$\|_{ctp}(\langle\rangle) = S$	ctpSI1
$\|_{ctp}(\langle S\rangle \frown \gamma) = \|_{ctp}(\gamma)$	ctpSI2
$\|_{ctp}(\langle D\rangle \frown \gamma) = S_D(\|_{ctp}(\gamma))$	ctpSI3
$\|_{ctp}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma) = \|_{ctp}(\langle x\rangle \frown \gamma) \trianglelefteq a \trianglerighteq \|_{ctp}(\langle y\rangle \frown \gamma)$	ctpSI4
$\|_{ctp}(\langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle \frown \gamma) = \mathtt{tau} \circ (\|_{ctp}(\gamma \frown \langle x\rangle) \lhd \gamma \neq \langle\rangle \rhd \|_{ctp}(\langle y\rangle))$	ctpSI5

Table 4.11: Axioms for the current thread persistent strategy $\|_{ctp}(-)$. Here $a \in \Sigma$.

$$\frac{x_1\downarrow,\ldots,x_n\downarrow, x \xrightarrow{\alpha} x'}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \gamma) \xrightarrow{\alpha} \|_{ctp}(\langle x'\rangle \frown \gamma)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow, x_m\uparrow, x \xrightarrow{\alpha} x'}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \gamma) \xrightarrow{\alpha} S_D(\|_{ctp}(\langle x'\rangle \frown \gamma))}\ 0 < m \leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle)\downarrow}\ n \geq 0 \qquad \frac{x_1\updownarrow,\ldots,x_n\updownarrow, x_m\uparrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle)\uparrow} 0 < m \leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{tau}} \|_{ctp}(\gamma \frown \langle x\rangle)}\ n \geq 0, \gamma \neq \langle\rangle$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow, x_m\uparrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{tau}} S_D(\|_{ctp}(\gamma \frown \langle x\rangle))}\ 0 < m \leq n, \gamma \neq \langle\rangle$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle) \xrightarrow{\mathtt{tau}} \|_{ctp}(\langle y\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow, x_m\uparrow}{\|_{ctp}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{YIELD} \trianglerighteq y\rangle) \xrightarrow{\mathtt{tau}} S_D(\|_{ctp}(\langle y\rangle))}\ 0 < m \leq n$$

Table 4.12: Transition rules for the current thread presistent strategy. Here $\alpha \in A$.

Thread creation or forking (CSIf)

A new additional thread is created by a forking action, and will be running in the same context. This forking action may succeed, giving rise to a new thread indeed or fail in which case no new thread is created. To deal with forking off, an operator *new thread* $\mathtt{NT}(x)$ is used to present the act of trying to fork off a new thread x. Therefore, $\mathtt{NT}(x)$ is viewed as a basic action ignoring the way the new thread may be dealt with by a strategic interleaving operator. An additional axiom for $\pi_n(-)$ with fork action is given in Table 4.16. Furthermore, an additional basic action $\mathtt{NT} \in \Sigma$ is required whose processing succeeds if the creation of a new thread takes place, and fails otherwise. Transition rules for the strategy $\|_{csi,f}\colon \mathrm{BTA}_\Sigma{}^* \to \mathrm{BTA}_\Sigma$ in the case that the current action is not a forking off action, are similar to the transition rules for the cyclic

$$\|^{W2}_{csi}(\langle\rangle) = S \qquad \text{CSIW1}$$
$$\|^{W2}_{csi}(\langle S\rangle \frown \gamma) = \|^{W2}_{csi}(\gamma) \qquad \text{CSIW2}$$
$$\|^{W2}_{csi}(\langle D\rangle \frown \gamma) = S_D(\|^{W2}_{csi}(\gamma)) \qquad \text{CSIW3}$$
$$\|^{W2}_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle) = \|^{W2}_{csi}(\langle x\rangle) \trianglelefteq a \trianglerighteq \|^{W2}_{csi}(\langle y\rangle) \qquad \text{CSIW4}$$
$$\|^{W2}_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \langle u \trianglelefteq b \trianglerighteq v\rangle \frown \gamma) = \quad a|b \circ \|^{W2}_{csi}(\gamma \frown \langle x\rangle \frown \langle u\rangle)$$
$$\triangleleft a\#b \triangleright$$
$$\|^{W2}_{csi}(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \gamma \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{W2}_{csi}(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \gamma \frown \langle y\rangle) \qquad \text{CSIW5}$$

Table 4.13: Axioms for the strategic interleaving operator $\|^{W2}_{csi}$. Here $a, b \in \Sigma$.

$$\frac{x \xrightarrow{\alpha} x'}{\|^{W2}_{csi}(x) \xrightarrow{\alpha} \|^{W2}_{csi}(x')}$$

$$\frac{x_1\downarrow, \ldots, x_n\downarrow, x \xrightarrow{a,T} x', y \xrightarrow{b,T} y'}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \langle y\rangle \frown \gamma) \xrightarrow{a|b} \|^{W2}_{csi}(\gamma \frown \langle x'\rangle \frown \langle y'\rangle)} \; n \geq 0, a\#b$$

$$\frac{x_1\downarrow, \ldots, x_n\downarrow, x \xrightarrow{a,\kappa} x', y \xrightarrow{b,\varrho} y'}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \langle y\rangle \frown \gamma) \xrightarrow{a,\kappa} \|^{W2}_{csi}(\langle y\rangle \frown \gamma \frown \langle x'\rangle)} \; n \geq 0, \neg(a\#b)$$

$$\frac{x_1\updownarrow, \ldots, x_n\updownarrow, x_m\uparrow, x \xrightarrow{a,T} x', y \xrightarrow{b,T} y'}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \langle y\rangle \frown \gamma) \xrightarrow{a|b} S_D(\|^{W2}_{csi}(\gamma \frown \langle x'\rangle \frown \langle y'\rangle))} \; 0 < m \leq n, a\#b$$

$$\frac{x_1\updownarrow, \ldots, x_n\updownarrow, x_m\uparrow, x \xrightarrow{a,\kappa} x', y \xrightarrow{b,\varrho} y'}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x\rangle \frown \langle y\rangle \frown \gamma) \xrightarrow{a,\kappa} S_D(\|^{W2}_{csi}(\langle y\rangle \frown \gamma \frown \langle x'\rangle))} \; 0 < m \leq n, \neg(a\#b)$$

$$\frac{x_1\downarrow, \ldots, x_n\downarrow}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle)\downarrow} \; n \geq 0 \qquad \frac{x_1\updownarrow, \ldots, x_n\updownarrow, x_m\uparrow}{\|^{W2}_{csi}(\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle)\uparrow} \; 0 < m \leq n$$

Table 4.14: Transition rules for the strategic interleaving operator $\|^{W2}_{csi}$. Here $a, b \in \Sigma$ and $\alpha \in A$.

interleaving strategy $\|_{csi}$ in Table 4.8 with $\|_{csi}$ replaced by $\|_{csi,f}$. Transition rules for forking off are given Table 4.17.

Blocked thread forking (CSIbf)

It can happen that thread forking is temporarily blocked (or disabled). Blocked thread forking is capable of postponing when thread forking is disabled. This strategy requires an additional test action `?NT` $\in \Sigma$. Its processing succeeds if thread forking

$$\|_{csi,f}\ (\langle\rangle) = S \qquad \text{CSIf1}$$
$$\|_{csi,f}\ (\langle S\rangle \frown \gamma) = \|_{csi,f}\ (\gamma) \qquad \text{CSIf2}$$
$$\|_{csi,f}\ (\langle D\rangle \frown \gamma) = S_D(\|_{csi,f}\ (\gamma)) \qquad \text{CSIf3}$$
$$\|_{csi,f}\ (\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma) = \|_{csi,f}\ (\gamma \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csi,f}\ (\gamma \frown \langle y\rangle) \qquad \text{CSIf4}$$
$$\|_{csi,f}\ (\langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) = \|_{csi,f}\ (\gamma \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathtt{NT} \trianglerighteq \|_{csi,f}\ (\gamma \frown \langle y\rangle) \qquad \text{CSIf5}$$
$$\|_{csi,f}\ (\langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) = \|_{csi,f}\ (\gamma \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathtt{?NT} \trianglerighteq \|_{csi,f}\ (\gamma \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle) \qquad \text{CSIbf5}$$
$$\|_{csi,bff}\ (\langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) = \|_{csi,bff}\ (\gamma \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathtt{NT} \trianglerighteq \|_{csi,f}\ (\gamma \frown \langle y\rangle) \trianglelefteq \mathtt{?NT} \trianglerighteq \|_{csi,bff}\ (\gamma \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle) \qquad \text{CSIbff5}$$

Table 4.15: Axioms for thread creation. Here $a \in \Sigma$.

$$\pi_{n+1}(x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y) = \pi_n(x) \trianglelefteq \mathtt{NT}(\pi_n(z)) \trianglerighteq \pi_n(x)$$

Table 4.16: An additional axiom for approximation operators with fork actions.

$$\frac{x_1\downarrow, \ldots, x_n\downarrow}{\|_{csi,f}\ (\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{NT},T} \|_{csi,f}\ (\gamma \frown \langle z\rangle \frown \langle x\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow, \ldots, x_n\updownarrow, x_m\uparrow}{\|_{csi,f}\ (\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{NT},T} S_D(\|_{csi,f}\ (\gamma \frown \langle z\rangle \frown \langle x\rangle))}\ 0 < m \leq n$$

$$\frac{x_1\downarrow, \ldots, x_n\downarrow}{\|_{csi,f}\ (\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{NT},F} \|_{csi,f}\ (\gamma \frown \langle y\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow, \ldots, x_n\updownarrow, x_m\uparrow}{\|_{csi,f}\ (\langle x_1\rangle \frown \cdots \frown \langle x_n\rangle \frown \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \frown \gamma) \xrightarrow{\mathtt{NT},F} S_D(\|_{csi,f}\ (\gamma \frown \langle y\rangle))}\ 0 < m \leq n$$

Table 4.17: Transition rules for $\|_{csi,f}\ (-)$ dealing with thread forking.

is enabled and fails if thread forking is disabled. Transition rules for thread creation with blocked thread forking are given in Table 4.18.

Blocked and failed fork actions

The previous strategy is only adequate if enabled thread forking always succeeds. A better strategy for thread creation is given by axiom CSIbff that separates blocked thread forking from failed thread forking. In this strategy, thread forking may still fail

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{P \overset{?\mathtt{NT},T}{\rightarrow} \|_{csi,f}\ (\gamma \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{P \overset{?\mathtt{NT},T}{\rightarrow} S_D(\|_{csi,f}\ (\gamma \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle))}\ 0 < m \leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{P \overset{?\mathtt{NT},F}{\rightarrow} \|_{csi,f}\ (\gamma \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_k\updownarrow,x_l\uparrow}{P \overset{?\mathtt{NT},F}{\rightarrow} S_D(\|_{csi,f}\ (\gamma \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle))}\ 0 < m \leq n$$

Table 4.18: Transition rules for $\|_{csi,f}\ (-)$ dealing with blocked thread forking. Here $P = \|_{csi,f}\ (\langle x_1\rangle \curvearrowright \cdots \curvearrowright \langle x_n\rangle \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \curvearrowright \gamma)$.

if it is not blocked. Transition rules for blocked and failed fork actions are presented in Table 4.19.

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{P \overset{?\mathtt{NT},T}{\rightarrow} \|_{csi,bff}\ (\gamma \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle) \trianglelefteq \mathtt{NT} \trianglerighteq \|_{csi,bff}\ (\gamma \curvearrowright \langle y\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{P \overset{?\mathtt{NT},T}{\rightarrow} S_D(\|_{csi,bff}\ (\gamma \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle) \trianglelefteq \mathtt{NT} \trianglerighteq \|_{csi,bff}\ (\gamma \curvearrowright \langle y\rangle))}\ 0 < m \leq n$$

$$\frac{x_1\downarrow,\ldots,x_n\downarrow}{P \overset{?\mathtt{NT},F}{\rightarrow} \|_{csi,bff}\ (\gamma \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle)}\ n \geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_n\updownarrow,x_m\uparrow}{P \overset{?\mathtt{NT},F}{\rightarrow} S_D(\|_{csi,bff}\ (\gamma \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle))}\ 0 < m \leq n$$

Table 4.19: Transition rules for $\|_{csi,bff}\ (-)$ with blocked and failed fork actions. Here $P = \|_{csi,bff}\ (\langle x_1\rangle \curvearrowright \cdots \curvearrowright \langle x_n\rangle \curvearrowright \langle x \trianglelefteq \mathtt{NT}(z) \trianglerighteq y\rangle \curvearrowright \gamma)$.

4.3.8 Transition rules for terminating a named thread

This section presents transition rules for the strategic interleaving operator $\|^{\beta}_{csi,fn}$: $\mathrm{BTA}_\Sigma{}^* \rightarrow \mathrm{BTA}_\Sigma$ of [23] that deals with terminating a named thread. Each thread now is named by a positive number which should occur as the first parameter of the thread forking action $\mathtt{NT}(k,x)$, and forking is possible unless a thread with the intended name already exists. Thread names initially are 0. The superscript β of the strategic operator $\|^{\beta}_{csi,fn}$ is a vector of thread names, one for each thread in the thread vector in the corresponding ordering. Two auxiliary operators on name vectors and

$$\|_{csi,fn}(\gamma) = \|^{\vec{0}}_{csi,fn}(\gamma) \quad \text{CSIfn0}$$

$$\|^{\beta}_{csi,fn}(\langle\rangle) = S \quad \text{CSIfn1}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle S\rangle \frown \gamma) = \|^{\beta}_{csi,fn}(\gamma) \quad \text{CSIfn2}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle D\rangle \frown \gamma) = S_D(\|^{\beta}_{csi,fn}(\gamma)) \quad \text{CSIfn3}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma) = \|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle y\rangle) \quad \text{CSIfn4}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle x \trianglelefteq \mathtt{NT}(k,z) \trianglerighteq y\rangle \frown \gamma) = \mathtt{tau} \circ (\|^{\beta\frown\langle k\rangle\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle z\rangle \frown \langle x\rangle) \lhd k \notin \beta \wedge k \neq n \rhd \|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle y\rangle)) \quad \text{CSIfn5}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle x \trianglelefteq \mathtt{terminate!}k \trianglerighteq y\rangle \frown \gamma) = \mathtt{tau} \circ (\|^{\beta}_{csi,fn}(\gamma) \lhd k = n \rhd (\|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle y\rangle) \lhd k \notin \beta \rhd \|^{\beta-k\frown\langle n\rangle}_{csi,fn}(\rho_{\beta-k}(\gamma) \frown \langle x\rangle))) \quad \text{CSIfn6}$$

$$\|^{\langle n\rangle\frown\beta}_{csi,fn}(\langle x \trianglelefteq \mathtt{isalive?}k \trianglerighteq y\rangle \frown \gamma) = \mathtt{tau} \circ (\|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle x\rangle) \lhd k = s \vee k \in \beta \rhd \|^{\beta\frown\langle n\rangle}_{csi,fn}(\gamma \frown \langle y\rangle)) \quad \text{CSIfn7}$$

Table 4.20: Axioms for $\|^{\beta}_{csi,fn}(-)$. Here $a \in \Sigma$.

thread vectors are needed: $\beta - k$ is the sequence obtained from β by removing k if it occurs in it and β itself otherwise; and $\rho_{\beta-k}(\gamma)$ removes from thread vector γ the thread(s) named k if k occurs in β and leaves the thread vector unchanged otherwise. These operators give the ability to terminate a named thread from within another (or the same) by an action of the form $\mathtt{terminate!}k$, and the option to test if a named thread is still alive by an action of the form $\mathtt{isalive?}k$. Note that the axiomatization of these operators is omitted. The axioms and transition rules for $\|^{\beta}_{csi,fn}$ are given in Table 4.20. and Table 4.21, respectively. Here a ranges over Σ and α ranges over A. Transition rules for the strategy $\|_{csi,fn}$ are similar to the transition rules for $\|^{\vec{0}}_{csi,fn}$.

One can see that the strategic interleaving operators presented in this section can be defined on regular threads as well, since regularity is closed with respect to these operators.

4.4 Bisimulation equivalence characterizes axiomatization

In this section, we show that bisimulation is a congruence with respect to our transition rules for thread algebra. This implies that bisimulation equivalence between regular threads characterizes the equality induced by the axioms of thread algebra. Let $\mathcal{E}_{\mathrm{TA}}$ be the set of axioms given in Table 4.1, Table 4.2, Table 4.3, Table 4.7, Table 4.9, Table 4.11, Table 4.13, Table 4.15, Table 4.16 and Table 4.20. Let $\mathcal{T}_{\mathrm{TA}}$ be the set of transition rules given in Table 4.5, Table 4.6, Table 4.8, Table 4.10, Table 4.12,

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{\|^{\theta}_{csi,fn}\,(\langle x_1\rangle\frown\cdots\frown\langle x_N\rangle)\downarrow}\ N\geq 0,|\theta|=N$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{\|^{\theta}_{csi,fn}\,(\langle x_1\rangle\frown\cdots\frown\langle x_N\rangle)\uparrow}\ 0<m\leq N,|\theta|=N$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow,x\xrightarrow{\alpha}x'}{P(x)\xrightarrow{\alpha}\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle x'\rangle)}\ N\geq 0$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow,x\xrightarrow{\alpha}x'}{P(x)\xrightarrow{\alpha}S_D(\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle x'\rangle))}\ 0<m\leq N$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{NT}(k,z)\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta\frown\langle k\rangle\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle z\rangle\frown\langle x\rangle)}\ N\geq 0,k\notin\beta$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{NT}(k,z)\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta\frown\langle k\rangle\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle z\rangle\frown\langle x\rangle))}\ 0<m\leq N,k\notin\beta$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{NT}(k,z)\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle)}\ N\geq 0,k\in\beta$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{NT}(k,z)\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta\frown\langle k\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle))}\ 0<m\leq N,k\in\beta$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta}_{csi,fn}\,(\gamma)}\ N\geq 0,k=n$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|_{csi}\,(\gamma))}\ 0<m\leq N,k=n$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle)}\ N\geq 0,k\neq n,k\notin\beta$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle))}\ 0<m\leq N,k\neq n,k\notin\beta$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta-k\frown\langle n\rangle}_{csi,fn}\,(\rho_{\beta-k}(\gamma)\frown\langle y\rangle)}\ N\geq 0,k\neq n,k\in\beta$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{terminate!}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta-k\frown\langle n\rangle}_{csi,fn}\,(\rho_{\beta-k}(\gamma)\frown\langle y\rangle))}\ 0<m\leq N,k\neq n,k\in\beta$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{isalive?}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle x\rangle)}\ N\geq 0,k=n\vee k\in\beta$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{isalive?}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle x\rangle))}\ 0<m\leq N,k=n\vee k\in\beta$$

$$\frac{x_1\downarrow,\ldots,x_N\downarrow}{P(x\trianglelefteq\mathtt{isalive?}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle)}\ N\geq 0,\neg(k=n\vee k\in\beta)$$

$$\frac{x_1\updownarrow,\ldots,x_N\updownarrow,x_m\uparrow}{P(x\trianglelefteq\mathtt{isalive?}k\trianglerighteq y)\xrightarrow{\mathtt{tau}}S_D(\|^{\beta\frown\langle n\rangle}_{csi,fn}\,(\gamma\frown\langle y\rangle))}\ 0<m\leq N,\neg(k=n\vee k\in\beta)$$

Table 4.21: Transition rules for the strategic interleaving operator $\|^{\beta}_{csi,fn}\,(-)$. Here $P(t)=\|^{\theta\frown\langle n\rangle\frown\beta}_{csi,fn}\,(\langle x_1\rangle\frown\cdots\frown\langle x_N\rangle\frown\langle t\rangle\frown\gamma)$ with $|\theta|=N$ and $\alpha\in A$.

Table 4.14, Table 4.17, Table 4.18, Table 4.19 and Table 4.21.

Theorem 4.13 *Bisimulation is a congruence with respect to* $\mathcal{T}_{\mathrm{TA}}$.

Proof: This is the case because the TSS $\mathcal{T}_{\mathrm{TA}}$ is in path format. □

Theorem 4.14 *The induced equality relation* $=$ *by the axioms in* $\mathcal{E}_{\mathrm{TA}}$ *characterizes bisimulation equivalence over* $\mathcal{T}_{\mathrm{TA}}$*, i.e, for regular threads* p *and* q, $p = q \Leftrightarrow p \leftrightarrows q$.

Proof:

1. Soundness ($\Rightarrow$): Since bisimulation is both equivalence and congruence for $\mathcal{T}_{\mathrm{TA}}$, we only need to check that if $p = q$ is obtained by an axiom $s = t$ in $\mathcal{E}_{\mathrm{TA}}$ and a substitution σ such that $\sigma(s) = p$ and $\sigma(t) = q$ then $p \leftrightarrows q$. We consider the case that $p = q$ derived from the induction principle $\pi_n(p) = \pi_n(q)$ for all $n \in \mathbb{N}$. It is not hard to see that $\pi_n(p) \leftrightarrows \pi_n(q)$. We define a binary relation $\mathcal{B}$ between threads p' and q' as follows: $(p', q') \in \mathcal{B}$ if $\pi_n(p') \leftrightarrows \pi_n(q')$ for all $n \in \mathbb{N}$. We show that $\mathcal{B}$ is a bisimulation. If $p' \in \{S, D\}$ then this is trivial. If $p' \xrightarrow{\alpha} p''$ then $\pi_{n+1}(p') \xrightarrow{\alpha} \pi_n(p'')$ for all $n \in \mathbb{N}$. Since $\pi_{n+1}(p') \leftrightarrows \pi_{n+1}(q')$, $q' \xrightarrow{\alpha} q''$ and $\pi_n(p'') \leftrightarrows \pi_n(q'')$. This implies that $(p'', q'') \in \mathcal{B}$. Therefore, $\mathcal{B}$ is a bisimulation. Hence, $p \leftrightarrows q$. The other cases are obvious.

2. Ground-completeness ($\Leftarrow$): Given regular threads p and q, these can be equated to $\langle X_1|E_1\rangle$ and $\langle Y_1|E_2\rangle$, respectively. By soundness, $p \leftrightarrows \langle X_1|E_1\rangle$ and $q \leftrightarrows \langle Y_1|E_2\rangle$. Since $p \leftrightarrows q$, it remains to be proved that if $\langle X_1|E_1\rangle \leftrightarrows \langle Y_1|E_2\rangle$ then $\langle X_1|E_1\rangle = \langle Y_1|E_2\rangle$. Let E be a recursive specification defined as follows. For variables X and Y of E_1 and E_2 such that $\langle X|E_1\rangle \leftrightarrows \langle Y|E_2\rangle$, we define a recursive variable Z_{XY} of E as $Z_{XY} = S$ if $X = S$, $Z_{XY} = D$ if $X = D$, and $Z_{XY} = Z_{X'Y'} \trianglelefteq a \trianglerighteq Z_{X''Y''}$ if $X = X' \trianglelefteq a \trianglerighteq X''$ and $Y = Y' \trianglelefteq a \trianglerighteq Y''$. It can be shown that $\langle X|E_1\rangle$ is a solution for the recursive variable Z_{XY} of E. Since a recursive specification has a unique solution, $\langle Z_{XY}|E\rangle = \langle X|E_1\rangle$. Similarly, $\langle Z_{XY}|E\rangle = \langle Y|E_2\rangle$. Thus, $\langle X|E_1\rangle = \langle Y|E_2\rangle$. Therefore, $\langle X_1|E_1\rangle = \langle Y_1|E_2\rangle$.

□

4.5 Concluding remarks

In this chapter, we have presented a structural operational semantics for thread algebra. This SOS is less general, but simpler than that of [23]. We have shown that the axioms of thread algebra [23] are sound and ground-complete with respect to bisimulation equivalence induced by this SOS, meaning that two processes are equal if and only if they are bisimilar.

Chapter 5

Denotational semantics for thread algebra

5.1 Introduction

In 2002, Bergstra and Loots proposed a semantics for sequential programming languages called *basic polarized process algebra* (BPPA) [22]. Later, Bergstra and Middelburg renamed BPPA to *basic thread algebra* (BTA) and extended BTA to *thread algebra* (TA) with a collection of *strategic interleaving operators* [23]. It has been outlined in [22, 23, 24, 25] that TA is a dominant form of concurrency provided by recent object-oriented programming languages such as C# and Java, where arbitrary interleaving is not an appropriate intuition when dealing with multi-threading.

In [102], a structural operational semantics of TA is given. This chapter is an extension of [104] which focuses on denotational semantics for TA. The difference between these two semantics is that the former generates expressions in a programming language in a stepwise fashion, while the latter constructs them as elements of some suitable domain equation [94]. We employ the metric methodology of de Bakker and Zucker [11] to give a denotational semantics for TA. This method turns the domain of single threads into a complete metric space, in which the distance between two threads that do not differ in behavior until the n-th step is at most 2^{-n}. We show that the metric space consisting of *projective* sequences of threads is an appropriate domain for TA by comparing it to other domains of TA. In particular, the infinite threads of this domain are represented in a unique way. Furthermore, it deals naturally with abstraction [73, 20]. Moreover, it is compatible with the domain based on complete partial orders (cpo's) of [16]. Our domain can be extended with strategic interleaving operators of [23] in a natural way while the domain based on cpo's cannot. Finally, by means of Banach's fixed point theorem, one can show that the specification of a regular thread has a unique solution.

In this chapter, we also propose a particular interleaving strategy for thread algebra, called the *cyclic internal persistence* operator, with respect to abstraction of

internal actions. In TA, *concrete internal actions* [23] may arise due to the interactions between *clients* and *servers* [17], threads and *services*, and threads and the execution environment. It is stated in [23] that the presence of concrete internal actions matters, and there is no abstraction made of it via equations that remove these actions. However, abstraction is still necessary in certain cases. For instance, in [17], abstraction is defined to *emulate* the interaction between clients and servers, assuming that clients and servers are threads in BTA. It would be natural if abstraction is *compositional* with respect to interleaving strategies of parallel threads. Unfortunately, this property does not hold for the existing interleaving operators of the thread algebra given in [23]. The cyclic internal persistence strategy is a variant of the *cyclic interleaving operator* of [23] which will not invoke the rotation of a thread sequence if the current action is internal. We will show that with the use of this strategy, abstraction can be made compositional, provided that threads cannot perform an infinite sequence of internal actions.

The structure of this chapter is as follows. Section 5.2 recalls the basic concepts of complete metric spaces, complete partial orders, BTA and TA. Section 5.3 turns the domains of BTA into complete metric spaces, and shows that the complete metric space consisting of projective sequences is an appropriate domain for BTA. Section 5.4 extends the domain of BTA with the strategic interleaving operators of thread algebra. Section 5.5 defines the cyclic internal persistence operator dealing with abstraction for thread algebra. The chapter is ended with some concluding remarks in Section 5.6.

5.2 Preliminaries

In this section, we provide some basic concepts that will be needed for the rest of the chapter.

5.2.1 Metric spaces and complete partial orders

Complete metric spaces and complete partial orders have major applications in denotational semantics. In this chapter, we will use a few basic concepts of the metric topology and the domain theory taken from [46, 94, 9] to give a denotational semantics for thread algebra.

Metric spaces

A metric space is a set where a notion of *distance* (or *metric*) between elements of the set is defined.

Definition 5.1 *A* **metric space** *is a pair* (M, d) *consisting of a set* M *and a* **metric** d *on* M*. The metric* $d(x, y)$ *defined for arbitrary* x *and* y *in* M *is a nonnegative, real valued function satisfying for all* $x, y, z \in M$ *the conditions:*

1. $d(x, y) = 0$ *if and only if* $x = y$,
2. $d(x, y) = d(y, x)$,

3. $d(x,z) \leq d(x,y) + d(y,z)$.

(M,d) *is said to be an* **ultra-metric space** *if* d *satisfies the strong triangle inequality: For all* $x,y,z \in M$, $d(x,z) \leq \max\{d(x,y), d(y,z)\}$.

We note that for all $x,y,z \in M$,

$$d(x,z) \leq \max\{d(x,y), d(y,z)\} \Rightarrow d(x,y) + d(y,z) \geq d(x,z).$$

The notion of complete metric spaces is based on *Cauchy sequences* defined as follows.

Definition 5.2 $(x_n)_n$ *is a* **Cauchy sequence** *in the space* (M,d) *if*

$$\forall \epsilon > 0\, \exists N\, \forall n,m > N : d(x_n, x_m) < \epsilon.$$

Definition 5.3 *If every Cauchy sequence in the metric space* M *converges to an element in* M, M *is said to be* **complete**.

Note that the space containing M, together with all limits of its Cauchy sequences is a *completion* of M, where the distance between the limit points $x^* = \lim_{n\to\infty} x_n$ and $y^* = \lim_{n\to\infty} y_n$ of M is defined as $d(x^*, y^*) = \lim_{n\to\infty} d(x_n, y_n)$.

Given a metric space (M,d), we define the metric d' on the set M^n $(n \geq 1)$ as follows.

Definition 5.4 *Let* (M,d) *be a metric space. Let* $X, Y \in M^n$ *for some* $n \geq 1$, $X = [X_1, \ldots, X_n]$, $Y = [Y_1, \ldots, Y_n]$. *Then*

$$d'(X,Y) = \max_{i \leq n} d(X_i, Y_i).$$

Then the pair (M^n, d') constitutes a complete metric space if (M,d) does.

Proposition 5.5 *If* (M,d) *is complete then so is* (M^n, d') *for all* $n \geq 1$.

By using Banach's fixed point theorem, one can guarantee the existence and uniqueness of *fixed points* of *contraction mappings* in complete metric spaces. These notions are given formally as in the following:

Definition 5.6 *An element* $x \in X$ *is said to be a* **fixed point** *of a function* $f : X \to X$ *if* $f(x) = x$.

Definition 5.7 *Let* (X,d) *be a metric space. A function* $f : X \to X$ *is a* **contraction mapping** *if there is a real number* $c < 1$ *such that* $d(f(x), f(y)) < c \cdot d(x,y)$ *for each* $x, y \in X$.

Theorem 5.8 *(***Banach's fixed point theorem***, see [59]) Every contraction mapping of a complete metric space has a unique fixed point.*

Complete partial orders

Complete partial orders are special classes of partially ordered sets. These orders are characterized by a completeness property which essentially says that every monotone sequence has a supremum. Formally:

Definition 5.9 *Let $\sqsubseteq$ be a partial order in a set $\mathcal{D}$. A* **monotone sequence** *$(P_n)_n$ in $\mathcal{D}$ is a sequence satisfying*

$$P_0 \sqsubseteq P_1 \sqsubseteq \cdots \sqsubseteq P_n \sqsubseteq P_{n+1} \sqsubseteq \cdots$$

Definition 5.10 *A* **complete partial order** *(cpo) $\mathcal{D} = (\mathcal{D}, \sqsubseteq)$ is a partially ordered set with a least element such that every monotone sequence has a supremum in $\mathcal{D}$.*

Compatibility between metric spaces and cpo's

A complete partial order and a metric space can be compared by the notion of *compatibility* [9]. More precisely, a complete partial order and a complete metric space of the same set are compatible if the supremum and the limit of every monotone Cauchy sequence are identified.

Definition 5.11 *A cpo $(\mathcal{D}, \sqsubseteq)$ and a complete metric space $(\mathcal{M}, d)$ are said to be* **compatible** *if $\mathcal{D} = \mathcal{M}$ and $\bigsqcup_n x_n = \lim_{n\to\infty} x_n$ for each monotone Cauchy sequence $(x_n)_n$.*

5.2.2 Basic thread algebra (BTA) and thread algebra (TA)

We recall from [22, 23, 25] the notions of basic thread algebra (BTA) and thread algebra (TA). We note that BTA was introduced as *basic polarized process algebra* (BPPA) in [22].

Basic thread algebra as a cpo

Let Σ be a set of actions. Each action returns a boolean value after its execution. Basic thread algebra (BTA) is defined by the following operators:

- **Termination**: $S \in$ BTA yields the terminating behavior.
- **Inactive behavior**: $D \in$ BTA, represents the inactive behavior.
- **Postconditional composition**: $(-) \trianglelefteq a \trianglerighteq (-)$ with $a \in \Sigma$. The thread $P \trianglelefteq a \trianglerighteq Q \in$ BTA with $P, Q \in$ BTA first performs a and then proceeds with P if true was returned or with Q otherwise. In case $P = Q$ we abbreviate this thread by the **action prefix** operator: $a \circ (-)$. In particular, $a \circ P = P \trianglelefteq a \trianglerighteq P$.

To provide domains for BTA, we consider the following domain equation

$$\mathcal{P} = \{S, D\} \bigcup (\mathcal{P} \trianglelefteq \Sigma \trianglerighteq \mathcal{P}). \tag{5.1}$$

where $X \trianglelefteq \Sigma \trianglerighteq Y = \{x \trianglelefteq a \trianglerighteq y | x \in X, y \in Y, a \in \Sigma\}$. We say that a solution of (5.1) is a domain of BTA.

Let BTA_Σ be the set of finite threads in BTA defined as follows.

Definition 5.12 BTA_Σ *is a set consisting of all finite threads which are made from* S *and* D *by means of a finite number of applications of postconditional compositions.*

Proposition 5.13 BTA_Σ *is a solution of (5.1), and therefore, it is a domain of* BTA.

Proof: It is obvious that $S, D \in \mathrm{BTA}_\Sigma$. If P and Q in BTA_Σ and $a \in \Sigma$ then $P \trianglelefteq a \trianglerighteq Q$ is also in BTA_Σ. Vice versa, if $R = P \trianglelefteq a \trianglerighteq Q \in \mathrm{BTA}_\Sigma$ then $P, Q \in \mathrm{BTA}_\Sigma$. □

Threads can be infinite. Infinite threads are given by sequences of finite approximations. In [16], a technique based on cpo's is described to give a domain for BTA. The main idea of this approach is to define a binary relation $\sqsubseteq$, a partial order, on threads. The expression $P \sqsubseteq Q$ means that P is an approximation of Q. It is shown that the set of *projective sequences* for threads is a cpo. This implies that it is a domain for BTA. Thus, it serves as a semantics for BTA in a natural way.

Definition 5.14

1. *The* **partial ordering** $\sqsubseteq$ *on* BTA_Σ *is generated by the clauses*
   (a) *for all* $P \in \mathrm{BTA}_\Sigma$, $D \sqsubseteq P$, *and*
   (b) *for all* $P, Q, X, Y \in \mathrm{BTA}_\Sigma$, $a \in \Sigma$,
   $$P \sqsubseteq X \,\&\, Q \sqsubseteq Y \Rightarrow P \trianglelefteq a \trianglerighteq Q \sqsubseteq X \trianglelefteq a \trianglerighteq Y.$$
2. *Let* $(P_n)_n$ *and* $(Q_n)_n$ *be two sequences in* BTA_Σ, *then*
   $$(P_n)_n \sqsubseteq (Q_n)_n \Leftrightarrow \forall n \in \mathbb{N} : P_n \sqsubseteq Q_n.$$

In order to define a projective sequence in BTA_Σ, an operator called the *approximation operator* that finitely approximates every thread is provided.

Definition 5.15 *For every* $n \in \mathbb{N}$, *the* **approximation operator** $\pi_n : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *is defined inductively by*

$$\begin{array}{lcl} \pi_0(P) & = & D, \\ \pi_{n+1}(S) & = & S, \\ \pi_{n+1}(D) & = & D, \\ \pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q), \end{array}$$

A **projective sequence** *is a sequence* $(P_n)_{n\in\mathbb{N}}$ *such that for each* $n \in \mathbb{N}$,

$$\pi_n(P_{n+1}) = P_n.$$

One can show that:

Lemma 5.16 *Every projective sequence is monotone.*

The following lemma gives an intuition for the projective approximations of a thread. Given $k, n \in \mathbb{N}$ with $k \leq n$, the k-th and the n-th projective approximations of a thread do not differ in behavior until the k-th step.

Lemma 5.17 *Let* $(P_n)_{n\in\mathbb{N}}$ *be a projective sequence. Then for all* $k \leq n$, $P_k = \pi_k(P_n)$.

Proof: This can be proven by induction on n. □
Let $\mathrm{BTA}_\Sigma^\infty$ be the set of projective sequences.

Definition 5.18 $\mathrm{BTA}_\Sigma^\infty = \{(P_n)_{n\in\mathbb{N}} | (P_n)_n$ *is a projective sequence in* $\mathrm{BTA}_\Sigma\}$

For a thread P represented by a projective sequence $(P_n)_n$ in $\mathrm{BTA}_\Sigma^\infty$, we denote $\pi_n(P) = P_n$.

Theorem 5.19 *([16]).* $\mathrm{BTA}_\Sigma \subset \mathrm{BTA}_\Sigma^\infty$ *and* $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ *is a complete partial order.*

The theorem above indicates that the cpo $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ is a domain for BTA in which infinite threads can be represented as supremums of monotone sequences of their finite approximations.

Abstraction in BTA

Abstraction [73, 20, 8] plays an important role in process algebras. It allows a simpler view of a thread, ignoring internal details. In [17] abstraction is used to emulate the interaction between clients and servers, assuming that clients and servers are threads in BTA. We assume the existence of a *concrete internal action* $\mathtt{tau} \in \Sigma$ that does not have any side effects and always replies true after its execution. This action can be abstracted away by an operator called the *abstraction operator* defined as follows.

Definition 5.20 *Let* $\tau_{\mathtt{tau}} : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *be defined by*

$$\begin{array}{lcl}
\tau_{\mathtt{tau}}(S) & = & S, \\
\tau_{\mathtt{tau}}(D) & = & D, \\
\tau_{\mathtt{tau}}(P \trianglelefteq \mathtt{tau} \trianglerighteq Q) & = & \tau_{\mathtt{tau}}(P), \\
\tau_{\mathtt{tau}}(P \trianglelefteq a \trianglerighteq Q) & = & \tau_{\mathtt{tau}}(P) \trianglelefteq a \trianglerighteq \tau_{\mathtt{tau}}(Q) \quad (a \neq \mathtt{tau} \in \Sigma).
\end{array}$$

It is shown in [17] that the abstraction operator is monotone, i.e.:

Lemma 5.21 *For all* $P, Q \in \mathrm{BTA}_\Sigma$, $P \sqsubseteq Q \Rightarrow \tau_{\mathtt{tau}}(P) \sqsubseteq \tau_{\mathtt{tau}}(Q)$.

Lemma 5.21 suggests the definition of abstraction of an infinite thread P given as the supremum of a monotone sequence of threads below.

Definition 5.22 *Let* $(P_n)_n$ *be a monotone sequence of finite approximations of a thread* $P \in \mathrm{BTA}_\Sigma^\infty$. *Then* $\tau_{\mathtt{tau}}(P) = \bigsqcup_n \tau_{\mathtt{tau}}(P_n)$.

Thread algebra

Thread algebra is an extension of BTA, which is designed for strategic interleaving of parallel threads. A *single thread* is defined in BTA. A *thread vector* is a finite sequence of threads. *Strategic interleaving operators* turn a thread vector of arbitrary length into a single thread. This single thread obtained via a strategic interleaving operator is called a *multi-thread*. Thread algebra is meant to specify the collection of strategic interleaving operators, capturing essential aspects of multi-threading. For a simplification, in this chapter, we only consider the simplest interleaving strategy called the *cyclic interleaving operator* [23].

Let $\langle\rangle$ denote the empty sequence, $\langle x\rangle$ stands for a sequence of length one, and $\alpha \frown \beta$ for the concatenation of two sequences. We assume that the following identity holds: $\alpha \frown \langle\rangle = \langle\rangle \frown \alpha = \alpha$.

Definition 5.23 *The axioms for* **the cyclic interleaving operator** *on finite threads are given as follows:*

$$\begin{array}{lcl}
\|_{csi}\,(\langle\rangle) & = & S \\
\|_{csi}\,(\langle S\rangle \frown \alpha) & = & \|_{csi}\,(\alpha) \\
\|_{csi}\,(\langle D\rangle \frown \alpha) & = & S_D(\|_{csi}\,(\alpha)) \\
\|_{csi}\,(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) & = & \|_{csi}\,(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csi}\,(\alpha \frown \langle y\rangle)
\end{array}$$

where the auxiliary deadlock at termination operator S_D *turns termination into deadlock and is defined by*

$$\begin{array}{lcl}
S_D(S) & = & D \\
S_D(D) & = & D \\
S_D(x \trianglelefteq a \trianglerighteq y) & = & S_D(x) \trianglelefteq a \trianglerighteq S_D(y)
\end{array}$$

We note that for a thread vector of length one, the cyclic interleaving operator turns the thread vector into the single thread contained in it.

5.3 BTA as a complete ultra-metric space

In the previous section, we have seen that BTA can be modeled as a complete partial order. In this section, we follow [11] to give a metric denotational semantics for BTA.

We prove that the domain of BTA can be turned into a complete metric space in which the distance d between two threads that do not differ in behavior until the n-th step is at most 2^{-n}. We show that the complete metric space $(\mathrm{BTA}_\Sigma^\infty, d)$ consisting of projective sequences is an appropriate domain for BTA by proving:

1. Infinite threads in $(\mathrm{BTA}_\Sigma^\infty, d)$ are represented in a unique way.
2. $(\mathrm{BTA}_\Sigma^\infty, d)$ is compatible with the domain $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$.
3. $(\mathrm{BTA}_\Sigma^\infty, d)$ deals with abstraction in a natural way, in comparison with the domain of Cauchy sequences.
4. Finally, the specification of a regular thread in $(\mathrm{BTA}_\Sigma^\infty, d)$ determines a unique thread by using Banach's fixed point theorem.

5.3.1 The metric d between threads

We formally define a metric (or distance) d between two threads in BTA_Σ as follows.

Definition 5.24

1. $d(S, S) = 0$, $d(D, D) = 0$,
   $d(P, P') = 1$ *if* $P \in \{S, D\}$ *and* $P' \neq P$ *with* $P' \in \mathrm{BTA}_\Sigma$ *or vice versa,*
2. $d(P_1 \trianglelefteq a_1 \trianglerighteq P_2, Q_1 \trianglelefteq a_2 \trianglerighteq Q_2) =$
   $$\begin{cases} 1 & if \quad a_1 \neq a_2, \\ \frac{1}{2}\max\{d(P_1, Q_1), d(P_2, Q_2)\} & otherwise \end{cases}$$
   with $P_1, Q_1, P_2, Q_2 \in \mathrm{BTA}_\Sigma$.

According to Definition 5.24, the metric between two finite threads that do not differ in behavior until the n-th step is at most 2^{-n}.

Lemma 5.25 *Let* $P, Q \in \mathrm{BTA}_\Sigma$. *Then for all* $n \in \mathbb{N}$,

$$d(P, Q) \leq \frac{1}{2^n} \Leftrightarrow \pi_n(P) = \pi_n(Q).$$

Proof: This can be proven by induction on n. □

One can show that the set of finite threads with the metric d constitutes an ultra-metric space.

Proposition 5.26 *(*BTA_Σ, d*) is an ultra-metric space.*

This proposition suggests the completion $(\mathrm{BTA}_\Sigma^\omega, d)$ of the metric space (BTA_Σ, d) whose elements are the limits of all Cauchy sequences in (BTA_Σ, d).

Definition 5.27 $\mathrm{BTA}_\Sigma^\omega = \{(P_n)_{n\in\mathbb{N}} | (P_n)_n$ *is a Cauchy sequence in* $\mathrm{BTA}_\Sigma\}$

5.3.2 The uniqueness of threads in $(\mathrm{BTA}_\Sigma^\infty, d)$

In this section, we show that completion $(\mathrm{BTA}_\Sigma^\omega, d)$ of all limits of Cauchy sequences of finite threads and the metric space $(\mathrm{BTA}_\Sigma^\infty, d)$ of projective sequences achieve two equivalent domains for BTA. However, the domain $(\mathrm{BTA}_\Sigma^\infty, d)$ represents (infinite) threads in a unique way.

First of all, we show that the complete metric space $(\mathrm{BTA}_\Sigma^\omega, d)$ is a solution of (5.1).

Lemma 5.28 $\mathrm{BTA}_\Sigma^\omega = \{S, D\} \bigcup (\mathrm{BTA}_\Sigma^\omega \trianglelefteq \Sigma \trianglerighteq \mathrm{BTA}_\Sigma^\omega)$.

Proof:

1. ($\supseteq$): Since $\{S, D\} \subseteq \mathrm{BTA}_\Sigma$, $\{S, D\} \subseteq \mathrm{BTA}_\Sigma^\omega$. We prove that if $P, Q \in \mathrm{BTA}_\Sigma^\omega$ then $(P \trianglelefteq a \trianglerighteq Q) \in \mathrm{BTA}_\Sigma^\omega$. Since $P, Q \in \mathrm{BTA}_\Sigma^\omega$, $P = \lim_{n\to\infty} P_n$, $Q = \lim_{n\to\infty} Q_n$ for some Cauchy sequences $(P_n)_n$ and $(Q_n)_n$. It is not hard to see that $(P_n \trianglelefteq a \trianglerighteq Q_n)_n$ is also a Cauchy sequence and $P \trianglelefteq a \trianglerighteq Q = \lim_{n\to\infty} P_n \trianglelefteq a \trianglerighteq Q_n$. Thus, $P \trianglelefteq a \trianglerighteq Q \in \mathrm{BTA}_\Sigma^\omega$.

2. ($\subseteq$): If $P \in \mathrm{BTA}_\Sigma^\omega$ then $P = S$ or $P = D$ or $P = Q \trianglelefteq a \trianglerighteq R$, $Q, R \in \mathrm{BTA}_\Sigma^\omega$. We only consider the case $P \notin \{S, D\}$. Since $P \in \mathrm{BTA}_\Sigma^\omega$, $P = \lim_{n\to\infty} P_n$ for some Cauchy sequence $(P_n)_n$. Without lack of generality we can assume that for all n, $P_n = Q_n \trianglelefteq a \trianglerighteq R_n$. Since $(P_n)_n$ is a Cauchy sequence and $d(P_n, P_m) = \frac{1}{2}\max\{d(Q_n, Q_m), d(R_n, R_m)\}$, $(Q_n)_n$ and $(R_n)_n$ are also Cauchy sequences. Therefore, there exist Q and R in $\mathrm{BTA}_\Sigma^\omega$ such that $Q = \lim_{n\to\infty} Q_n$, $R = \lim_{n\to\infty} R_n$. Hence $P = Q \trianglelefteq a \trianglerighteq R$.

□

The previous lemma shows that the completion $(\mathrm{BTA}_\Sigma^\omega, d)$ is a domain for BTA in which an infinite thread can be represented by a class of Cauchy sequences with the same limit. It can be seen that these representations are equivalent to projective sequences of threads. More precisely, we show that the metric space $(\mathrm{BTA}_\Sigma^\infty, d)$ yields an equivalent domain in which all limits of Cauchy sequences are represented in a unique way. We will use some supporting results.

In the next lemma, we prove that every projective sequence is a Cauchy sequence.

Lemma 5.29 $(\mathrm{BTA}_\Sigma^\infty, d) \subseteq (\mathrm{BTA}_\Sigma^\omega, d)$.

Proof: Let $(P_n)_n$ be an element in $\mathrm{BTA}_\Sigma^\infty$. By Lemma 5.17, for all $m, n \in \mathbb{N}$, $m > n > 0$, $P_{n-1} = \pi_{n-1}(P_n) = \pi_{n-1}(P_m)$. Therefore, by Lemma 5.25, $d(P_n, P_m) \leq \frac{1}{2^{n-1}}$. This implies that $(P_n)_n$ is a Cauchy sequence. □

We now show that for every Cauchy sequence, there always exists a projective sequence having the same limit.

Lemma 5.30 $(\mathrm{BTA}_\Sigma^\omega, d) \subseteq (\mathrm{BTA}_\Sigma^\infty, d)$.

Proof: Let Q be an element in $(\mathrm{BTA}_\Sigma^\omega, d)$. We will show that there exists $P = (P_k)_k$ in $(\mathrm{BTA}_\Sigma^\infty, d)$ such that $P = Q$. Since Q is an element in $(\mathrm{BTA}_\Sigma^\omega, d)$, $Q = \lim_{k\to\infty} Q_k$ for some Cauchy sequence $(Q_k)_k$. By Definition 5.2, we have that

$$\forall \epsilon > 0\, \exists N \in \mathbb{N}\, \forall m, n > N : d(Q_m, Q_n) < \epsilon.$$

We choose a sequence $N_0, N_1, \ldots$ of natural numbers satisfying $\pi_k(\pi_{k+1}(Q_{N_{k+1}})) = \pi_k(Q_{N_k})$ inductively as follows.

- Let $\epsilon = \frac{1}{2^0}$. Then there exists $N_0 \in \mathbb{N}$ such that for all $m, n \geq N_0$, $d(Q_m, Q_n) < \frac{1}{2^0}$. It follows from Lemma 5.25 that for all $n \geq N_0$, $\pi_0(Q_n) = \pi_0(Q_{N_0})$.
- Assume that we have chosen the numbers $N_0, \ldots, N_k$ such that for all $n \geq N_k$, $\pi_k(Q_n) = \pi_k(Q_{N_k})$. We choose N_{k+1} as follows. Let $\epsilon = \frac{1}{2^{k+1}}$. Then there exists $N \in \mathbb{N}$ such that for all $m, n \geq N$, $d(Q_m, Q_n) < \frac{1}{2^{k+1}}$. Thus, by Lemma 5.25, for all $n \geq N$, $\pi_{k+1}(Q_n) = \pi_{k+1}(Q_N)$. Let $N_{k+1} = \max\{N_k, N\}$. Then by the induction hypothesis, $\pi_k(Q_{N_{k+1}}) = \pi_k(Q_{N_k})$. It follows from Lemma 5.17 that $\pi_k(\pi_{k+1}(Q_{N_{k+1}})) = \pi_k(Q_{N_{k+1}}) = \pi_k(Q_{N_k})$.

Let $P_k = \pi_k(Q_{N_k})$ for all $k \in \mathbb{N}$. Then $P = (P_k)_k$ is an element of $\mathrm{BTA}_\Sigma^\infty$. To see that $d(P, Q) = 0$, consider $m, n \in \mathbb{N}$ such that $m > \max\{N_n, n\}$. Then $\pi_n(Q_m) = \pi_n(Q_{N_n}) = P_n = \pi_n(P_m)$. Thus, $d(P_m, Q_m) < \frac{1}{2^n}$. Hence $\lim_{m\to\infty} d(P_m, Q_m) = 0$ or $d(P, Q) = 0$. □

It follows from Lemma 5.29 and Lemma 5.30 that the metric spaces $(\mathrm{BTA}_\Sigma^\omega, d)$ and $(\mathrm{BTA}_\Sigma^\infty, d)$ are equivalent. Formally:

Theorem 5.31 $(\mathrm{BTA}_\Sigma^\infty, d) = (\mathrm{BTA}_\Sigma^\omega, d)$.

In addition, pointwise equal threads in $(\mathrm{BTA}_\Sigma^\infty, d)$ are identified. That is, the approximations of two equivalent threads in $(\mathrm{BTA}_\Sigma^\infty, d)$ are equal at every step. To prove this, we will use an auxiliary lemma which essentially says that the distance between the n-th projective approximations P_n and Q_n of two infinite threads P and Q is monotone. Since these distances are less than or equal to 1, the distance of two (finite or infinite) threads is equal to the supremum of the distances between their n-th projective approximations.

Lemma 5.32 *For all $(P_n)_n, (Q_n)_n \in \mathrm{BTA}_\Sigma^\infty$, $d(P_n, Q_n)$ is a non-decreasing sequence. Therefore,*

$$\lim_{n\to\infty} d(P_n, Q_n) = \bigsqcup_{n\in\mathbb{N}} d(P_n, Q_n).$$

Proof: We show that for all $n \in \mathbb{N}$, $d(P_n, Q_n) \leq d(P_{n+1}, Q_{n+1})$. It follows from Lemma 5.17 that

$$d(P_n, Q_n) = d(\pi_n(P_{n+1}), \pi_n(Q_{n+1})) \leq d(P_{n+1}, Q_{n+1}).$$

□

The previous lemma implies the uniqueness of representations of infinite threads by projective sequences, i.e.:

Proposition 5.33 *Let P and Q be two threads in $(\mathrm{BTA}_\Sigma^\infty, d)$ which are represented by two projective sequences $(P_n)_n$ and $(Q_n)_n$, respectively. Then*

$$P = Q \Leftrightarrow \forall n \in \mathbb{N} : P_n = Q_n.$$

Proof: If $P_n = Q_n$ for all $n \in \mathbb{N}$ then $d(P, Q) = \lim_{n\to\infty} d(P_n, Q_n) = 0$. Therefore, $P = Q$. We now show that if $P = Q$ then $P_n = Q_n$ for all $n \in \mathbb{N}$. It follows from Lemma 5.32 that for all $n \in \mathbb{N}$, $d(P_n, Q_n) \leq d(P, Q) = 0$. Hence $d(P_n, Q_n) = 0$ or $P_n = Q_n$ for all $n \in \mathbb{N}$. □

5.3.3 Compatibility between $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ and $(\mathrm{BTA}_\Sigma^\infty, d)$

This section shows the compatibility of the two domains $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ and $(\mathrm{BTA}_\Sigma^\infty, d)$ based on complete partial orders and complete metric spaces for BTA. We will use the following lemma which states that given a monotone sequence of finite threads and a number $n \in \mathbb{N}$, there always exists a subsequence with the property that the n-th projective approximations of the threads in the subsequence do not differ.

Lemma 5.34 *Let $(P_n)_n$ be a monotone sequence of finite threads. Then*

$$\forall n \, \exists N \, \forall m > N : \pi_n(P_m) = \pi_n(P_N).$$

Proof: We distinguish two cases. If for all m, $P_m \in \{D, S\}$ then there exists a minimal N such that for all $m > N$, $P_m = P_N$. Thus, for all n, $\pi_n(P_m) = \pi_n(P_N)$. The other case is that there exists a minimal N_0 such that for all $m \geq N_0$, $P_m = Q_m \trianglelefteq a \trianglerighteq R_m$. It is not hard to see that $(Q_m)_m$ and $(R_m)_m$ are also monotone sequences. We note that for all $m < N_0$, $Q_m = R_m = D$. We employ induction on n.

1. If $n = 0$ then $N = 0$.
2. If $n > 0$ then for all $m \geq N_0$, $\pi_n(P_m) = \pi_{n-1}(Q_m) \trianglelefteq a \trianglerighteq \pi_{n-1}(R_m)$. Applying the induction hypothesis, there exist N_1 and N_2 such that for all $m > N_1$, $\pi_{n-1}(Q_m) = \pi_{n-1}(Q_{N_1})$ and for all $m > N_2$, $\pi_{n-1}(R_m) = \pi_{n-1}(R_{N_2})$. Let $N = \max\{N_0, N_1, N_2\}$. Then for all $m > N$, $\pi_n(P_m) = \pi_n(P_N)$.

Therefore, for all $n \in \mathbb{N}$, there exists $N \in \mathbb{N}$ such that for all $m > N$, $\pi_n(P_m) = \pi_n(P_N)$. □

Lemma 5.34 implies that a monotone sequence of finite threads is also a Cauchy sequence. Formally:

Lemma 5.35 *Every monotone sequence $(P_n)_n$ of finite threads is a Cauchy sequence. As a consequence, $\bigsqcup_n P_n = \lim_{n\to\infty} P_n$.*

Proof: Let $P = \bigsqcup_n P_n$. It follows from Lemma 5.34 and Lemma 5.25 that for all $n \in \mathbb{N}$, there exists $N \in \mathbb{N}$ such that for all $m > N$, $d(P_m, P) < \frac{1}{2^n}$. This implies that $\lim_{n\to\infty} P_n = P$. Hence, $(P_n)_n$ is a Cauchy sequence and $\bigsqcup_n P_n = \lim_{n\to\infty} P_n$. □
Hence, by Definition 5.11, the two domains of BTA based on complete partial orders and complete metric spaces are compatible.

Theorem 5.36 *$(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ and $(\mathrm{BTA}_\Sigma^\infty, d)$ are compatible.*

5.3.4 Abstraction in $(\mathrm{BTA}_\Sigma^\infty, d)$ and $(\mathrm{BTA}_\Sigma^\omega, d)$

This section shows an advantage of the domain $(\mathrm{BTA}_\Sigma^\infty, d)$ of projective sequences, in comparison with the domain $(\mathrm{BTA}_\Sigma^\omega, d)$ of Cauchy sequences. More precisely, the former can deal with abstraction in a natural way, while the latter cannot.

As we have seen in the previous section, the two domains $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ and $(\mathrm{BTA}_\Sigma^\infty, d)$ are compatible. As a result, for a projective sequence $(P_n)_n$, the monotone sequence $(\tau_{\mathtt{tau}}(P_n))_n$ has a limit. Hence, the abstraction of an (infinite) thread $P \in (\mathrm{BTA}_\Sigma^\infty, d)$ represented by the projective sequence $(P_n)_n$ can be defined as the limit of the sequence $(\tau_{\mathtt{tau}}(P_n))_n$. This definition coincides with the definition of abstraction of infinite threads in the domain $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$.

Proposition 5.37 *Let $(P_n)_n$ be a projective sequence representing a thread $P \in \mathrm{BTA}_\Sigma^\infty$. Then $\lim_{n\to\infty} \tau_{\mathtt{tau}}(P_n)$ exists. In particular,*

$$\lim_{n\to\infty} \tau_{\mathtt{tau}}(P_n) = \bigsqcup_{n\to\infty} \tau_{\mathtt{tau}}(P_n) = \tau_{\mathtt{tau}}(P).$$

Abstraction, however, cannot be defined by means of Cauchy sequences. In particular, abstraction is not continuous in $(\mathrm{BTA}_\Sigma^\omega, d)$.

Proposition 5.38 *There exists $P = \lim_{n\to\infty}(P_n)_n$ for a Cauchy sequence $(P_n)_n$ such that $\lim_{n\to\infty} \tau_{\mathtt{tau}}(P_n) \neq \tau_{\mathtt{tau}}(P)$.*

Proof: Let $(P_n)_n$ be defined as follows

$$(P_n)_n = D, \mathtt{tau} \circ S, \mathtt{tau}^2 \circ D, \ldots, \mathtt{tau}^{2n} \circ D, \mathtt{tau}^{2n+1} \circ S, \ldots$$

One can see that $(P_n)_n$ is a Cauchy sequence. Let $P = \lim_{n\to\infty} P_n$. Then $P \in \mathrm{BTA}_\Sigma^\omega$. Thus, there exists $\tau_{\mathtt{tau}}(P) \in \mathrm{BTA}_\Sigma^\omega$. However, the sequence

$$(\tau_{\mathtt{tau}}(P_n))_n = D, S, D, \ldots, D, S, \ldots$$

is not a Cauchy sequence. Thus, it does not have a limit in $\mathrm{BTA}_\Sigma^\omega$. Therefore, $\lim_{n\to\infty} \tau_{\mathtt{tau}}(P_n) \neq \tau_{\mathtt{tau}}(P)$. □

5.3.5 The uniqueness of regular threads in $(\mathrm{BTA}_\Sigma^\infty, d)$

When dealing with infinite threads in concurrency theories, besides the method of providing finite approximations of an (infinite) thread, there is another way to construct infinite threads by means of *guarded recursive specifications* [73, 20, 30]. The threads defined by these specifications are called *regular threads*. By using Banach's fixed point theorem, we can show the uniqueness of regular threads in the complete metric space $(\mathrm{BTA}_\Sigma^\infty, d)$. This suggests the existence of a domain consisting of regular threads for BTA.

Definition 5.39 *A thread P is* **regular** *if $P = E_1$, where E_1 is defined by a finite system of the form ($n \geq 1$):*

$$\{E_i = t_i | 1 \leq i \leq n, t_i = S \text{ or } t_i = D \text{ or } t_i = E_{il} \trianglelefteq a_i \trianglerighteq E_{ir}\}$$

with $E_{il}, E_{ir} \in \{E_1, \ldots, E_n\}$ and $a_i \in \Sigma$.

The finite system in the previous definition is called a *guarded recursive specification*. To show that this specification determines a unique thread, we consider the thread represented by it as a component of the solution of the equation $X = T(X)$, where the definition of T is given as follows.

Definition 5.40 *Let $T : (\mathrm{BTA}_\Sigma^\infty)^n \to (\mathrm{BTA}_\Sigma^\infty)^n$ be defined such that*

$$T = \lambda X.[t_1(X), \ldots, t_n(X)]$$

where

$$\begin{array}{lcll} t_i & = & \lambda X_1, \ldots, X_n.S & \text{or} \\ t_i & = & \lambda X_1, \ldots, X_n.D & \text{or} \\ t_i & = & \lambda X_1, \ldots, X_n.X_{il} \trianglelefteq a_i \trianglerighteq X_{ir} & \end{array}$$

with $X_{il}, X_{ir} \in \{X_1, \ldots, X_n\}$.

Given a complete metric space $(\mathrm{BTA}_\Sigma^\infty, d)$, we define the metric d' on $(\mathrm{BTA}_\Sigma^\infty)^n$ as in Definition 5.4, assuming that $\mathrm{BTA}_\Sigma^\infty = M$. Thus, by Proposition 5.5, the metric space $((\mathrm{BTA}_\Sigma^\infty)^n, d')$ is also complete. We now show that:

Theorem 5.41 *T has a unique fixed point.*

Proof: Let I be the set of all indexes i such that $t_i = X_{il} \trianglelefteq a_i \trianglerighteq X_{ir}$. Then $d(t_i(X), t_i(Y)) = 0$ if $i \notin I$, since $t_i(X)$ is a constant, and $d(t_i(X), t_i(Y)) = \frac{1}{2}\max\{d(X_{il}, Y_{il}), d(X_{ir}, Y_{ir})\}$ otherwise. Let X, Y be elements of $(\mathrm{BTA}_\Sigma^\infty)^n$. By Definition 5.4 we have

$$\begin{array}{lcl} d'(T(X), T(Y)) & = & \max_{i \leq n} d(t_i(X), t_i(Y)) \\ & = & \max_{i \in I}(\frac{1}{2}\max\{d(X_{il}, Y_{il}), d(X_{ir}, Y_{ir})\}) \\ & \leq & \frac{1}{2}\max_{i \leq n} d(X_i, Y_i) = \frac{1}{2}d'(X, Y). \end{array}$$

It follows from Definition 5.7 that T is a contraction mapping. Since $((\mathrm{BTA}_\Sigma^\infty)^n, d')$ is complete and by Banach's fixed point theorem, T has a unique solution. □
The previous theorem implies the uniqueness of regular threads. This suggests the domain consisting of regular threads in $(\mathrm{BTA}_\Sigma^\infty, d)$ given below.

Definition 5.42 BTA_Σ^r *is the set of regular threads in* $\mathrm{BTA}_\Sigma^\infty$.

Proposition 5.43 BTA_Σ^r *is a domain of* BTA*, and* $\mathrm{BTA}_\Sigma \subset \mathrm{BTA}_\Sigma^r \subset \mathrm{BTA}_\Sigma^\infty$.

Proof: It is straightforward that BTA_Σ^r is a domain of BTA, and $\mathrm{BTA}_\Sigma \subseteq \mathrm{BTA}_\Sigma^r \subseteq \mathrm{BTA}_\Sigma^\infty$. In the following, we give two examples to show the strictness of the inclusions.

1. $\mathrm{BTA}_\Sigma \subset \mathrm{BTA}_\Sigma^r$: Let $R = a \circ R$. Then $R \in \mathrm{BTA}_\Sigma^r$ but $R \notin \mathrm{BTA}_\Sigma$.
2. $\mathrm{BTA}_\Sigma^r \subset \mathrm{BTA}_\Sigma^\infty$: Let P be the thread taken from [27].

$$\begin{array}{lcl} P & = & a \circ Q_{1,0}, \\ Q_{i+1,j} & = & b \circ Q_{i,j+1}, \\ Q_{0,j} & = & a \circ Q_{j+1,0}. \end{array}$$

Then $P \in \mathrm{BTA}_\Sigma^\infty$ but $P \notin \mathrm{BTA}_\Sigma^r$ (P performs $a \circ b \circ a \circ b^2 \circ a \circ b^3 \circ \ldots$).

□

Our results show that $(\mathrm{BTA}_\Sigma^\infty, d)$ is an appropriate domain for BTA, called the **projective limit domain** of BTA.

5.4 Extending BTA with strategic interleaving operators to TA

In this section, we show that the domain $(\mathrm{BTA}_\Sigma^\infty, d)$ of BTA can be extended with the strategic interleaving operators of thread algebra. For simplicity, we will consider only the basic strategy, the cyclic interleaving operator in [23]. Other strategies can be done in the same way. We denote the extensions of BTA_Σ and $\mathrm{BTA}_\Sigma^\infty$ with strategic interleaving operators as TA_Σ and $\mathrm{TA}_\Sigma^\infty$, respectively. It will be shown that multi-threads in $\mathrm{TA}_\Sigma^\infty$ can be defined by means of Cauchy sequences in the domain $(\mathrm{TA}_\Sigma^\infty, d)$, but not by means of monotone sequences in the domain $(\mathrm{TA}_\Sigma^\infty, \sqsubseteq)$. We will provide the projective sequence for a multi-thread by the projective sequences of its components.

5.4.1 $(\mathrm{TA}_\Sigma^\infty, d)$ as an appropriate domain for TA

As followed from the previous section, the complete metric space $(\mathrm{TA}_\Sigma^\infty, d)$ contains all limits of Cauchy sequences of finite threads. Given a thread vector of some limits in $\mathrm{TA}_\Sigma^\infty$, we will show that the limit of the sequence, whose elements are the multi-threads obtained by vectors of the approximations of those limits via the cyclic interleaving operator, exists. We will use some supporting results.

The first auxiliary lemma shows the compositionality property of the deadlock at termination operator S_D.

Lemma 5.44 *Let $P_i \in \mathrm{TA}_\Sigma$ $(1 \leq i \leq m)$ be finite threads. Then*

$$S_D(\|_{csi}\ (\langle P_1\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle)) = \|_{csi}\ (\langle S_D(P_1)\rangle \curvearrowright \cdots \curvearrowright \langle S_D(P_m)\rangle)$$

Proof: This can be proven by induction on the lengths of threads. □

Next, we prove that the distance of two finite threads after turning termination into deadlock decreases.

Lemma 5.45 *Let P and Q be finite threads. Then $d(S_D(P), S_D(Q)) \leq d(P, Q)$.*

Proof: Straightforward. □

The following lemma shows that the distance of two multi-threads with the same length obtained via the cyclic interleaving operator is always less than or equal to the maximum distance of their corresponding components.

Lemma 5.46 *Let P_i and Q_i $(1 \leq i \leq m)$ be finite threads. Then*

$$d(\|_{csi}\ (\langle P_1\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle), \|_{csi}\ (\langle Q_1\rangle \curvearrowright \cdots \curvearrowright \langle Q_m\rangle)) \leq \max_{1\leq i\leq m}\{d(P_i, Q_i)\}.$$

Proof: We prove this lemma by induction on the length and the number of threads. Let $d = d(\|_{csi}\ (\langle P_1\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle), \|_{csi}\ (\langle Q_1\rangle \curvearrowright \cdots \curvearrowright \langle Q_m\rangle))$. We distinguish the following cases:

1. $P_1 \neq Q_1$ and ($P_1 \in \{S, D\}$ or $Q_1 \in \{S, D\}$). Then $d(P_1, Q_1) = 1$. Thus, $d \leq \max_{1\leq i\leq m}\{d(P_i, Q_i)\} = 1$.
2. $P_1 = Q_1 = S$ (or $P_1 = Q_1 = D$). Then $d = d(\|_{csi}\ (\langle P_2\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle), \|_{csi}\ (\langle Q_2\rangle \curvearrowright \cdots \curvearrowright \langle Q_m\rangle))$ (or $d = d(\|_{csi}\ (\langle S_D(P_2)\rangle \curvearrowright \cdots \curvearrowright \langle S_D(P_m)\rangle), \|_{csi}\ (\langle S_D(Q_2)\rangle \curvearrowright \cdots \curvearrowright \langle S_D(Q_m)\rangle))$). By the induction hypothesis and Lemma 5.45, $d \leq \max_{1\leq i\leq m}\{d(P_i, Q_i)\}$.
3. $P_1 = P' \trianglelefteq a \trianglerighteq P''$ and $Q_1 = Q' \trianglelefteq a \trianglerighteq Q''$. Then

$$\begin{aligned} d = \ & \tfrac{1}{2}\max\{ \\ & d(\|_{csi}\ (\langle P_2\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle \curvearrowright \langle P'\rangle), \|_{csi}\ (\langle Q_2\rangle \curvearrowright \cdots \curvearrowright \langle Q_m\rangle \curvearrowright \langle Q'\rangle)), \\ & d(\|_{csi}\ (\langle P_2\rangle \curvearrowright \cdots \curvearrowright \langle P_m\rangle \curvearrowright \langle P''\rangle), \|_{csi}\ (\langle Q_2\rangle \curvearrowright \cdots \curvearrowright \langle Q_m\rangle \curvearrowright \langle Q''\rangle)) \\ & \}. \end{aligned}$$

   By the induction hypothesis and $d(P_1, Q_1) = \frac{1}{2}\max\{d(P', Q'), d(P'', Q'')\}$, $d \leq \max_{1\leq i\leq m}\{d(P_i, Q_i)\}$.

□

Proposition 5.47 *Let $(P_n^k)_n$ be Cauchy sequences for $1 \leq k \leq m$. Then $(\|_{csi}\ (\langle P_n^1\rangle \frown \cdots \frown \langle P_n^m\rangle))_n$ is also a Cauchy sequence.*

Proof: By Definition 5.2, we have

$$\forall 1 \leq k \leq m\, \forall \epsilon > 0\, \exists N_k \in \mathbb{N}\, \forall i, j > N_k : d(P_i^k, P_j^k) < \epsilon.$$

Let $Q_n = \|_{csi}\ (\langle P_n^1\rangle \frown \cdots \frown \langle P_n^m\rangle)$ for all $n \in \mathbb{N}$ and $N = \max_{1\leq k\leq m}\{N_k\}$. It follows from Lemma 5.46 that

$$\forall \epsilon > 0\, \exists N \in \mathbb{N}\, \forall i, j > N : d(Q_i, Q_j) < \epsilon.$$

Therefore, $(Q_n)_n$ is a Cauchy sequence. □

Proposition 5.47 suggests a definition for multi-threads obtained by thread vectors in $(\mathrm{TA}_\Sigma^\infty, d)$ via the cyclic interleaving operator $\|_{csi}\ (-)$ as follows.

Definition 5.48 *Let $P_j = \lim_{n\to\infty} P_n^j$ $(1 \leq j \leq m)$ be threads in $\mathrm{TA}_\Sigma^\infty$, where $(P_n^j)_n$ $(1 \leq j \leq m)$ are Cauchy sequences. Then*

$$\|_{csi}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle) = \lim_{n\to\infty} \|_{csi}\ (\langle P_n^1\rangle \frown \cdots \frown \langle P_n^m\rangle)$$

We note that these multi-threads cannot be defined by means of monotone sequences as can be seen in the following example.

Example 5.49 Let $(P_n)_n$ and $(Q_n)_n$ be monotone sequences of finite threads defined as follows: $P_0 = D$, $P_n = a \circ D$ for all $n > 0$ and $Q_n = b \circ D$ for all $n \geq 0$. Let $R_n = \|_{csi}\ (\langle P_n\rangle \frown \langle Q_n\rangle)$ for all $n \geq 0$. Then the supremum of the sequence $(R_n)_n$ does not exist, since it is not monotone as $R_0 = b \circ D$ and $R_1 = a \circ b \circ D$.

5.4.2 Projective sequences of multi-threads

This section shows that the projective sequence of a multi-thread in $\mathrm{TA}_\Sigma^\infty$ can be computed by the projective sequences of its components.

First of all, we prove that two multi-threads are the same in behavior until the n-th step if their corresponding components also are.

Lemma 5.50 *Let P_i be single threads in $\mathrm{TA}_\Sigma^\infty$ for all $1 \leq i \leq m$. Then*

$$\pi_n(\|_{csi}\ (\langle \pi_n(P_1)\rangle \frown \cdots \frown \langle \pi_n(P_m)\rangle))) = \pi_n(\|_{csi}\ (\langle \pi_{i_1}(P_1)\rangle \frown \cdots \frown \langle \pi_{i_m}(P_m)\rangle)))$$

with $i_j \geq n$ for all $1 \leq j \leq m$.

Proof: This can be proven by induction on n and m. □
We now show that a multi-thread and the multi-thread obtained by the n-th projective approximations of its components do not differ until the n-th step. This property allows us to compute the projective sequence of a multi-thread by the projective sequences of its components.

Theorem 5.51 *Let P_i be single threads in $\mathrm{TA}_\Sigma^\infty$ for all $1 \leq i \leq m$. Then*

$$\pi_n(\|_{csi}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle)) = \pi_n(\|_{csi}\ (\langle \pi_n(P_1)\rangle \frown \cdots \frown \langle \pi_n(P_m)\rangle))).$$

Proof: Let $Q = \|_{csi}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle)$ and $Q_n = \|_{csi}\ (\langle \pi_n(P_1)\rangle \frown \cdots \frown \langle \pi_n(P_m)\rangle)$. We show that $\pi_n(Q) = \pi_n(Q_n)$ for all $n \in \mathbb{N}$. It follows from Lemma 5.50 that $(\pi_n(Q_n))_n$ is a projective sequence. Since $d(\pi_n(Q_n), Q_n) \leq \frac{1}{2^n}$, $\lim_{n\to\infty} \pi_n(Q_n) = \lim_{n\to\infty} Q_n = Q$. Thus, $\pi_n(Q_n)$ is a projective sequence of Q. Hence, $\pi_n(Q) = \pi_n(Q_n)$ for all $n \in \mathbb{N}$. □

5.5 An interleaving strategy with respect to abstraction

We have introduced and discussed abstraction of single threads in Section 5.2.2 and Section 5.3.4. It would be natural if abstraction is compositional with respect to the interleaving strategies in [23]. Unfortunately, this property does not hold for the cyclic interleaving operator, the basic strategy of the interleaving strategies in [23], as can be seen in the following example.

Example 5.52 Let $P = \mathtt{tau} \circ a \circ S$ and $Q = b \circ S$ be two single threads. Then $\|_{csi}\ (\langle P\rangle \frown \langle Q\rangle) = \mathtt{tau} \circ b \circ a \circ S$. One can see that $\tau_{\mathtt{tau}}(\|_{csi}\ (\langle P\rangle \frown \langle Q\rangle))\ (= b \circ a \circ S)$ and $\|_{csi}\ (\langle \tau_{\mathtt{tau}}(P)\rangle \frown \langle \tau_{\mathtt{tau}}(Q)\rangle)\ (= a \circ b \circ S)$ are not equal.

In this section, we propose a variant of the cyclic interleaving operator called the *cyclic internal persistence* operator for thread algebra. We will show that this interleaving strategy deals with abstraction in a natural way.

5.5.1 The cyclic internal persistence operator

The phrase *cyclic internal persistence* means that upon the execution of a thread vector, the internal action $\mathtt{tau}$ is persistent. That is, its execution will not invoke the rotation of the thread vector. The definition of the cyclic internal persistence strategy is given below.

Definition 5.53 *The axioms for the* **cyclic internal persistence** *operator* $\|_{cip}\ (-)$ *on finite threads are given by*

$$\begin{array}{lcl}
\|_{cip}\ (\langle\rangle) & = & S \\
\|_{cip}\ (\langle S\rangle \frown \alpha) & = & \|_{cip}\ (\alpha) \\
\|_{cip}\ (\langle D\rangle \frown \alpha) & = & S_D(\|_{cip}\ (\alpha)) \\
\|_{cip}\ (\langle \mathtt{tau} \circ x\rangle \frown \alpha) & = & \mathtt{tau} \circ \|_{cip}\ (\langle x\rangle \frown \alpha) \\
\|_{cip}\ (\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) & = & \|_{cip}\ (\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csi}\ (\alpha \frown \langle y\rangle)
\end{array}$$

Like the cyclic interleaving operator, we define the cyclic internal operator on infinite threads as follows.

Definition 5.54 *Let* $P_j = \lim_{n\to\infty} P_n^j$ *(*$1 \leq j \leq m$*) be threads in* $\mathrm{TA}_\Sigma^\infty$*, where* $(P_n^j)_n$ *(*$1 \leq j \leq m$*) are Cauchy sequences. Then*

$$\|_{cip}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle) = \lim_{n\to\infty} \|_{cip}\ (\langle P_n^1\rangle \frown \cdots \frown \langle P_n^m\rangle)$$

Furthermore, one can approximate the multi-threads obtained via the cyclic internal persistence operator by the projective approximations of its components.

Theorem 5.55 *Let* P_i *be single threads in* $\mathrm{TA}_\Sigma^\infty$ *for all* $1 \leq i \leq m$*. Then*

$$\pi_n(\|_{cip}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle)) = \pi_n(\|_{cip}\ (\langle \pi_n(P_1)\rangle \frown \cdots \frown \langle \pi_n(P_m)\rangle))).$$

Proof: Similar to the proof of Theorem 5.51. □

5.5.2 Compositionality of abstraction with respect to the cyclic internal persistence strategy

This section shows that abstraction satisfies compositionality with respect to the cyclic internal persistence operator, provided that threads cannot perform an infinite sequence of internal actions. The condition suggests an approximation operator $\pi_n^{\mathtt{tau}}(-)$ which respects concrete internal actions. This operator only takes the performance of non-internal actions into account.

Definition 5.56 *The approximation operator with respect to* $\mathtt{tau}$ $\pi_n^{\mathtt{tau}} : \mathrm{TA}_\Sigma \to \mathrm{TA}_\Sigma$ *is defined on finite threads by*

$$\begin{array}{lcl}
\pi_0^{\mathtt{tau}}(P) & = & D, \\
\pi_{n+1}^{\mathtt{tau}}(S) & = & S, \\
\pi_{n+1}^{\mathtt{tau}}(D) & = & D, \\
\pi_{n+1}^{\mathtt{tau}}(\mathtt{tau} \circ P) & = & \mathtt{tau} \circ \pi_{n+1}^{\mathtt{tau}}(P) \\
\pi_{n+1}^{\mathtt{tau}}(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n^{\mathtt{tau}}(P) \trianglelefteq a \trianglerighteq \pi_n^{\mathtt{tau}}(Q)
\end{array}$$

A **projective sequence with respect to** $\mathtt{tau}$ *is a sequence* $(P_n)_{n\in\mathbb{N}}$ *such that for each* $n \in \mathbb{N}$*,*

$$\pi_n^{\mathtt{tau}}(P_{n+1}) = P_n.$$

One can prove that every projective sequence with respect to $\mathtt{tau}$ is monotone, and therefore, its supremum is in $(\mathrm{TA}_\Sigma^\infty, \sqsubseteq)$. Let $\mathrm{TA}_\Sigma^{\mathtt{tau}}$ be the set of the threads represented by these projective sequences given formally as follows.

Definition 5.57

$$\mathrm{TA}_\Sigma^{\mathtt{tau}} = \{(P_n)_n | \, (P_n)_n \textit{ is a projective sequence with respect to } \mathtt{tau}\}$$

For a thread $P \in \mathrm{TA}_\Sigma^{\mathtt{tau}}$ represented by a projective sequence $(P_n)_n$ with respect to $\mathtt{tau}$, we denote $\pi_n^{\mathtt{tau}}(P) = P_n$.

In the following lemma, we will see that the abstraction of the n-th projective approximation with respect to $\mathtt{tau}$ of a finite thread coincides with the n-th projective approximation of its abstraction.

Lemma 5.58 *Let P be a finite thread. Then for all $n \in \mathbb{N}$,*

$$\tau_{\mathtt{tau}}(\pi_n^{\mathtt{tau}}(P)) = \pi_n(\tau_{\mathtt{tau}}(P)).$$

Proof: This can be proven by induction on n. □

By using the approximation operator $\pi_n^{\mathtt{tau}}(-)$, one can approximate a multi-thread in $\mathrm{TA}_\Sigma^{\mathtt{tau}}$ obtained via the cyclic internal persistence strategy, by the approximations of its components as follows.

Theorem 5.59 *Let P_i $(1 \leq i \leq m)$ be threads in $\mathrm{TA}_\Sigma^{\mathtt{tau}}$. Then*

$$\|_{cip}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle) = \bigsqcup_n \pi_n^{\mathtt{tau}}(\|_{cip}\ (\langle \pi_n^{\mathtt{tau}}(P_1)\rangle \frown \cdots \frown \langle \pi_n^{\mathtt{tau}}(P_m)\rangle))$$

Proof: Let $Q = \|_{csi}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle)$ and $Q_n = \|_{csi}\ (\langle \pi_n^{\mathtt{tau}}(P_1)\rangle \frown \cdots \frown \langle \pi_n^{\mathtt{tau}}(P_m)\rangle)$. Similar to the proof of Theorem 5.51, one can show that the sequence $(\pi_n^{\mathtt{tau}}(Q_n))_n$ is a projective sequence with respect to $\mathtt{tau}$. Therefore, $\bigsqcup_n(\pi_n^{\mathtt{tau}}(Q_n)) = \lim_n(\pi_n^{\mathtt{tau}}(Q_n)) = \lim_n Q_n = Q$. □

Finally, abstraction is compositional with respect to the cyclic internal persistence operator, provided that threads cannot perform an infinite sequence of internal actions.

Theorem 5.60 *Let P_i $(1 \leq i \leq m)$ be threads in $\mathrm{TA}_\Sigma^{\mathtt{tau}}$. Then*

$$\tau_{\mathtt{tau}}(\|_{cip}\ (\langle P_1\rangle \frown \cdots \frown \langle P_m\rangle)) = \|_{cip}\ (\langle \tau_{\mathtt{tau}}(P_1)\rangle \frown \cdots \frown \langle \tau_{\mathtt{tau}}(P_m)\rangle)$$

Proof: We consider two possibilities:

1. The threads P_i $(1 \leq i \leq m)$ are finite. The theorem can be proven by induction on the length of threads.

2. The threads P_i $(1 \leq i \leq m)$ are infinite. Let $P = (\|_{csi}\ (\langle P_1 \rangle \frown \cdots \frown \langle P_m \rangle))$. It follows from Theorem 5.55, Theorem 5.59, Lemma 5.58 and the previous case that

$$\begin{aligned}
&\tau_{\mathtt{tau}}(P)\\
&= \bigsqcup_n \tau_{\mathtt{tau}}(\pi_n^{\mathtt{tau}}(\|_{cip}\ (\langle \pi_n^{\mathtt{tau}}(P_1) \rangle \frown \cdots \frown \langle \pi_n^{\mathtt{tau}}(P_m) \rangle))))
&&\text{(by Theorem 5.59)}\\
&= \bigsqcup_n \pi_n(\tau_{\mathtt{tau}}(\|_{cip}\ (\langle \pi_n^{\mathtt{tau}}(P_1) \rangle \frown \cdots \frown \langle \pi_n^{\mathtt{tau}}(P_m) \rangle))))
&&\text{(by Lemma 5.58)}\\
&= \bigsqcup_n \pi_n(\|_{csi}\ (\langle \tau_{\mathtt{tau}}(\pi_n^{\mathtt{tau}}(P_1)) \rangle \frown \cdots \frown \langle \tau_{\mathtt{tau}}(\pi_n^{\mathtt{tau}}(P_m)) \rangle)))
&&\text{(by 1.)}\\
&= \bigsqcup_n \pi_n(\|_{cip}\ (\langle \pi_n(\tau_{\mathtt{tau}}(P_1)) \rangle \frown \cdots \frown \langle \pi_n(\tau_{\mathtt{tau}}(P_m)) \rangle)))
&&\text{(by Lemma 5.58)}\\
&= \|_{cip}\ (\langle \tau_{\mathtt{tau}}(P_1) \rangle \frown \cdots \frown \langle \tau_{\mathtt{tau}}(P_m) \rangle)
&&\text{(by Theorem 5.55)}
\end{aligned}$$

□

5.6 Concluding remarks

We have studied a metric denotational semantics for thread algebra. We have shown that the projective limit domain $(\mathrm{BTA}_\Sigma^\infty, d)$ is an appropriate domain for BTA. In particular, this domain represents infinite threads in a unique way. Furthermore, it is compatible with the domain based on cpo's in [16]. Moreover, it deals naturally with abstraction. As a consequence of Banach's fixed point theorem, the specification of a regular thread yields a unique thread. In the setting of multi-threads, we have shown that $(\mathrm{BTA}_\Sigma^\infty, d)$ can be extended with the cyclic interleaving operator. The extension of $(\mathrm{BTA}_\Sigma^\infty, d)$ with the other strategic interleaving operators in [23] can be done in the same way. We have also proposed an interleaving strategy with respect to abstraction, namely the cyclic internal persistence operator, for thread algebra.

Part II

Program algebra

Chapter 6

Projection semantics for while-languages in program algebra

6.1 Introduction

Program algebra (PGA) is a simple programming language for the study of sequential programming [22]. The aim of [22] is to provide a better understanding of sequential programming and to answer the question "What is a programming language?", since there is no a proper definition of a computer program in the literature [14]. In [22], programs are defined as expressions in PGA. Based on PGA, more complex programming languages can be developed and studied, by adding simple and general constructs. In [82], Ponse provides a flexible style of programming by extending PGA with the *unit instruction operator* to the program notation PGAu. This operator takes a program and wraps it into a unit of length one. By defining a projection from PGAu to PGA, Ponse shows that the flexibility does not come with an increase in the expressiveness.

The extensions of PGA with conditional statements and while-loops, as defined in [22], are also natural extensions for PGA. Apparently, with the use of these instructions, PGA is close to the programming languages that are used in practice. These instructions are introduced in [22] with two semantics: a full projection semantics and a lazy projection semantics. The extension PGLEcw of PGA with conditional and while-loop instructions respecting the full projection semantics has been studied in [22]. In this chapter, we investigate extensions of PGAu with conditional and while-loop instructions respecting the lazy projection semantics. This semantics is less flexible than the full projection semantics. However, it is much simpler. We will explore the expressive power of the programming language generated by PGAucw (PGA extended with unit, conditional and while-loop instructions), in comparison with that of PGA.

For simplicity, we will first study the extension PGAuc of PGAu with conditional instructions. We present a projection from PGAuc to PGAu, and prove correctness for this projection.

In the study of the extension PGAucw of PGAuc with while-loops, we aim to provide a projection from PGAucw to PGA. It turns out that the lazy projective semantics of while-loops produces non-regular behaviors in certain cases. Under the restriction that no consecutive occurrence of while-loops is allowed, we present a projection from PGAucw to PGLBu, a variant of PGAu containing backward jumps given in [82]. The projection from PGAucw to PGA is a composition of this projection and the projection from PGLBu to PGA covered in [82]. The correctness of our projections implies that the conditional and while-loop instructions are not needed as primitive instructions in terms of expressiveness, provided that consecutive occurrences of while-loops are forbidden.

The structure of this chapter is as follows. Section 6.2 recalls the basic concepts of PGA, BTA and summarizes a projection semantics for PGAu. Section 6.3 defines behavior extraction equations on positions for primitive instructions in PGAu and PGLBu. In Section 6.4, a projection `pgauc2pgau` from PGAuc to PGAu is defined. In Section 6.5, under the restriction of no consecutive occurrences of while-loops, a projection `pgaucw2pga` from PGAucw to PGA is described in detail and its correctness is proved. Section 6.6 draws some conclusions.

6.2 PGA, BTA and PGAu

This section recalls some basic concepts of PGA, BTA and PGAu from [22, 25] and [82].

6.2.1 Program algebra

Program notation syntax

Program algebra (PGA) is defined over a set Σ of *basic instructions*. Each basic instruction returns a boolean value upon execution. The collection of program expressions in PGA over Σ, denoted by PGA_Σ, is generated by *primitive instructions* and two *composition constructs*. These primitive instructions are defined by

Basic instruction All $a \in \Sigma$ are basic instructions. Upon execution of a basic instruction, a boolean value is generated and a state may be modified. After execution, its subsequent instruction is performed.

Termination instruction Termination instruction, denoted by !, indicates termination of the program. (It does not modify the state and does not return a boolean value.)

Positive test instruction For each $a \in \Sigma$, there is a positive test instruction, denoted by $+a$. If $+a$ is performed by a program then first a is executed. The

state is affected according to a. In case true is returned, the subsequent instruction is performed. In case false is returned, the next instruction is skipped and the execution continues with the following instruction.

Negative test instruction For each $a \in \Sigma$, there also exists a negative test instruction, denoted by $-a$. If $-a$ is performed by a program then first a is executed. The state is affected according to a. In case false is returned, the subsequent instruction is executed. In case true is returned, the next instruction is skipped and the execution proceeds with the following instruction.

Forward jump instruction For any natural number k, there is an instruction $\#k$ which denotes a jump of length k. The number k is the counter of the jump instruction.

- If $k = 0$, the jump is to itself (zero steps forward). In this case inaction will result.
- If $k = 1$, the subsequent instruction is performed.
- If $k > 1$, the execution skips the next $k - 1$ instructions. The instruction after that will be performed.

If there is no instruction to be executed, inaction will occur. The two composition constructs are defined by

Concatenation The concatenation of two programs X and Y in PGA_Σ, denoted by $X; Y$, is also in PGA_Σ.

Repetition The repetition of a program X in PGA_Σ, denoted by X^ω, is also in PGA_Σ.

Typical program expressions in PGA are given in the following example.

Example 6.1 Consider the two programs below:

$$\begin{array}{lcl} X & ::= & +a; \#2; b; ! \\ Y & ::= & -a; !; (b)^\omega. \end{array}$$

Program X first performs action a. If true is returned then it jumps two steps and terminates otherwise it skips the jump instruction $\#2$ and performs action b. After the execution of b it terminates. Program Y first performs action a as well. If true is returned then it performs action b repeatedly otherwise it terminates.

A program in PGA is *finite* if it does not contain a repetition, otherwise it is *infinite*. Formally:

Definition 6.2 *A program expression X is* **finite** *if it has the form $u_1; \ldots; u_k$ for primitive instructions $u_1, \ldots, u_k$.*

According to the definition above, program X in Example 6.1 is finite, while program Y is infinite.

Instruction sequence congruence and canonical forms

If two program expressions can be shown to be equal by means of the *program object equations* in Table 6.1, the program expressions are said to be instruction sequence equivalent. Here $X^1 = X$, $X^{n+1} = X; X^n$, n is a positive integer.

$(X;Y);Z$	$=$	$X;(Y;Z)$	(PGA1)
$(X^n)^\omega$	$=$	X^ω	(PGA2)
$X^\omega;Y$	$=$	X^ω	(PGA3)
$(X;Y)^\omega$	$=$	$X;(Y;X)^\omega$	(PGA4)

Table 6.1: Program object equations.

By these program object equations, the unfolding of a repetition can be obtained: $X^\omega = (X;X)^\omega = X;(X;X)^\omega = X;X^\omega$. Furthermore, each program expression in PGA_Σ can be written into one of the following *first canonical forms*.

Definition 6.3 *Let X be a program expression in* PGA. *Then X is in first canonical form iff*

- *X does not contain a repetition, or*
- *$X = Y;Z^\omega$, with Y and Z not containing a repetition.*

Note that in [22], Bergstra and Loots also provide the *second canonical form* for program expressions in PGA_Σ. For our purpose, we do not need to consider programs in this form.

6.2.2 Basic thread algebra

Behaviors of programs in PGA can be defined as *regular* threads in *basic thread algebra* (BTA) [22, 25], a semantics of sequential programming languages. We note that BTA was introduced as *basic polarized process algebra* (BPPA) in [22].

Primitives of BTA

We call basic instructions in Σ *actions*. Then finite threads in BTA are given as follows.

Definition 6.4 *Let BTA_Σ denote the set of finite threads over Σ. It is generated inductively by the following operators:*

- **Termination***: $S \in \mathrm{BTA}_\Sigma$.*
- **Inactive behavior***: $D \in \mathrm{BTA}_\Sigma$.*

- **Postconditional composition**: $(-) \trianglelefteq a \trianglerighteq (-)$ *with* $a \in \Sigma$. *The thread* $P \trianglelefteq a \trianglerighteq Q$ *with* $P, Q \in \mathrm{BTA}_\Sigma$ *first performs* a *and then proceeds with* P *if true was produced and with* Q *otherwise. In case* $P = Q$ *we abbreviate this thread by the* **action prefix**: $a \circ (-)$. *In particular,*

$$a \circ P = P \trianglelefteq a \trianglerighteq P.$$

Infinite threads

Threads can be infinite. An infinite thread in BTA is represented by a projective sequence consisting of its finite approximations. These approximations are determined by means of *approximation operators* $\pi_n(-)$ for $n \in \mathbb{N}$ given as follows.

Definition 6.5 *For every* $n \in \mathbb{N}$, *the* **approximation operator** $\pi_n : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *is defined inductively by*

$$\begin{array}{lcl} \pi_0(P) & = & D, \\ \pi_{n+1}(S) & = & S, \\ \pi_{n+1}(D) & = & D, \\ \pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q), \end{array}$$

and hence, $\pi_{n+1}(a \circ P) = a \circ \pi_n(P)$.
A **projective sequence** *is a sequence* $(P_n)_{n \in \mathbb{N}}$ *such that for each* $n \in \mathbb{N}$,

$$\pi_n(P_{n+1}) = P_n.$$

We say that two (finite or infinite) threads are equal exactly if for each $n \in \mathbb{N}$, their n-th approximations are equal. Let $\mathrm{BTA}_\Sigma^\infty$ be the set of projective sequences.

Regular threads

Regular threads in BTA are used to represent program behaviors in PGA.

Definition 6.6 *A thread* P *is* **regular** *over* Σ *if* $P = E_1$, *where* E_1 *is defined by a finite system of the form* $(n \geq 1)$*:*

$$\{E_i = t_i | 1 \leq i \leq n, t_i = S \text{ or } t_i = D \text{ or } t_i = E_{il} \trianglelefteq a_i \trianglerighteq E_{ir}\}$$

with $E_{il}, E_{ir} \in \{E_1, \ldots, E_n\}$ *and* $a_i \in \Sigma$.

The finite system in the definition above is called a *guarded recursive specification*.

Theorem 6.7 *A guarded recursive specification has a unique solution in* $\mathrm{BTA}_\Sigma^\infty$.

Proof: See Theorem 5.41. □

6.2.3 Assigning a thread in BTA to a program in PGA

This section assigns a regular thread in BTA to a program in PGA by means of *behavior extraction equations*. This suggests the notions of *behavioral equivalence* and *program algebra projections* that map programs of a programming language to behaviorally equivalent programs in PGA.

Behavior extraction equations

Program behaviors in PGA are given by means of an operator called the *behavior extraction operator* $|-|$ and *behavior extraction equations*.

Definition 6.8 *For a finite program X, the behavior is given by*

$$|X| = |X; (\#0)^\omega|.$$

Definition 6.9 *The behavior $|X|$ of an (infinite) program X is determined recursively by the* **behavior extraction equations** *below:*

$$\begin{aligned}
|!; X| &= S,\\
|a; X| &= a \circ |X|,\\
|+a; u; X| &= |u; X| \trianglelefteq a \trianglerighteq |X|,\\
|-a; u; X| &= |X| \trianglelefteq a \trianglerighteq |u; X|,\\
|\#0; X| &= D,\\
|\#1; X| &= |X|,\\
|\#(k+2); u; X| &= |\#(k+1); X|.
\end{aligned}$$

where u is a primitive instruction, $a \in \Sigma$ and $k \in \mathbb{N}$.

By means of these equations, successive steps of the behavior of a program can be obtained. In the case that a program has a non-trivial loop in which no action occurs, its behavior will be identified with D. Phrased differently: if for a behavior $|X|$ the behavior extraction equations fail to prove $|X| = S$ or $\pi_1(|X|) = a \circ D$ for some $a \in \Sigma$, then $|X| = D$.

Theorem 6.10 *Each regular thread in* $\mathrm{BTA}_\Sigma^\infty$ *can be specified in* PGA_Σ. *Vice versa, each program behavior specified in* PGA_Σ *is regular.*

Proof: See [30]. □

Behavioral equivalence

Behavioral equivalence identifies programs whose behaviors are the same.

Definition 6.11 *Two programs X and Y are* **behaviorally equivalent** *if $|X| = |Y|$.*

Program algebra transformations

Program algebra transformations are used to map a program in a programming language to a behaviorally equivalent one in another programming language. This notion allows us to study and develop more complex programming languages based on PGA.

Definition 6.12 *A* **transformation** *is a mapping φ from a programming language L_1 to another programming language L_2. This transformation φ is* **correct** *if for every $X \in L_1$, $|X|_{L_1} = |\varphi(X)|_{L_2}$ where $|-|_L$ is an assignment of behaviors to elements of L. This transformation is called a* **projection** *if $L_2 \subseteq L_1$, and an* **embedding** *if $L_1 \subseteq L_2$.*

We can write $|X|$ instead of $|X|_L$ if L is fixed.

6.2.4 PGAu and projecting PGAu into PGA

This section introduces the extension PGAu of PGA with the *unit* instruction and the projection from PGAu into PGA given in [82]. This extension allows for a more flexible style of programming, and can be used to study the program algebra itself.

The unit instruction operator

The *unit* instruction operator takes a PGA program and wraps it into a unit of length one, and can be regarded as a primitive instruction. This length matters in connection with the evaluation of jumps and tests in Definition 6.9.

Definition 6.13 *The behavior extraction equation of unit instructions is given by*

$$|\mathbf{u}(X); Y| = |X; Y|.$$

A PGAu program is in first canonical form if the program itself and the bodies of all units are in first canonical form when units are regarded as primitive instructions.

PGLB, PGLBg, PGLBu and PGLBur

In order to define the projection `pgau2pga` from PGAu to PGA, some variants of PGA are needed. They are the program notations PGLB, PGLBg and PGLBu. In particular, the program notation PGLB is a modification of PGA by adding *backward jump instructions* and omitting repetition. The program notation PGLBg is a variant of PGLB with *labels* and *gotos* instead of forward and backward jumps. The program notation PGLBu is the extension of PGLB with the unit instruction operator. For our projections, we will extend PGAu with backward jump instructions to the program notation PGLBur (r stands for repetition). We recall the notion of backward jump instructions from [22].

PGLBu	$\xrightarrow{\texttt{pglbu2pglbg}}$	PGLBg	$\xrightarrow{\texttt{pglbg2pglb}}$	PGLB
$\uparrow\texttt{pgau2pglbu}$				$\downarrow\texttt{pgau2pglbu}$
PGAu				PGA

Table 6.2: Projection from PGAu to PGA.

Backward jump instruction For any natural number k, there is an instruction $\backslash\#k$ which denotes a backward jump of length k. If $k = 0$, this jump is to itself and inaction occurs. If $k > 0$, it moves execution to proceed at k instructions backwards. If there are not that many instructions in the preceding part of the program, inaction occurs.

Projecting PGAu into PGA

The projection `pgau2pga` from PGAu into PGA is defined as a composition of four mappings `pgau2pglbu`, `pglbu2pglbg`, `pglbg2pglb` and `pglb2pga` in Table 6.2. These mappings are described in detail in [82].

6.3 Behavior extraction equations based on positions

The behavior extraction equations given in Definition 6.9 enable single pass execution for programs in PGA. However, they cannot be applied for determining program behaviors in the case that the programs contain a backward jump instruction. In this section, we define behavior extraction equations based on the positions of instructions in a program containing backward jumps, in order to determine its behavior in a natural way.

6.3.1 Positions of instructions as sequences of natural numbers

To define behavior extraction equations based on positions of instructions in a program, we locate all instructions in a PGAu program by sequences of natural numbers. The empty sequence is written as ϵ, and the concatenation of a sequence σ with a natural number n is σ, n. This location serves to keep track of the relative position in a unit and in all encompassing units. We define a partial order $<$ between sequences of natural numbers as follows.

Definition 6.14

1. *$\epsilon < \gamma$ for every sequence γ.*
2. *For two sequences $\delta = i, \delta'$ and $\sigma = j, \sigma'$ with $i, j \in \mathbb{N}$ and δ', σ' some sequences, $\delta < \sigma$ if $i < j$ or $i = j$ and $\delta' < \sigma'$.*

Definition 6.15 *The* **disunification operator** Δ *is defined on unit instructions* $\mathbf{u}(Y)$ *by* $\Delta\mathbf{u}(Y) = Y$.

We use the notation $[X]_\sigma$ to denote the instruction at position σ of a program X.

Definition 6.16 *Let* X *be a program in* PGAu *in first canonical form. The notation* $[X]_\sigma$ *is defined inductively as follows.*

1. $[X]_\epsilon = \mathbf{u}(X)$.
2. *If* $\Delta[X]_\sigma = v_1; \ldots; v_n$ *then for all* $1 \leq i \leq n : [X]_{\sigma,i} = v_i$.
3. *If* $\Delta[X]_\sigma = v_1; \ldots; v_n; (v_{n+1}; \ldots; v_{n+m})^\omega$ *then for all* $1 \leq i \leq n+m : [X]_{\sigma,i} = v_i$ *and for all* $i > n+m : [X]_{\sigma,i} = v_{n+1+((i-n-1) \mod m)}$.

If $[X]_\sigma$ is not defined then we write $\nexists[X]_\sigma$. If $\sigma \neq \epsilon$ and $[X]_\sigma$ is defined then we say that σ is a position of program X.

Example 6.17 Let $X = +a; \mathbf{u}(+b; \#2; \backslash\#3); !$. Then the instructions of X are located as follows:

$[X]_1 = +a$,
$[X]_2 = \mathbf{u}(+b; \#2; \backslash\#3)$,
$[X]_{2,1} = +b$, $[X]_{2,2} = \#2, [X]_{2,3} = \backslash\#3$,
$[X]_3 = !$.

We now define *addition* and *subtraction* of a sequence with a natural number in order to compute the position reached from a position by jumping some steps forward or backward.

Definition 6.18 *(***Addition***.) The function* $\sigma \oplus_X l$ *computes the position of the instruction or unit in a program* X *that one reaches by jumping* l *steps forward from position* σ. *Suppose* $\sigma = \delta, i$. *If* $\delta = \epsilon$ *then* $Y = X$ *otherwise let* $Y = \Delta[X]_\delta$. *We consider the following possibilities:*

1. $Y = u_1; \ldots; u_n$. *If* $\delta = \epsilon$ *or* $i + l \leq n$ *then* $\sigma \oplus_X l = \delta, i + l$ *otherwise* $\sigma \oplus_X l = \delta \oplus_X (l - n + i)$;
2. $Y = u_1; \ldots; u_n; (u_{n+1}; \ldots; u_{n+k})^\omega$. *Then* $\sigma \oplus_X l = \delta, i + l$.

Definition 6.19 *(***Subtraction***.) The function* $\sigma \ominus_X l$ *computes the position of the instruction or unit in a program* X *that one reaches by jumping* l *instructions backward from position* σ *of* X. *Suppose* $\sigma = \delta, i$. *We consider the following possibilities:*

1. $\delta = \epsilon$. *Then* $\sigma \ominus_X l = \max(0, i - l)$.
2. $\delta \neq \epsilon$. *If* $i > l$ *then* $\sigma \ominus_X l = \delta, i - l$ *otherwise* $\sigma \ominus_X l = \delta \ominus_X (l - i + 1)$.

We can simply use the notations $\oplus$ and $\ominus$ instead of $\oplus_X$ and $\ominus_X$ if X is fixed.

Example 6.20 Consider program X in Example 6.17. By Definition 6.18 and Definition 6.19, $(2, 2) \oplus 2 = 3$ and $(2, 3) \ominus 3 = 1$.

6.3.2 Behavior extraction equations on positions for PGAu

The behavior extraction equations based on positions of programs in PGAu are given as follows.

Definition 6.21 *Let X be a* PGAu *program in the first canonical form. Then $|\sigma, X|$ denotes the behavior of X at position σ and is defined by:*

$$|\sigma, X| = \begin{cases} S & \textit{if} \quad [X]_\sigma =!, \\ D & \textit{if} \quad [X]_\sigma = \#0 \text{ or } \nexists [X]_\sigma, \\ a \circ |\sigma \oplus 1, X| & \text{if} \quad [X]_\sigma = a, \\ |\sigma \oplus 1, X| \trianglelefteq a \trianglerighteq |\sigma \oplus 2, X| & \textit{if} \quad [X]_\sigma = +a, \\ |\sigma \oplus 2, X| \trianglelefteq a \trianglerighteq |\sigma \oplus 1, X| & \textit{if} \quad [X]_\sigma = -a, \\ |\sigma \oplus l, X| & \textit{if} \quad [X]_\sigma = \#l, \\ |(\sigma, 1), X| & \textit{if} \quad [X]_\sigma = \mathbf{u}(Y) \end{cases}$$

If $|\sigma, X|$ cannot be computed then inaction occurs, i.e., $|\sigma, X| = D$.

One can show that the behavior at the first position of a program in PGAu coincides with its behavior computed by behavior extraction equations in Definition 6.9.

Lemma 6.22 *Let X be a program in* PGAu. *Then $|1, X| = |X|$.*

Proof: Omitted. □

6.3.3 Behavior extraction equations based on positions for PGLBur

In this section, we provide behavior extraction equations based on positions of instructions for programs containing backward jumps in PGLBur.

We note that the notion of first canonical form for programs in PGLBur is defined as in Section 6.2.4 (backward jump instructions are regarded as primitive instructions).

Definition 6.23 *Let X be a* PGLBur *program in the first canonical form. The behavior extraction equations at position σ for backward jumps of X are defined by*

$$|\sigma, X| = |\sigma \ominus l, X| \textit{ if } [X]_\sigma = \backslash\#l.$$

The behavior extraction equations at position σ for the other instructions in PGLBur *are given as in Definition 6.21. The behavior $|X|$ of X is its behavior at the first position, i.e., $|X| = |1, X|$.*

Example 6.24 The behavior $|X|$ of program $X = +a; \mathbf{u}(+b; \#2; \backslash\#3); !$ in Example 6.17 is determined by

$$\begin{aligned}
|X| = |1, X| &= |1 \oplus 1, X| \trianglelefteq a \trianglerighteq |1 \oplus 2, X| \\
&= |2, X| \trianglelefteq a \trianglerighteq |3, X| \\
&= |(2,1), X| \trianglelefteq a \trianglerighteq S \\
&= (|(2,1) \oplus 1, X| \trianglelefteq b \trianglerighteq |(2,1) \oplus 2, X|) \trianglelefteq a \trianglerighteq S \\
&= (|(2,2), X| \trianglelefteq b \trianglerighteq |(2,3), X|) \trianglelefteq a \trianglerighteq S \\
&= (|(2,2) \oplus 2, X| \trianglelefteq b \trianglerighteq |(2,3) \ominus 3, X|) \trianglelefteq a \trianglerighteq S \\
&= (|3, X| \trianglelefteq b \trianglerighteq |1, X|) \trianglelefteq a \trianglerighteq S \\
&= (S \trianglelefteq b \trianglerighteq |X|) \trianglelefteq a \trianglerighteq S.
\end{aligned}$$

6.4 PGA with conditional instructions

It is observed in [22] that PGA extended with *conditional instructions* is a natural extension for PGA. The conditional instruction is introduced in [22] with two semantics: a full projection semantics and a lazy projection semantics. The latter semantics is less flexible than the former, however, it is much simpler. The extension of PGA with conditional instructions respecting the full projection semantics, denoted by PGLEc, has been studied in [22]. In this section, we investigate the extension PGAuc of PGAu with conditional instructions, with respect to the lazy projection semantics, by providing a projection from PGAuc to PGAu.

6.4.1 The conditional instruction

We recall the notion of conditional instructions from [22]. There are two types of conditional instructions:

Positive conditional For each basic instruction $a \in \Sigma$, there is a positive conditional choice for PGA, denoted by $\mathtt{if} + a$. Its execution starts with a, if true is returned the next instruction is performed and the subsequent instruction is skipped. If false is returned the next instruction is skipped and execution continues thereafter.

Negative conditional For each basic instruction $a \in \Sigma$, there is a negative conditional choice for PGA, denoted by $\mathtt{if} - a$. Its execution starts with a, if false is returned the next instruction is performed and the subsequent instruction is skipped. If true is returned the next instruction is skipped and execution continues thereafter.

The definitions of finite programs and first canonical form in PGAuc are given as usual (see Definition 6.2, Definition 6.3 and Section 6.2.4) where units and conditional instructions are regarded as primitive instructions.

The behavior of a PGAuc program is determined by the behavior extraction equations in Definition 6.8, Definition 6.9, Definition 6.13, and the behavior extraction equations for conditional instructions defined as follows.

Definition 6.25 *The behavior extraction equations for conditional instructions are*

$$|\mathtt{if}+a;u;X| = |-a;\#3;u;\#2;X| = |u;\#2;X| \trianglelefteq a \trianglerighteq |X|,$$
$$|\mathtt{if}-a;u;X| = |+a;\#3;u;\#2;X| = |X| \trianglelefteq a \trianglerighteq |u;\#2;X|.$$

Example 6.26

$$|\mathtt{if}+a;b;c;!| = |b;\#2;c;!| \trianglelefteq a \trianglerighteq |c;!| = (b \circ S) \trianglelefteq a \trianglerighteq (c \circ S)$$

It should be noticed that the behavior extraction equations for conditional instructions sometimes give undesired behaviors for programs as can be seen in the following example.

Example 6.27

1. $|\mathtt{if}+a;\#2;b;!| = |\#2;\#2;b;!| \trianglelefteq a \trianglerighteq |b;!| = |b;!| \trianglelefteq a \trianglerighteq |b;!| = a \circ b \circ S$,
2. $|\mathtt{if}+a;+b;c;!| = |+b;\#2;c;!| \trianglelefteq a \trianglerighteq |c;!| = (S \trianglelefteq b \trianglerighteq (c \circ S)) \trianglelefteq a \trianglerighteq (c \circ S)$,
3. $|\mathtt{if}+a;\mathtt{if}+b;c;!| = |\mathtt{if}+b;\#2;c;!| \trianglelefteq a \trianglerighteq |c;!| = (b \circ c \circ S) \trianglelefteq a \trianglerighteq (c \circ S)$.

6.4.2 Projecting PGAuc into PGAu

To define the projection `pgauc2pgau`, we will use the 2-*update* $\mathtt{Upd}_2(X)$ that updates the contents of jumps with two more steps in some cases. This operation uses $\mathtt{least\_jumps}(\sigma, X)$ to count the least jump from position σ that takes one outside X.

Definition 6.28 *Let X be a program in first canonical form. The function* $\mathtt{least\_jumps}(\sigma, X)$ *calculates the* **least jump** *that takes one outside X from position σ of X. Suppose $\sigma = \delta, i$. We consider the following possibilities:*

1. *$\delta = \epsilon$. There are two cases:*
   (a) *$X = u_1; \ldots; u_n$. Then* $\mathtt{least\_jumps}(\sigma, X) = n - i + 1$.
   (b) *$X = Y; Z^\omega$. Then* $\mathtt{least\_jumps}(\sigma, X) = \infty$.
2. *$\delta \neq \epsilon$. There are also two cases:*
   (a) *$\Delta[X]_\delta = u_1; \ldots; u_n$. Then* $\mathtt{least\_jumps}(\sigma, X) = \mathtt{least\_jumps}(\delta, X) + n - i$.
   (b) *$\Delta[X]_\delta = Y; Z^\omega$. Then* $\mathtt{least\_jumps}(\sigma, X) = \infty$.

Example 6.29 Let $X = \mathbf{u}(a;\#3;\#3);b$ be a program in PGAu. Then $\mathtt{least\_jumps}((1,2), X) = 3$ and $\mathtt{least\_jumps}((1,2), X) = 2$.

Definition 6.30 *Let X be a program expression in* PGAu. *Then the* 2-**update** $\mathtt{Upd}_2(X)$ *is the program expression Y which is the same as X except that when $[X]_\sigma = \#l$ with $l > \mathtt{least\_jumps}(\sigma, X)$, $[Y]_\sigma = \#l + 2$.*

Example 6.31 Let X be the program in Example 6.29. Then $\mathtt{Upd}_2(X) = \mathbf{u}(a; \#2; \#4); b$.

Definition 6.32 *The projection* pgauc2pgau *from* PGAuc *to* PGAu *is the operation on program expressions in first canonical form that replaces a conditional instruction* $\mathtt{if} \pm a$ *at position σ of X by the unit*

$$\mathbf{u}(\pm a; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgauc2pgau}([X]_{\sigma \oplus_X 1}; \#2))); \#2)$$

leaving the rest unchanged.

Let Y be the program after projecting program X from PGAuc to PGAu. The function $\mathtt{Upd}_2()$ in the previous definition ensures that the behavior at position $(\sigma, 2, 1)$ of instruction $[Y]_{\sigma,2,1}$ (the replacement of $[X]_{\sigma \oplus_X 1}$ in $[Y]_\sigma$) is the same as the behavior at position $\sigma \oplus 1$ of instruction $[Y]_{\sigma \oplus_Y 1}$ (the replacement of $[X]_{\sigma \oplus_X 1}$ in Y). If there is a forward jump l in $[Y]_{\sigma \oplus_Y 1}$ that takes one outside $[Y]_{\sigma \oplus_Y 1}$ then this jump is updated to $l + 2$ in $[Y]_{\sigma,2,1}$ in order to reach the same instruction, since there are two instructions (#2) between $[Y]_{\sigma,2,1}$ and $[Y]_{\sigma \oplus_Y 1}$.

Since $[X]_{\sigma \oplus_X 1}; \#2$ contains fewer conditional instructions than X, the projection $\mathtt{pgauc2pgau}(X)$ is well-defined. We illustrate this projection by the following example.

Example 6.33 Consider the program given in Example 6.26. Its projection into PGAu is given as follows.

$$\begin{aligned} \mathtt{pgauc2pgau}(\mathtt{if} + a; b; c; !) &= \mathbf{u}(+a; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgauc2pgau}(b; \#2))); \#2); b; c; ! \\ &= \mathbf{u}(+a; \mathbf{u}(b; \#4); \#2); b; c; ! \end{aligned}$$

In Table 6.3, a more complicated example taken from Example 6.27 for the projection pgauc2pgau is given. In this example, the occurrence of consecutive conditional statements produces an undesired behavior for the program.

In the following, we prove correctness for the projection pgauc2pgau. Let Ψ abbreviate this projection. We provide some supporting results.

Lemma 6.34 *Let X, Y be programs in* PGAuc. *Then*

$$\begin{aligned} |\Psi(!; X)| &= S, \\ |\Psi(a; X)| &= a \circ |\Psi(X)|, \\ |\Psi(+a; u; X)| &= |\Psi(u; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|, \\ |\Psi(-a; u; X)| &= |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; X)|, \\ |\Psi(\#0; X)| &= D, \\ |\Psi(\#1; X)| &= |\Psi(X)|, \\ |\Psi(\#k + 2; u; X)| &= |\Psi(\#k + 1; X)|, \\ |\Psi(\mathbf{u}(X); Y)| &= |\Psi(X; Y)|. \end{aligned}$$

$\mathtt{pgauc2pgau}(\mathtt{if}+a;\mathtt{if}+b;c;!)$

$= \quad \mathbf{u}(+a;\mathbf{u}(\mathtt{Upd}_2(\mathtt{pgauc2pgau}(\mathtt{if}+b;\#2)));\#2);$
$\mathbf{u}(+b;\mathbf{u}(\mathtt{Upd}_2(\mathtt{pgauc2pgau}(c;\#2)));\#2);c;!$

$= \quad \mathbf{u}(+a;\mathbf{u}(\mathtt{Upd}_2(\mathbf{u}(+b;\mathbf{u}(\mathtt{Upd}_2(\mathtt{pgauc2pgau}(\#2;\#2)));\#2);\#2));\#2);$
$\mathbf{u}(+b;\mathbf{u}(c;\#4);\#2);c;!$

$= \quad \mathbf{u}(+a;\mathbf{u}(\mathtt{Upd}_2(\mathbf{u}(+b;\mathbf{u}(\#2;\#4);\#2);\#2));\#2);$
$\mathbf{u}(+b;\mathbf{u}(c;\#4);\#2);c;!$

$= \quad \mathbf{u}(+a;\mathbf{u}(\mathbf{u}(+b;\mathbf{u}(\#2;\#6);\#2);\#4);\#2);\ \mathbf{u}(+b;\mathbf{u}(c;\#4);\#2);c;!$

Table 6.3: An example on pgauc2pgau.

Proof: This follows from Definition 6.9, Definition 6.13 and Definition 6.32. □

Lemma 6.35 *Let X be a program in* PGAuc*, and u a unit. Then*

$$|\#k+1;\Psi(u;X)| = |\#k;\Psi(X)|.$$

Proof: This follows from the fact that $\Psi(u;X) = v;\Psi(X)$ for some primitive instruction or unit v. □

Lemma 6.36 *Let X be a program in* PGAuc*. Then*

$$\begin{aligned} |\Psi(\mathtt{if}+a;u;X)| &= |\Psi(u;\#2;X)| \trianglelefteq a \trianglerighteq |\Psi(X)|, \\ |\Psi(\mathtt{if}-a;u;X)| &= |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u;\#2;X)|. \end{aligned}$$

Proof: It follows from Definition 6.32 and Lemma 6.35 that

$$\begin{aligned} |\Psi(\mathtt{if}+a;u;X)| &= |\mathbf{u}(+a;\mathbf{u}(\mathtt{Upd}_2(\Psi(u;\#2)));\#2);\Psi(u;X)| \\ &= |+a;\mathbf{u}(\mathtt{Upd}_2(\Psi(u;\#2)));\#2;\Psi(u;X)| \\ &= |\mathbf{u}(\mathtt{Upd}_2(\Psi(u;\#2)));\#2;\Psi(u;X)| \trianglelefteq a \trianglerighteq |\#2;\Psi(u;X)| \\ &= |\mathtt{Upd}_2(\Psi(u;\#2));\#2;\Psi(u;X)| \trianglelefteq a \trianglerighteq |\Psi(X)| \end{aligned}$$

Let $Y = \mathtt{Upd}_2(\Psi(u;\#2));\#2;\Psi(u;X)$ and $Z = \Psi(u;\#2;X)$. We define a binary relation $\sim$ between positions of Y and Z as follows.

1. $1,\sigma \sim 1,\sigma$ for every sequence σ. Here $[Y]_{1,\sigma} = [Z]_{1,\sigma}$ if $[Z]_{1,\sigma} \in \{a, \pm a, !\}$. Moreover, if $[Z]_{1,\sigma} = \#l$ then $[Y]_{1,\sigma} = \#l$ if $l \leq \mathtt{least\_jumps}((1,\sigma), \Psi(u;\#2))$ otherwise $[Y]_{1,\sigma} = \#l + 2$.

2. $2 \sim 2$. Here $[Y]_2 = \#4$ (by Definition 6.30) and $[Z]_2 = \#2$.

3. $3 \sim 3$. Here $[Y]_3 = \#2$ and $[Z]_3 = [\Psi(X)]_1$.

4. Finally, $(5 \oplus_Y l, \sigma) \sim (3 \oplus_Z l, \sigma)$ for all $l \geq 0$ and for all sequences σ. Here $[Y]_{5\oplus_Y l} = [Z]_{3\oplus_Z l} = [\Psi(X)]_{1\oplus_{\Psi(X)} l}$.

Let $P_\sigma = |\sigma, Y|$ and $Q_\delta = |\delta, Z|$ for positions σ of Y and positions δ of Z. We show that for all positions σ of Y and δ of Z with $\sigma \sim \delta$,

$$\begin{array}{l} \text{if } P_\sigma = S \text{ then } Q_\delta = S;\\ \text{if } P_\sigma = D \text{ then } Q_\delta = D;\\ \text{if } P_\sigma = P_{\sigma'} \text{ then } Q_\delta = Q_{\delta'} \text{ with } \sigma' \sim \delta';\\ \text{if } P_\sigma = P_{\sigma'} \trianglelefteq a \trianglerighteq P_{\sigma''} \text{ then } Q_\delta = Q_{\sigma'} \trianglelefteq a \trianglerighteq Q''_\sigma \text{ with } \sigma' \sim \delta' \text{ and } \sigma'' \sim \delta'' \end{array} \tag{6.1}$$

and vice versa.

Let α and β be the positions of Y (if they exist) such that $\alpha \oplus_Y 2 = \beta \oplus_Y 1 = 2$. We consider the following possibilities:

1. $\sigma = \delta = \alpha$. There are eight cases:

   (a) $[Y]_\sigma = [Z]_\sigma = !$. By Definition 6.21, $P_\sigma = Q_\sigma = S$.

   (b) $[Y]_\sigma = [Z]_\sigma = a$ for some $a \in \Sigma$. Then $P_\sigma = a \circ P_{\sigma\oplus_Y 1}$ and $Q_\sigma = a \circ Q_{\sigma\oplus_Z 1}$ with $\sigma \oplus_Y 1 \sim \sigma \oplus_Z 1$.

   (c) $[Y]_\sigma = [Z]_\sigma = +a$ for some $a \in \Sigma$. Then

   $$\begin{array}{rcl} P_\sigma & = & P_{\sigma\oplus_Y 1} \trianglelefteq a \trianglerighteq P_{\sigma\oplus_Y 2} = P_{\sigma\oplus_Y 1} \trianglelefteq a \trianglerighteq P_2 = P_{\sigma\oplus_Y 1} \trianglelefteq a \trianglerighteq P_{2\oplus_Y 4}\\ & = & P_{\sigma\oplus_Y 1} \trianglelefteq a \trianglerighteq P_5\\ Q_\sigma & = & Q_{\sigma\oplus_Z 1} \trianglelefteq a \trianglerighteq Q_{\sigma\oplus_Y 2} = Q_{\sigma\oplus_Z 1} \trianglelefteq a \trianglerighteq Q_2 = Q_{\sigma\oplus_Z 1} \trianglelefteq a \trianglerighteq Q_{2\oplus_Z 2}\\ & = & Q_{\sigma\oplus_Z 1} \trianglelefteq a \trianglerighteq Q_3 \end{array}$$

   with $\sigma \oplus_Y 1 \sim \sigma \oplus_Z 1$ and $5 \sim 3$.

   (d) $[Y]_\sigma = [Z]_\sigma = -a$ for some $a \in \Sigma$. Similar to the previous case, we likewise get $P_\sigma = P_5 \trianglelefteq a \trianglerighteq P_{\sigma\oplus_Y 1}$ and $Q = Q_3 \trianglelefteq a \trianglerighteq Q_{\sigma\oplus_Z 1}$ with $5 \sim 3$ and $\sigma \oplus_Y 1 \sim \sigma \oplus_Z 1$.

   (e) $[Y]_\sigma = [Z]_\sigma = \#0$. Then $P_\sigma = Q_\sigma = D$.

   (f) $[Y]_\sigma = [Z]_\sigma = \#l$ for $1 \leq l \leq 3$. Then $P_\sigma = P_{\sigma\oplus_Y l}$ and $Q_\sigma = Q_{\sigma\oplus_Z l}$ with $\sigma \oplus_Y l \sim \sigma \oplus_Z l$.

   (g) $[Y]_\sigma = l+2$ and $[Z]_\sigma = l$ for some $l > 3$. Then $P_\sigma = P_{\sigma\oplus_Y l+2} = P_{5\oplus_Y (l-3)}$ and $Q_\sigma = Q_{\sigma\oplus_Z l} = Q_{3\oplus_Z (l-3)}$ with $5 \oplus_Y (l-3) \sim 3 \oplus_Z (l-3)$.

   Hence, (6.1) also holds for this case.

2. $\sigma = \delta = \beta$. The proof of (6.1) is similar to the previous case.
3. $\sigma = \delta = 2$. Here $[Y]_2 = \#4$ and $[Z]_2 = \#2$. Therefore $P_2 = P_{2\oplus_Y 4} = P_6$ and $Q_2 = Q_{2\oplus_Z 2} = Q_4$ with $6 \sim 4$.
4. $\sigma = \delta = 3$. Here $[Y]_3 = \#2$. Therefore $P_3 = P_{3\oplus_Y 2} = P_5$ with $5 \sim 3$.
5. $\sigma < \alpha$ or $\sigma \geq 5$. (6.1) can be derived similarly to the first case.

It follows from Theorem 6.7 that $P_\sigma = Q_\delta$ for all positions σ of Y and δ of Z with $\sigma \sim \delta$. Therefore $|Y| = |Z|$. Hence,

$$|\Psi(X)| = |\Psi(u; \#2; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|.$$

We likewise get $|\Psi(\mathtt{if} - a; u; X)| = |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; \#2; X)|$. □

Theorem 6.37 *The projection* pgauc2pgau *from* PGAuc *to* PGAu *is correct, i.e, for every program X in* PGAuc*:*

$$|\mathtt{pgauc2pgau}(X)| = |X|.$$

Proof: This follows from Definition 6.9, Definition 6.25, Lemma 6.34, Lemma 6.36 and Theorem 6.7. □

Our projection pgauc2pgau from PGAucw to PGAu together with the projection pgau2pga from PGAu to PGA constitutes a projection from PGAuc to PGA. This projection is useful for the study of PGA itself. Furthermore, its correctness shows that the conditional instruction is not needed as a primitive instruction in terms of expressiveness.

6.5 PGA with while-loops

A while-loop is a control structure used in most programming languages that allows code to be executed repeatedly based on a given condition. Similar to the condition instruction, the while-loop instruction is also introduced in [22] with a full projection semantics and a lazy projection semantics. The extension of PGA with conditional and while-loop instructions respecting the full projection semantics has been given as the program notation PGLEcw in [22]. This section studies the extension PGAucw of PGA with units, conditional instructions and while-loops, with respect to the lazy projection semantics. We discuss non-regularity of PGA with while-loops. Under a restriction that no consecutive occurrences of while-loops are permitted, we present a projection pgaucw2pga from PGAucw into PGA.

6.5.1 The while-loop instruction

We recall from [22] the notion of while-loop instructions. There are two types of while-loop instructions:

Positive while header For each $a \in \Sigma$, there is a positive while header for PGA, denoted by $\mathtt{while} + a$. Its execution starts with a. If true is returned, the next instruction is executed and then we return to $\mathtt{while} + a$. If false is returned, the next instruction is skipped and execution continues thereafter.

Negative while header For each $a \in \Sigma$, there is a negative while header for PGA, denoted by $\mathtt{while} - a$. Its execution starts with a. If false is returned, the next instruction is executed and then we return to $\mathtt{while} - a$. If true is returned, the next instruction is skipped and execution continues thereafter.

Definition 6.2 for finite programs in PGA is extended straightforwardly to PGAucw where units, conditional instructions and while-loops are regarded as primitive instructions.

Definition 6.38 *The behavior extraction equations for while-loop instructions are*

$$\begin{array}{lcl} |\mathtt{while} + a; u; X| & = & |-a; \#4; u; \mathtt{while} + a; u; X| \\ & = & |u; \mathtt{while} + a; u; X| \trianglelefteq a \trianglerighteq |X|, \\ |\mathtt{while} - a; u; X| & = & |+a; \#4; u; \mathtt{while} - a; u; X| \\ & = & |X| \trianglelefteq a \trianglerighteq |u; \mathtt{while} - a; u; X|. \end{array}$$

It should be noticed that PGA programs with while-loop instructions can yield curious behaviors, as seen in the following example:

Example 6.39

1. $|\mathtt{while} + a; \#2; b| = |\#2; \mathtt{while} + a; \#2; b| \trianglelefteq a \trianglerighteq |b| = D \trianglelefteq a \trianglerighteq b \circ D$.
2. $|\mathtt{if} + c; \mathtt{while} + a; b| = |\mathtt{while} + a; \#2; b| \trianglelefteq c \trianglerighteq |b| = (D \trianglelefteq a \trianglerighteq b \circ D) \trianglelefteq c \trianglerighteq (b \circ D)$
3. $|\mathtt{while} + a; \mathtt{if} + c; b; !| = |\mathtt{if} + c; \mathtt{while} + a; \mathtt{if} + c; b; !| \trianglelefteq a \trianglerighteq |b; !|$
   $= (|\mathtt{while} + a; \#2; \mathtt{if} + c; b; !| \trianglelefteq c \trianglerighteq |\mathtt{if} + c; b; !|) \trianglelefteq a \trianglerighteq (b \circ S)$
   $= (((b \circ S) \trianglelefteq a \trianglerighteq ((b \circ D) \trianglelefteq c \trianglerighteq S)) \trianglelefteq c \trianglerighteq ((b \circ D) \trianglelefteq c \trianglerighteq (S))) \trianglelefteq a \trianglerighteq (b \circ S)$

6.5.2 Non-regularity of programs containing while-loops

This section shows that programs containing while-loops may yield non-regular behaviors.

Proposition 6.40 *There exists a program in* PGAucw *that produces a non-regular behavior.*

Proof: Let $X = \mathtt{while} + a; \mathtt{while} + b; c$ be a program in PGAucw. We will show that the behavior $|X|$ of X is not regular. Let $E_1 = |X|$, $E_0 = |\mathtt{while} + b; c| = c \circ E_0 \trianglelefteq b \trianglerighteq D$

and $E_{-1} = |c| = c \circ D$. By Definition 6.38,

$$\begin{aligned}
E_1 &= E_2 \trianglelefteq a \trianglerighteq E_{-1}\\
E_2 &= E_3 \trianglelefteq b \trianglerighteq E_0\\
E_3 &= E_4 \trianglelefteq a \trianglerighteq E_1\\
&\vdots\\
E_{2k+1} &= E_{2k+2} \trianglelefteq a \trianglerighteq E_{2k-1}\\
E_{2k+2} &= E_{2k+3} \trianglelefteq b \trianglerighteq E_{2k}\\
&\vdots
\end{aligned}$$

where for all $k > 0$,

$$\begin{aligned}
E_{2k-1} &= |(\mathtt{while} + a; \mathtt{while} + b)^k; c|,\\
E_{2k} &= |\mathtt{while} + b; (\mathtt{while} + a; \mathtt{while} + b)^k; c|.
\end{aligned}$$

Suppose that P is regular. Then there is a least $n \in \mathbb{N}$ such that $\exists i < n$ with $E_n = E_i$. But, by the equations above, this implies that $E_{n+1} = E_{i+1}$ and $E_{n-2} = E_{i-2}$: a contradiction. So, P is not regular. □

The undesired result above implies that the program notation PGAucw containing while-loops cannot be projected into PGA (see Theorem 6.10). To circumvent this problem, we impose a restriction on PGAucw in order to avoid non-regular behaviors as follows: No consecutive occurrence of while-loops is permitted, that is to say a while-loop instruction $\mathtt{while} \pm a$ must not be followed by another while-loop instruction $\mathtt{while} \pm b$ or a unit u with $\mathtt{li}(u) = \mathtt{while} \pm b$, where the function $\mathtt{li}(u)$ denotes the last primitive instruction performed by u given as follows.

Definition 6.41 *The* **last primitive instruction** $\mathtt{li}(X)$ *of a program X in the first canonical form is defined by*

1. $\mathtt{li}(X) = X$ *if X consists of a single primitive instruction;*
2. $\mathtt{li}(\mathbf{u}(X)) = \mathtt{li}(X)$*;*
3. *if* $X = u_1; \ldots; u_n$ *then* $\mathtt{li}(X) = \mathtt{li}(u_n)$*;*
4. *if* $X = Y; Z^\omega$ *then* $\mathtt{li}(X) = \#0$.

Example 6.42

1. The program $X = \mathtt{while} + a; \mathbf{u}(c; \mathtt{while} + b); d$ produces a non-regular behavior although the two while-loop instructions of X are not placed next to each other. It is because after the application of the lazy projection semantics for the instruction $\mathtt{while} + a$, the code fragment $\mathtt{while} + b; \mathtt{while} + a$ will occur and produce a non-regular behavior for the program as seen in Proposition 6.40.
2. The program $Y = \mathtt{while} + a; \mathbf{u}(\mathtt{while} + b; c); !$ produces a regular behavior although Y seems to contain a consecutive occurrence of while-loops. After the application of the lazy projection semantics for the instruction $\mathtt{while} + a$, the instructions $\mathtt{while} + b$ and $\mathtt{while} + a$ is separated by the instruction c.

In the next section we define the projection pgaucw2pga from PGAucw to PGA under the restriction above and show its correctness.

6.5.3 Projecting PGAucw into PGA

Replacing while-loop instructions requires an instruction that can move back the control-flow such as backward jumps. Thus, instead of defining a projection from PGAucw to PGAu as considered in Section 6.4, we define a projection from PGAucw to PGLBu, a variant of PGA with units and backward jumps (see Section 6.2.4). We first describe a transformation pgaucw2pgbur from PGAucw to PGLBur under the restriction that no consecutive occurrences of while-loops are permitted, where PGLBur is an extension of PGLBu with repetition. We then define a transformation pglbur2pglbu from PGLBur to PGLBu. The program algebra projection pgaucw2pga from PGAucw to PGA is a composition of three mappings pgaucw2pgbur, pglbur2pglbu and pglbu2pga. We note that the projection pglbu2pga from PGLBu to PGA is covered in [82].

Transformation from PGAucw to PGLBur

The transformation pgaucw2pgbur from PGAucw to PGLBur removes all conditional instructions and while-loops of the programs with the use of backward jumps.

Definition 6.43 *The transformation* pgaucw2pgbur *from* PGAucw *to* PGLBur *is the operation on program expressions in first canonical form that replaces a conditional instruction* $\mathtt{if} \pm a$ *at position* σ *of* X *by*

$$\mathbf{u}(\pm a; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgaucw2pgbur}([X]_{\sigma \oplus_X 1}; \#2))); \#2)$$

and a while-loop $\mathtt{while} \pm a$ *at position* δ *by*

$$\mathbf{u}(\pm a; \mathbf{u}(\mathtt{Upd}_2(\chi_\delta([X]_{\delta \oplus_X 1})); \backslash\#2; \#2); \#2)$$

leaving the rest unchanged. The operation χ_δ *is the same as* pgaucw2pgbur *except that if the last primitive instruction of* $[X]_{\delta \oplus_X 1}$ *is a conditional instruction* $\mathtt{if} \pm c$ *then it is replaced by*

$$\begin{aligned} &\mathbf{u}(\pm c; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgaucw2pgbur}([X]_\delta; \#2))); \#2) \\ &= \mathbf{u}(\pm c; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgaucw2pgbur}(\mathtt{while} \pm a; \#2)); \#2)) \\ &= \mathbf{u}(\pm c; \mathbf{u}(\mathbf{u}(\pm a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#4); \#2) \end{aligned} \tag{6.2}$$

This transformation is defined under the restriction that $\mathtt{li}([X]_{\delta \oplus_X 1}) \neq \mathtt{while} \pm b$.

The meaning of the expressions χ_σ depends, of course, on the program X that we are considering. If it is not obvious by itself which program this is, we write χ_σ^X to emphasize that the current projecting objects are in program X.

The projection above is illustrated by three examples in Table 6.4, Table 6.5 and Table 6.6. The programs in Table 6.5 and Table 6.6 are taken from Example 6.39 in

$$\begin{aligned}
\texttt{pgaucw2pgbur}(X) &= \texttt{pgaucw2pgbur}(\texttt{while} + a; b)\\
&= \mathbf{u}(+a; \mathbf{u}(\texttt{Upd}_2(\chi_1^X(b)); \backslash\#2; \#2); \#2); b\\
&= \mathbf{u}(+a; \mathbf{u}(b; \backslash\#2; \#2); \#2); b
\end{aligned}$$

Table 6.4: First example on pgaucw2pgbur

$$\begin{aligned}
&\texttt{pgaucw2pgbur}(X) = \texttt{pgaucw2pgbur}(\texttt{if} + c; \texttt{while} + a; b)\\
&= \mathbf{u}(+c; \mathbf{u}(\texttt{Upd}_2(\texttt{pgaucw2pgbur}(\texttt{while} + a; \#2))); \#2);\\
&\quad \mathbf{u}(+a; \mathbf{u}(\texttt{Upd}_2(\chi_2^X(b)); \backslash\#2; \#2); \#2); b\\
&= \mathbf{u}(+c; \mathbf{u}(\texttt{Upd}_2(\mathbf{u}(+a; \mathbf{u}(\texttt{Upd}_2(\chi_1^{\texttt{while}+a;\#2}(\#2)); \backslash\#2; \#2); \#2); \#2)); \#2);\\
&\quad \mathbf{u}(+a; \mathbf{u}(b; \backslash\#2; \#2); \#2); b\\
&= \mathbf{u}(+c; \mathbf{u}(\texttt{Upd}_2(\mathbf{u}(+a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#2)); \#2);\\
&\quad \mathbf{u}(+a; \mathbf{u}(b; \backslash\#2; \#2); \#2); b\\
&= \mathbf{u}(+c; \mathbf{u}(\mathbf{u}(+a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#4); \#2);\\
&\quad \mathbf{u}(+a; \mathbf{u}(b; \backslash\#2; \#2); \#2); b
\end{aligned}$$

Table 6.5: Second example on pgaucw2pgbur

which a conditional instruction is followed by a while-loop and a while-loop is followed by a conditional instruction. Both programs produce undesired behaviors after the transformation.

Lemma 6.44 *The transformation* pgaucw2pgbur *is well-defined.*

Proof: It is because both $[X]_{\sigma\oplus_X 1}$ and $[X]_{\delta\oplus_X 1}$ in Definition 6.43 contain fewer conditional instructions and while-loops than X. □

Let Ψ abbreviate the projection pgaucw2pgbur. To prove correctness for Ψ, we use some supporting results.

Lemma 6.45 *Let X be a* PGLBur *program that does not contain any backward jump instruction that takes one outside X. Then for all $k \geq 0$,*

$$|1 \oplus_X k + 1, u; X| = |1 \oplus_X k, X|.$$

$\mathtt{pgaucw2pgbur}(X) = \mathtt{pgaucw2pgbur}(\mathtt{while} + a; \mathtt{if} + c; b; !)$

$= \mathbf{u}(+a; \mathbf{u}(\mathtt{Upd}_2(\chi_1^X(\mathtt{if} + c)); \backslash\#2; \#2); \#2);$
$\mathbf{u}(+c; \mathbf{u}(\mathtt{Upd}_2(\mathtt{pgaucw2pgbur}(b; \#2))); \#2); b; !$

$= \mathbf{u}(+a; \mathbf{u}(\mathtt{Upd}_2(\mathbf{u}(+c; \mathbf{u}(\mathbf{u}(\pm a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#4); \#2)); \backslash\#2; \#2); \#2);$
$\mathbf{u}(+c; \mathbf{u}(b; \#4); \#2); b; !$

$= \mathbf{u}(+a; \mathbf{u}(\mathbf{u}(+c; \mathbf{u}(\mathbf{u}(+a; \mathbf{u}(\#4, \backslash\#2; \#2); \#2); \#6); \#4); \backslash\#2; \#2); \#2);$
$\mathbf{u}(+c; \mathbf{u}(b; \#4); \#2); b; !$

Table 6.6: Third example on `pgaucw2pgbur`

Proof: Straightforward. □

Lemma 6.46 *Let X, Y be program in* PGAucw. *Then*

$$\begin{aligned}
|\Psi(!; X)| &= S,\\
|\Psi(a; X)| &= a \circ |\Psi(X)|,\\
|\Psi(+a; u; X)| &= |\Psi(u; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|,\\
|\Psi(-a; u; X)| &= |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; X)|,\\
|\Psi(\#0; X)| &= D,\\
|\Psi(\#1; X)| &= |\Psi(X)|,\\
|\Psi(\#k + 2; u; X)| &= |\Psi(\#k + 1; X)|,\\
|\Psi(\mathbf{u}(X); Y)| &= |\Psi(X; Y)|,\\
|\Psi(\mathtt{if} + a; u; X)| &= |\Psi(u; \#2; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|,\\
|\Psi(\mathtt{if} - a; u; X)| &= |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; \#2; X)|.
\end{aligned}$$

Proof: Similar to the proofs of Lemma 6.34 and Lemma 6.36. □

Lemma 6.47 *Let X be a* PGAucw *program. Then*

1. $|\Psi(\mathtt{while} + a; u; X)| = |\Psi(u; \mathtt{while} + a; u; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|$,
2. $|\Psi(\mathtt{while} - a; u; X)| = |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; \mathtt{while} - a; u; X)|$

for some instruction u with $\mathtt{li}(u) \neq \mathtt{while} \pm b$.

Proof: We prove Clause 1 as follows. Let $X' = \mathtt{while} + a; u; X$, $Y = \Psi(X')$ and $Z = \Psi(u, \mathtt{while} + a; u; X)$. Then Y and Z are PGLBur programs. It follows from Definition 6.43 that

$$Y = \mathbf{u}(+a; \mathbf{u}(\mathtt{Upd}_2(\chi_1^{X'}(u)); \backslash\#2; \#2); \#2); \Psi(u; X).$$

By Definition 6.21, Definition 6.23 and Lemma 6.45,

$$\begin{array}{rcl} |Y| & = & |(1,2), Y| \trianglelefteq a \trianglerighteq |(1,3), Y| \\ & = & |(1,2), Y| \trianglelefteq a \trianglerighteq |(1,3) \oplus_Y 2, Y| \\ & = & |(1,2), Y| \trianglelefteq a \trianglerighteq |\Psi(X)|. \end{array}$$

Let α and β be the positions in Y and Z of the replacements of $\mathtt{li}(u)$ occurring in $[Y]_{1,2,1}$ and $[Z]_1$, $\alpha \oplus_Y 1 = 1, 2, 2$ and $\beta \oplus_Z 1 = 2$. It follows from Definition 6.43 that if $\mathtt{li}(u) = \mathtt{if} \pm c$ then

$$\begin{array}{rcl} [Y]_\alpha & = & \mathbf{u}(\pm c; \mathbf{u}(\mathbf{u}(+a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#6); \#4), \\ [Z]_\beta & = & \mathbf{u}(\pm c; \mathbf{u}(\mathbf{u}(+a; \mathbf{u}(\#4; \backslash\#2; \#2); \#2); \#4); \#2) \end{array}$$

(see Table 6.5 and Table 6.6 for an illustration). It can be derived that that $[Y]_{1,2,1} = \mathtt{Upd}_2([Z]_1)$. We define a binary relation $\sim$ between positions of programs Y and Z as follows.

1. $1 \oplus_Y l, \sigma \sim 2 \oplus_Z l, \sigma$ for all $l \geq 0$ and for all sequences σ. Here $[Y]_{1 \oplus_Y l, \sigma} = [Z]_{2 \oplus_Z l, \sigma}$. Moreover, $\backslash\#2$ is the only backward jump that can occur in Y and Z. If $[Y]_{1 \oplus_Y l, \sigma} = \backslash\#2$ then by Definition 6.43, $\sigma = \delta, 1, 2, 2$ for some sequence δ and $(1 \oplus_Y l, \sigma) \ominus_Y 2 = 1 \oplus_Y l, \delta, 1, 1$. Similarly, if $[Z]_{2 \oplus_Z l, \sigma} = \backslash\#2$ then $\sigma = \delta, 1, 2, 2$ for some sequence δ and $(2 \oplus_Z l, \sigma) \ominus_Z 2 = 2 \oplus_Z l, \delta, 1, 1$;
2. $1, 2 \sim 1$.
3. $1, 2, 1, \sigma \sim 1, \sigma$ for every sequence σ. Here $[Y]_{1,2,1} = \mathtt{Upd}_2([Z]_1)$;
4. $1, 2, 2 \sim 2, 1$. Here $[Y]_{1,2,2} = \backslash\#2$, $[Y]_{1,2,2 \ominus_Y 2} = [Y]_{1,1}$ and $1, 1 \sim 2, 1$;
5. $1, 2, 3 \sim 3$. Here $[Y]_{1,2,3} = \#2$, $[Y]_{1,2,3 \oplus_Y 2} = [Y]_2$ and $2 \sim 3$;
6. $1, 3 \sim 4$. Here $[Y]_{1,3} = \#2$, $[Y]_{1,3 \oplus_Y 2} = [Y]_3$ and $3 \sim 4$;

Let $P_\sigma = |\sigma, Y|$ and $Q_\delta = |\delta, Z|$ for positions σ of Y and positions δ of Z. Similar to the proof of Lemma 6.36, we can show that if $\sigma \sim \delta$ and

$P_\sigma = S$ then $Q_\delta = S$;
$P_\sigma = D$ then $Q_\delta = D$;
$P_\sigma = P_{\sigma'}$ then $Q_\delta = Q_{\delta'}$ with $\sigma' \sim \delta'$;
$P_\sigma = P_{\sigma'} \trianglelefteq a \trianglerighteq P_{\sigma''}$ then $Q_\delta = Q_{\sigma'} \trianglelefteq a \trianglerighteq Q''_\sigma$ with $\sigma' \sim \delta'$ and $\sigma'' \sim \delta''$

and vice versa. By Theorem 6.7, $P_\sigma = Q_\delta$ for all positions σ of Y and δ of Z with $\sigma \sim \delta$. Therefore, $|(1,2), Y| = |1, Z|$. This implies that $|\Psi(\mathtt{while} + a; u; X)| = |\Psi(u; \mathtt{while} + a; u; X)| \trianglelefteq a \trianglerighteq |\Psi(X)|$. Likewise we get $|\Psi(\mathtt{while} - a; u; X)| = |\Psi(X)| \trianglelefteq a \trianglerighteq |\Psi(u; \mathtt{while} - a; u; X)|$. □

Theorem 6.48 *The transformation* pgaucw2pgbur *from* PGAucw *to* PGLBur *is correct, i.e., for every program X in* PGAucw*:*

$$|\mathtt{pgaucw2pgbur}(X)| = |X|.$$

Proof: This follows from Definition 6.9, Definition 6.25, Definition 6.38, Lemma 6.46, Lemma 6.47 and Theorem 6.7. □

Projection from PGLBur to PGLBu

This section defines the projection pglbur2pglbu that removes repetition from a program in PGLBur.

Definition 6.49 *The projection* pglbur2pglbu *from* PGLBur *to* PGLBu *is defined on program expressions in first canonical form.*

$$\begin{aligned}
&\mathtt{pglbur2pglbu}(u_1;\ldots;u_k) = \mathtt{pglbur2pglbu}(u_1);\ldots;\mathtt{pglbur2pglbu}(u_k),\\
&\mathtt{pglbur2pglbu}(u_1;\ldots;u_k;(u_{k+1};\ldots;u_{k+n})^\omega) =\\
&\qquad\qquad \mathtt{pglbur2pglbu}(u_1);\ldots;\mathtt{pglbur2pglbu}(u_k);\\
&\qquad\qquad \mathtt{pglbur2pglbu}(u_{k+1});\ldots;\mathtt{pglbur2pglbu}(u_{k+n});(\backslash\#n)^{\max(m,2)},\\
&\mathtt{pglbur2pglbu}(\mathbf{u}(X)) = \mathbf{u}(\mathtt{pglbur2pglbu}(X)),\\
&\mathtt{pglbur2pglbu}(u) = u \text{ otherwise.}
\end{aligned}$$

where m is the maximum of jump counters occurring in $u_1;\ldots;u_{k+n}$ and 0 otherwise.

One can see that the projection pglbur2pglbu from PGLBur to PGLBu is well-defined. Furthermore:

Theorem 6.50 *The projection* pglbur2pglbu *from* PGLBur *to* PGLBu *is correct, i.e., for every program X in* PGLBur*:*

$$|\mathtt{pglbur2pglbu}(X)| = |X|.$$

Proof: Let Ψ denote the projection pglbur2pglbu and $Y = \Psi(X)$. A binary relation $\sim$ between positions of programs X and Y is defined inductively as follows.

1. If $X = u_1;\ldots;u_k$ then for all $1 \leq i \leq k : i \sim i$.
2. If $X = u_1;\ldots;u_k;(u_{k+1};\ldots;u_{k+n})^\omega$ then for all $1 \leq i \leq k+n : i \sim i$ and for all $i > k+n : i \sim k+1+((i-k-1) \mod n)$. Furthermore, for all $k+n < i \leq k+n+\max(m,2) : i-n \sim i$.

For a position σ of X:

1. If $\Delta[X]_\sigma = v_1; \ldots; v_n$ then for all $1 \leq i \leq n : \sigma, i \sim \sigma, i$.
2. If $\Delta[X]_\sigma = v_1; \ldots; v_n; (v_{n+1}; \ldots; v_{n+m})^\omega$ and
$$\Delta[Y]_\sigma = \Psi(v_1); \ldots; \Psi(v_n); \Psi(v_{n+1}); \ldots; \Psi(v_{n+m}); (\backslash \#m)^{\max(t,2)},$$
where t is the maximum of jump counters occurring in $v_1; \ldots; v_{n+m}$ and 0 otherwise, then for all $1 \leq i \leq n+m : \sigma, i \sim \sigma, i$ and for all $i > n+m : \sigma, i \sim \sigma, n+1+(i-n-1) \mod m$. Furthermore, for all $n+m < i \leq n+m+\max(t,2) : \sigma, i-m \sim \sigma, i$.

Let $P_\sigma = |\sigma, X|$ and $Q_\delta = |\delta, Y|$ for positions σ of X and positions δ of Y. Similar to the proof of Lemma 6.36, we can show that if $\sigma \sim \delta$ and

$P_\sigma = S$ then $Q_\delta = S$;
$P_\sigma = D$ then $Q_\delta = D$;
$P_\sigma = P_{\sigma'}$ then $Q_\delta = Q_{\delta'}$ with $\sigma' \sim \delta'$;
$P_\sigma = P_{\sigma'} \trianglelefteq a \trianglerighteq P_{\sigma''}$ then $Q_\delta = Q_{\sigma'} \trianglelefteq a \trianglerighteq Q''_\sigma$ with $\sigma' \sim \delta'$ and $\sigma'' \sim \delta''$

and vice versa. By Theorem 6.7, $P_\sigma = Q_\delta$ for all positions σ of X and δ of Y with $\sigma \sim \delta$. This implies that $|X| = |1, X| = |1, Y| = |Y|$. □

In summary, we have defined and shown the correctness of the projection `pgaucw2pgbur` from PGAucw to PGLBur and the projection `pglbur2pglbu` from PGLBur to PGLBu under the restriction that no consecutive occurrences of while-loops are allowed. These two projections together with the projection `pglbu2pga` from PGLBu to PGA obtained from [82] constitute a projection from PGAucw to PGA. Hence, the while-loop instruction is not needed as a primitive instruction in terms of expressiveness, provided that consecutive occurrences of while-loops are not allowed.

6.6 Concluding remarks

In this chapter, we have explored the expressiveness of two extensions PGAuc and PGAucw of PGA with units, conditional instructions and while-loops respecting the lazy projection semantics. With these extensions, PGA is closer to program notations that are used in practice. For instance, the program written in a modern programming language below

```
X::= WHILE a DO
           WHILE b DO
                 IF c THEN d ELSE e END IF;
           END WHILE;
     END WHILE;
```

is formulated in PGAucw as

$$X = \texttt{while} + a; \mathbf{u}(\texttt{while} + b; \mathbf{u}(\texttt{if} + c; d; e)); !.$$

First of all, we have described the projection `pgauc2pgau` from PGAuc to PGAu. We note that if programs do not contain units, we can define another projection `pgac2pgau` from PGAc to PGAu as follows.

PGAu	$\xrightarrow{\texttt{pgau2pglbu}}$	PGLBu	$\xrightarrow{\texttt{pglbu2pglbg}}$	PGLBg
$\uparrow\texttt{pgauc2pgau}$		$\uparrow\texttt{pglbur2pglbu}$		$\downarrow\texttt{pglbg2pglb}$
PGAuc		PGLBur		PGLB
		$\uparrow\texttt{pgaucw2pgbur}$		$\downarrow\texttt{pglb2pga}$
		PGAucw		PGA

Table 6.7: Projection from PGAuc and PGAucw to PGA.

Definition 6.51 *The projection* pgac2pgau *from* PGAc *to* PGAu *is defined on program expressions in first canonical form.*

$$\begin{aligned}&\texttt{pgac2pgau}(u_1;\ldots;u_k) = \psi_1(u_1);\ldots;\psi_k(u_k),\\ &\texttt{pgac2pgau}(u_1;\ldots;u_n;(u_{n+1};\ldots;u_{n+m})^\omega) =\\ &\qquad \psi_1(u_1);\ldots;\psi_n(u_n);(\psi_{n+1}(u_{n+1});\ldots;\psi_{n+m}(u_{n+m}))^\omega\end{aligned}$$

where $u_{k+1} = \#0$, $u_{n+m+1} = u_{n+1}$ *and the auxiliary operations* ψ_j *are given by*

$$\begin{aligned}\psi_j(\texttt{if} \pm a) &= u(\pm a;\Phi(u_{j+1});\#2),\\ \psi_j(u) &= u \textit{ otherwise}\end{aligned}$$

with

$$\begin{aligned}\Phi(\texttt{if} \pm a) &= a,\\ \Phi(!) &= !,\\ \Phi(a) &= \mathbf{u}(a;\#4),\\ \Phi(\pm a) &= \mathbf{u}(\pm a;\#4),\\ \Phi(\#0) &= \#0,\\ \Phi(\#1) &= \#4,\\ \Phi(\#l+2) &= \#l+3 \textit{ otherwise.}\end{aligned}$$

This projection works only on programs in PGA extended with conditional instructions. However, its resulting program in PGAu is much shorter than the one projected by pgauc2pgau when PGAuc is restricted to PGAc as can be seen in the following example.

Example 6.52

$$\begin{aligned}&\texttt{pgac2pgau}(\texttt{if} + a;\texttt{if} + b;c;!) = \mathbf{u}(+a;b;\#2);\mathbf{u}(+b;\mathbf{u}(c;\#4);\#2);c;!\\ &\texttt{pgauc2pgau}(\texttt{if} + a;\texttt{if} + b;c;!) =\\ &\mathbf{u}(+a;\mathbf{u}(\mathbf{u}{+}b;\mathbf{u}(\#2;\#6);\#2;\#4);\#2);\mathbf{u}(+b;\mathbf{u}(c;\#4);\#2);c;!\end{aligned}$$

Next, we have shown that PGA extended with while-loops yields non-regular program behaviors in certain cases. Under the restriction that consecutive occurrences of while-loops are forbidden, we have defined two projections pgaucw2pgbur from PGAucw to PGLBur and pglbur2pglbu from PGLBur to PGLBu.

Our projections together with the projections covered in [82] constitute two projections `pgauc2pga` and `pgaucw2pga` from PGAuc and PGAucw into PGA, illustrated in Table 6.7. More precisely, the projection `pgauc2pga` is a composition of five mappings `pgauc2pgau`, `pgau2pglbu`, `pglbu2pglbg`, `pglbg2pglb` and `pglb2pga`. The projection `pgaucw2pga` is a composition of five mappings `pgaucw2pgbur`, `pglbur2pglbu`, `pglbu2pglbg`, `pglbg2pglb` and `pglb2pga`.

The existence of these projections shows that conditional statements and while-loops, while allowing for a flexible style of programming, are not needed as primitive instructions in terms of expressiveness.

Part III

Applications of thread and program algebra

Chapter 7

Noninterference in thread algebra

7.1 Introduction

Thread algebra (TA) is introduced by Bergstra and Middelburg in [23, 24, 25] as a semantics for sequential and multi-threaded programming languages. It contains a set of *strategic interleaving operators* that turn a sequence of threads into a single thread capturing the essential aspects of multi-threading. One may argue that thread algebra with strategic interleaving is technically less elegant in dealing with parallelism than process algebras such as CCS [75] and ACP [19] with arbitrary interleaving. However, thread algebra is designed specifically for multi-threaded programs whose executions are on virtual machines that make use of scheduling. In addition, process algebras introduce nondeterminism which might be disadvantageous for a programmer's intuition. On the other hand, in thread algebra, the programmer can always expect what might happen by considering a significant collection of different interleaving strategies. TA is a promising approach for the study of computer viruses and virtual machines [28, 26].

Noninterference [53] (see [88] for an overview) is an important property of secure information flow [13, 44]. It characterizes systems whose execution does not reveal secret information. Formalizing and analyzing this property becomes increasingly important because of the privacy question raised in real-life applications such as mail and banking transactions. Various definitions of noninterference have been introduced. However, as stated in [88], "existing theoretical frameworks for expressing these security properties are inadequate, and practical techniques for enforcing these properties are unsatisfactory".

There are two main classes of information flow security: language-based security [88] and process calculus-based information security [47, 84, 69]. The former deals with the leaking of secret data through the execution of programs, while the latter is concerned with the prevention of secret events from being revealed through the exe-

cution of communicating processes. This chapter discusses noninterference problems for language-based security in a process-algebraic framework like thread algebra.

7.1.1 Related work

A typical example of leaking secret data is described as follows. Suppose that program variables are classified into two classes: $\mathtt{Var}_H$ and $\mathtt{Var}_L$. Program variables $l \in \mathtt{Var}_L$ hold low-security information, while program variables $h \in \mathtt{Var}_H$ hold high-security information. Consider the program

```
X ::= IF h==1 THEN l:=0 ELSE l:=1 END IF.
```

X is insecure since an attacker may learn the value of h through branching on the condition $\langle h == 1 \rangle$. After this condition, the value of l becomes significant.

Most approaches on language-based security are based on type systems [99, 91, 71] which have been shown to be effective. They provide a collection of *typing rules* describing what security type $\mathcal{L}$ (*low*) or $\mathcal{H}$ (*high*) is assigned to a program (or expression) based on the types of subprograms, under two notions of noninterference: *termination-insensitive* and *termination-sensitive*. The former classifies programs under the condition that *the programs terminate successfully*, while the latter considers termination to be *observable*. The typing rules of Volpano, Smith and Irvines [99] concern termination-insensitive noninterference in which any expression can have type $\mathcal{H}$, and an expression can have type $\mathcal{L}$ only if it has no occurrences of high variables. Furthermore, given a conditional statement or while-loop with a high guard, the branches or loop body cannot have type $\mathcal{L}$. For instance, according to these typing rules, the program X above is insecure. The typing rules treated by Volpano and Smith [91] (and others) consider termination-insensitive noninterference in which loop guards are required to have low security. The advantage of the approaches on type systems is that the type checker only needs to work on program texts, so these approaches are decidable and easy to implement. However, the programs must satisfy some structure. Furthermore, these approaches appear confusing when parallelism is introduced in the language.

Other approaches on language-based security are the works in the context of concurrency [86, 89, 87, 36, 85]. They consider the worst-case scenario where an attacker may observe the timing of program execution. These approaches characterize *timing-sensitive* noninterference which essentially says that given two low-equivalent input states, the high parts of memories at any point of computation do not introduce any difference between the low parts of the memories throughout the computation. Recently, a formal link between them and the approaches on process-calculus information security has been established [50]. Since these works are based on states of programs, it is difficult to decide noninterference properties [43].

7.1.2 Contributions and outline of the chapter

In this chapter, we show that thread algebra provides an elegant process-algebraic framework for reasoning about and classifying various notions of noninterference in

sequential and multi-threaded programming languages.

We define *termination-insensitive noninterference* (TINI), *termination-sensitive noninterference* (TSNI) and *timing-sensitive noninterference* (TISNI) for threads in BTA which are analogous to the existing notions of noninterference from the literature. More precisely, a thread is termination-insensitive (termination-sensitive) noninterfering if its behavior, from the view of low actions, is always the same. We note that TINI requires the condition that *the program terminates normally* [99], while TSNI considers termination to be observable. A thread is timing-sensitive noninterfering if its behavior is always the same, assuming that high actions behave similarly. We prove soundness for these definitions, meaning that if a thread satisfies one of these properties then it satisfies the noninterference property proposed by Goguen and Meseguer [53]. In the setting of thread algebra, we show that TISNI satisfies *compositionality* with respect to the *cyclic interleaving operator*, the basic strategic interleaving of [23] which turns a thread vector of arbitrary length into a single thread in a round-robin fashion. To preserve compositionality for TINI and TSNI, we propose a particular interleaving strategy for thread algebra called the *cyclic strategic interleaving with persistence* operator. This strategy invokes the rotation of the thread vector only in the case that the current action is not a high action. Hence, it maintains the order of the high actions, and therefore, the analysis can be made compositional.

The advantage of our approach is that it is suitable for considering security properties of unstructured and multi-threaded programming languages. In case of TINI and TSNI, we accept all secure programs that are typable by type systems such as in [99, 91]. Furthermore, we also accept certain commands which are rejected by most type systems. For instance, the commands $l{:=}h{-}h$, `IF` $h{==}0$ `THEN` $l{:=}1$ `ELSE` $l{:=}1$ `END IF` and `WHILE` $l+h>h$ `DO` $l{:=}1$ `END WHILE` where h holds high-security information and l holds low-security information, are not typable by the checker in [99]. (These commands have recently been considered in [35].) In case of TISNI, our definition is not as precise as the definition given in [86]. However, it is decidable and easy to implement. Like [50], we can also use existing tools such as in [48, 49, 31] for checking process-equivalence to develop our security checkers.

The structure of the chapter is as follows. Section 7.2 recalls the basic concepts of basic thread algebra and thread algebra. We also introduce the notion of noninterference given by Goguen and Meseguer [53] and the type systems of [99, 91]. Section 7.3 characterizes actions and defines abstraction of high actions for threads. Section 7.4 provides bisimulation and bisimulation up to a set of high actions, in order to define noninterference properties. Section 7.5 presents our definitions of noninterference. Section 7.6 shows compositionality of TISNI with respect to the cyclic interleaving operator. Section 7.7 introduces the cyclic interleaving with persistence operator for thread algebra. Section 7.8 shows that TINI and TSNI satisfy compositionality with respect to the cyclic interleaving with persistence operator. The chapter is ended with some concluding remarks in Section 7.9.

7.2 Preliminaries

This section recalls from [22, 23, 25] the basic concepts of *basic thread algebra* (BTA) and *thread algebra* (TA). We note that basic thread algebra was introduced as *basic polarized process algebra* (BPPA) in [22]. Furthermore, we introduce a simple programming language `Lang` which is used to illustrate our approach. We also describe the earliest notion of noninterference given by Goguen and Meseguer [53] and the type systems of [99, 91] for checking this property.

7.2.1 Basic thread algebra (BTA)

Primitives of basic thread algebra

Let Σ be a set of *actions*. Each action is supposed to return a boolean value after its execution. BTA is constructed over Σ by the following operators:

Successful termination : $S \in \mathrm{BTA}$ yields successful terminating behavior.

Unsuccessful termination or **deadlock**: $D \in \mathrm{BTA}$ represents inactive behavior.

Postconditional composition : $(-) \trianglelefteq a \trianglerighteq (-)$ with $a \in \Sigma$. The thread $P \trianglelefteq a \trianglerighteq Q$ first performs a and then proceeds with P if true was returned and with Q otherwise. In case $P = Q$ we abbreviate this thread by the **action prefix** $a \circ (-)$: $a \circ P = P \trianglelefteq a \trianglerighteq P$.

Let BTA_Σ denote the set of **finite** threads which are made from S and D by means of a finite number of applications of postconditional composition. Threads can be infinite. To define an **infinite** thread in BTA, we use a sequence of its finite approximations.

Definition 7.1 *For every $n \in \mathbb{N}$, the* **approximation operator** $\pi_n : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *is defined inductively by*

$$\begin{array}{lcl} \pi_0(P) & = & D, \\ \pi_{n+1}(S) & = & S, \\ \pi_{n+1}(D) & = & D, \\ \pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q). \end{array}$$

A **projective sequence** *is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$, $\pi_n(P_{n+1}) = P_n$.*

We note that for all (finite or infinite) threads P and Q, $P = Q \Leftrightarrow \forall n \in \mathbb{N} : \pi_n(P) = \pi_n(Q)$. If $P = a \circ a \circ \cdots$ (P can do subsequently infinitely many actions a), then we write $P = a^\infty$. Let $\mathrm{BTA}_\Sigma^\infty$ be the set of all threads represented by projective sequences in BTA_Σ. Then $\mathrm{BTA}_\Sigma \subseteq \mathrm{BTA}_\Sigma^\infty$.

BTA as a complete partial order

It is shown in [16] that the domain $\mathrm{BTA}_\Sigma^\infty$ of BTA is a *complete partial order* (cpo) which essentially says that there always exists a supremum for every monotone sequence. This is an important property of BTA which allows us to represent an infinite thread in BTA as a *monotone sequence* of finite threads. Formally:

Definition 7.2 *A* **complete partial order** *(cpo) $\mathcal{D} = (\mathcal{D}, \sqsubseteq)$ is a partially ordered set with a least element such that every monotone sequence has a supremum in $\mathcal{D}$.*

In the next definition, we define a partial order on threads in BTA.

Definition 7.3

1. *The* **partial ordering** $\sqsubseteq$ *on finite threads is generated by the clauses*
   (a) *for all $P \in \mathrm{BTA}_\Sigma$, $D \sqsubseteq P$, and*
   (b) *for all $P, Q, X, Y \in \mathrm{BTA}_\Sigma$, $a \in \Sigma$, $P \sqsubseteq X \,\&\, Q \sqsubseteq Y \Rightarrow P \trianglelefteq a \trianglerighteq Q \sqsubseteq X \trianglelefteq a \trianglerighteq Y$.*
2. *For two infinite threads $P = (P_n)_n$ and $Q = (Q_n)_n$ with $(P_n)_n$ and $(Q_n)_n$ projective sequences, $P \sqsubseteq Q \Leftrightarrow \forall n \in \mathbb{N} \quad P_n \sqsubseteq Q_n$.*

Definition 7.4 *A* **monotone sequence** *is a sequence $(P_n)_n$ satisfying*

$$P_0 \sqsubseteq P_1 \sqsubseteq \cdots \sqsubseteq P_n \sqsubseteq P_{n+1} \sqsubseteq \cdots$$

Theorem 7.5 *($\mathrm{BTA}_\Sigma^\infty, \sqsubseteq$) is a complete partial order.*

Proof: See [16]. □

Regular threads

Regular threads in BTA are defined as follows.

Definition 7.6 *A thread P is* **regular** *over Σ if $P = E_1$, where E_1 is defined by a finite system of the form ($n \geq 1$):*

$$\{E_i = t_i | 1 \leq i \leq n, t_i = S \text{ or } t_i = D \text{ or } t_i = E_{il} \trianglelefteq a_i \trianglerighteq E_{ir}\}$$

with $E_{il}, E_{ir} \in \{E_1, \ldots, E_n\}$ and $a_i \in \Sigma$.

These regular threads are well-defined in the domain $(\mathrm{BTA}_\Sigma^\infty, \sqsubseteq)$ (see Chapter 5) and are used to represent program behaviors.

7.2.2 Thread algebra

Thread algebra is designed on top of BTA and is meant to specify a collection of *strategic interleaving operators*, capturing some essential aspects of multi-threading. We assume that a collection of threads to be interleaved takes the form of a sequence, called a *thread vector*. Strategic interleaving operators turn a thread vector of arbitrary length into a single thread. This single thread obtained via a strategic interleaving operator is called a *multi-thread*. For simplicity, we consider in this chapter only the basic interleaving strategy called *cyclic interleaving* [23]. This interleaving strategy works in a round-robin fashion which invokes the rotation of a thread vector at every step. Furthermore, if one thread in the thread vector deadlocks, the whole does not deadlock till all others have terminated or deadlocked.

Let $\langle\rangle$ denote the empty sequence, $\langle x\rangle$ stand for a sequence of length one, and $\alpha \frown \beta$ the concatenation of two sequences. We assume that the following identity holds: $\alpha \frown \langle\rangle = \langle\rangle \frown \alpha = \alpha$.

Definition 7.7 *The axioms for the* **cyclic strategic interleaving** $\|_{csi}$ *operator are given as follows:*

$$\begin{array}{lcl}
\|_{csi}\,(\langle\rangle) & = & S \\
\|_{csi}\,(\langle S\rangle \frown \alpha) & = & \|_{csi}\,(\alpha) \\
\|_{csi}\,(\langle D\rangle \frown \alpha) & = & S_D(\|_{csi}\,(\alpha)) \\
\|_{csi}\,(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) & = & \|_{csi}\,(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csi}\,(\alpha \frown \langle y\rangle)
\end{array}$$

where the auxiliary deadlock at termination operator S_D turns termination into deadlock and is defined by

$$\begin{array}{lcl}
S_D(S) & = & D \\
S_D(D) & = & D \\
S_D(x \trianglelefteq a \trianglerighteq y) & = & S_D(x) \trianglelefteq a \trianglerighteq S_D(y).
\end{array}$$

For a thread vector α of arbitrary (finite or infinite) threads $\alpha = \alpha_1 \frown \cdots \frown \alpha_n$, $\|_{csi}\,(\alpha)$ is determined by its projective sequence:

$$\pi_n(\|_{csi}\,(\alpha)) = \pi_n(\|_{csi}\,(\pi_n(\alpha_1) \frown \cdots \frown \pi_n(\alpha_n))).$$

Example 7.8 Let $P = a^\infty$ and $Q = b^\infty$ be two single threads. Then $\|_{csi}\,(\langle P\rangle \frown \langle Q\rangle) = a \circ b \circ a \circ b \cdots$.

7.2.3 The programming language Lang

It has been outlined in [22, 15, 23] that program behaviors of sequential and multi-threaded programming languages can be represented as threads in BTA. To illustrate our approach, we consider threads as program behaviors of programs written in a simple imperative programming language `Lang` which, similar to that of [99, 91], is defined as follows.

$$\begin{array}{lcl}
X, Y, \ldots & ::= & x := e \mid \\
 & & \texttt{IF}\ e\ \texttt{THEN}\ X\ \texttt{ELSE}\ Y\ \texttt{END IF} \mid \\
 & & \texttt{WHILE}\ e\ \texttt{DO}\ X\ \texttt{END WHILE}
\end{array}$$

where e stands for a boolean or an arithmetic expression, whose syntax we do not describe here.

We now consider assignments $x{:=}e$ and tests e of `Lang` as the actions in Σ, written as $[x{:=}e]$ and $\langle e\rangle$. Then program behaviors of `Lang` are regular threads, as illustrated in the following example.

Example 7.9 Let X and Y be given as follows.

```
X::= IF h==1 THEN l:=1 ELSE l:=0 END IF.
Y::= l:=0;
     WHILE h==0 DO
           h:=0;
     END WHILE;
     l:=1.
```

The behaviors of X and Y, denoted by $|X|$ and $|Y|$, are determined by: $|X| = ([l{:=}1] \circ S) \trianglelefteq \langle h{==}1\rangle \trianglerighteq ([l{:=}0] \circ S)$ and $|Y| = [l{:=}0] \circ P$, where $P = ([h{:=}0] \circ P) \trianglelefteq \langle h{==}0\rangle \trianglerighteq ([l{:=}1] \circ S)$.

7.2.4 Input-output transformations

In this section, we present the notion of *input-output transformations* of program behaviors. This notion is based on the *effect* and *yield* of an action on a state space. We assume the existence of a set `Var` of program variables, and a state space $\mathbb{S}$ whose elements play the role of inputs as well as of outputs of programs (or threads) in the programming language `Lang`. All results of the chapter are related to these two sets.

The effect and yield of an action on the state space $\mathbb{S}$

Suppose that upon the execution of a program, the values of a program variable x are ranging over the set $\texttt{values}(x)$. A **state of a variable** is a pair of the form $\langle x, v\rangle$ with $v \in \texttt{values}(x)$. A **state of the state space** $\mathbb{S}$ is a set s consisting of states of all variables in `Var`, in which there is a one-to-one correspondence between variables and their states. If s is a state of the state space and $\langle x, v\rangle \in s$, we write $s.x.\texttt{value} = v$.

For an action $a \in \Sigma$, there is an operation $\texttt{effect}_a : \mathbb{S} \to \mathbb{S}$ that changes the *state* due to the execution of a called the **effect** operation, and an operation $y_a : \mathbb{S} \to \{\texttt{true}, \texttt{false}\}$ that determines a boolean value when a is performed in a state of $\mathbb{S}$, called the **yield** operation.

We assume that if an action has some effect on the state space then its yield always determines true. Formally, for an action $a \in \Sigma$, if there exists a state $s \in \mathbb{S}$ such that $\texttt{effect}_a(s) \neq s$ then $y_a(s') = \texttt{true}$ for all states $s' \in \mathbb{S}$.

In the following, we will specifically define the effect and yield of an assignment and a test on the state space $\mathbb{S}$. An assignment $[x{:=}e]$ may have effect on the state space and always returns `true` after its execution. Formally, for all states $s \in \mathbb{S}$,

$$\begin{array}{lcl} \texttt{effect}_{[x:=e]}(s) & = & s \setminus \{\langle x, s.x.\texttt{value}\rangle\} \cup \{\langle x, \sigma(e)\rangle\}, \\ y_{[x:=e]}(s) & = & \texttt{true} \end{array}$$

where $\sigma(e)$ is obtained by first replacing occurrences of variables y in the expression e by $s.y.\texttt{value}$, and then calculating its value in the set $\texttt{values}(x)$.

A test $\langle e \rangle$ has no effect on the state space, and can produce a negative reply. Formally, for all states $s \in \mathbb{S}$, $\texttt{effect}_{\langle e \rangle}(s) = s$ and $y_{\langle e \rangle} = \sigma(e)$, where $\sigma(e)$ is obtained by first replacing occurrences of variables y in e by $s.y.\texttt{value}$, and then computing the result in the set $\{\texttt{true}, \texttt{false}\}$.

Program behavior and input-output relations

Input-output transformations are derived from program behaviors rather than from programs themselves. An action of a program behavior must be viewed as a transformation of the states in a state space, producing a boolean whenever applied. This action is taken as an **input value** of a behavior. The state reached after the final action has been performed represents the **output value** of a computation. Let D represent a failure value that can't be computed. The notion of input-output transformations can be captured formally as follows.

Definition 7.10 *Given a finite thread P, a function $P \bullet (-) : \mathbb{S} \to \mathbb{S} \cup \{D\}$ which represents what P computes on an input s in $\mathbb{S}$ is defined inductively as follows.*

1. $D \bullet s = D$,
2. $S \bullet s = s$,
3. $(a \circ P) \bullet s = P \bullet \texttt{effect}_a(s)$,
4. $P \trianglelefteq a \trianglerighteq Q \bullet s = (a \circ P) \bullet s$ *if* $y_a(s) = \texttt{true}$, *otherwise* $(a \circ Q) \bullet s$.

In case P is infinite, i.e. $P = (P_n)_n$ for $(P_n)_n$ a monotone sequence, $P \bullet s = \bigsqcup_n P_n \bullet s$. Note that the partial ordering $\leq$ on $\mathbb{S}$ is defined by $D \leq s$ for all $s \in \mathbb{S}$. If $P \bullet s = D$ for a state $s \in \mathbb{S}$ then we say that this computation produces no result. In other words, $P \bullet s = D$ precisely if for all $n \in \mathbb{N}$, $\pi_n(P) \bullet s = D$.

The previous definition suggests an equivalence called *input-output equivalence* for threads.

Definition 7.11 *Two threads P and Q are* **input-output equivalent** *($P \approx_{\mathbb{S}} Q$) over the state space $\mathbb{S}$ if $P \bullet s = Q \bullet s$ for all $s \in \mathbb{S}$.*

We adopt the following convention on states of the state space $\mathbb{S}$: if in a program behavior only the variables $x_1, \ldots, x_k$ occur, we represent states simply as $[\langle x_1, v_1 \rangle, \ldots, \langle x_k, v_k \rangle]$ with $v_i \in \texttt{values}(x_i)$ for $1 \leq i \leq k$.

Example 7.12 Consider the program

```
X ::= WHILE x>0 DO
            x:=x+1;
      END WHILE.
```

Then the behavior $|X|$ of X is defined as $|X| = ([x := x + 1] \circ |X|) \trianglelefteq \langle x > 0 \rangle \trianglerighteq S$. The effect and yield of the actions $[x{:=}x+1]$ and $\langle x > 0 \rangle$ are given as follows. For all $v \in \mathtt{values}(x)$, $\mathtt{effect}_{[x:=x+1]}([\langle x, v \rangle]) = [\langle x, v + 1 \rangle]$ and $y_{[x:=x+1]}([\langle x, v \rangle]) = \mathtt{true}$. Moreover, $\mathtt{effect}_{\langle x>0 \rangle}([\langle x, v \rangle]) = [\langle x, v \rangle]$, and $y_{\langle x>0 \rangle}([\langle x, v \rangle]) = \mathtt{true}$ if $v > 0$ and $y_{\langle x>0 \rangle}([\langle x, v \rangle]) = \mathtt{false}$ otherwise. It is easy to see that if initially $v > 0$, then the computation $|X| \bullet [\langle x, v \rangle]$ goes on forever, i.e., X produces no result for every input v of x that is greater than 0.

7.2.5 Noninterference based on input-output transformations

The earliest definition of *noninterference* of security information flow was given by Goguen and Meseguer [53]. Following the idea of [53], we provide a definition of noninterference based on input-output transformations for threads in BTA.

We suppose that program variables in $\mathtt{Var}$ are classified into two security classes $\mathtt{Var}_L$ (low) and $\mathtt{Var}_H$ (high), $\mathtt{Var}_L \cap \mathtt{Var}_H = \emptyset$ and $\mathtt{Var}_L \cup \mathtt{Var}_H = \mathtt{Var}$. We provide some notions of equivalence for states and threads based on security classes as follows.

Definition 7.13 *Two states s and t of the state space $\mathbb{S}$ are* **low equivalent**, *denoted by $s =^L_{\mathbb{S}} t$, if for all $l \in \mathtt{Var}_L$, $s.l.value = t.l.value$, otherwise $s \neq^L_{\mathbb{S}} t$. Similarly, two states s and t of the state space $\mathbb{S}$ are* **high equivalent**, *denoted by $s =^H_{\mathbb{S}} t$, if for all $h \in \mathtt{Var}_H$, $s.h.value = t.h.value$, otherwise $s \neq^H_{\mathbb{S}} t$.*

Definition 7.14 *Two threads P and Q are* **low quasi-equivalent** *($P \approx^L_{\mathbb{S}} Q$) over the state space $\mathbb{S}$ if for all low equivalent states s and t ($s =^L_{\mathbb{S}} t$), $P \bullet s = Q \bullet t = D$ or $P \bullet s$ and $Q \bullet t$ are low equivalent ($P \bullet s =^L_{\mathbb{S}} Q \bullet t$).*

Informally speaking, a program is secure if its low output does not depend on its high input. This notion is translated to threads in BTA as follows.

Definition 7.15 *A thread P is* **noninterfering** *($P \in \mathtt{NI}$) if it is low quasi-equivalent to itself. A program is* **secure** *if its behavior is noninterfering.*

Example 7.16 Let $h \in \mathtt{Var}_H$ and $l \in \mathtt{Var}_L$. Consider the program behavior below.

$$\begin{array}{lcl} P & = & Q \trianglelefteq \langle h == 1 \rangle \trianglerighteq R, \\ Q & = & [l := l + 1] \circ Q, \\ R & = & [l := l - 1] \circ R. \end{array}$$

It can be derived that $P \approx_{\mathbb{S}} D$. Hence $P \approx^L_{\mathbb{S}} P$. Thus, $P \in \mathtt{NI}$.

The previous example shows that non-termination implies NI. The following example illustrates insecure programs.

Example 7.17 Let $h \in \mathtt{Var}_H$ and $l \in \mathtt{Var}_L$. Consider the programs X and Y given in Example 7.9.

$$\begin{array}{lcl} |X| & = & ([l{:=}1] \circ S) \trianglelefteq \langle h{==}1 \rangle \trianglerighteq ([l{:=}0] \circ S) \\ |Y| & = & [l{:=}0] \circ P \end{array}$$

where $P = ([h{:=}0] \circ P) \trianglelefteq \langle h{==}0\rangle \trianglerighteq ([l{:=}\,1] \circ S)$. One can check that Y produces no result in the case that the input of h is 0. Furthermore, X and Y are not secure, since

$$|X| \bullet [\langle h,0\rangle,\langle l,0\rangle] = [\langle h,0\rangle,\langle l,0\rangle] \quad \textit{while} \quad |X| \bullet [\langle h,1\rangle,\langle l,0\rangle] = [\langle h,1\rangle,\langle l,1\rangle];$$
$$|Y| \bullet [\langle h,0\rangle,\langle l,0\rangle] = D \quad \textit{while} \quad |Y| \bullet [\langle h,1\rangle,\langle l,0\rangle] = [\langle h,1\rangle,\langle l,1\rangle].$$

7.2.6 Noninterference based on type systems

The definition of noninterference given by Goguen and Meseguer [53] is precise, but might require a complex computation. To simplify the checking, the approaches based on *type systems* [99, 91] (see [88] for an overview) have been developed. In these approaches, if a program is *well-typed* according to the *typing rules* of a type system then it has the noninterference property. This section introduces the type systems of [99, 91] in order to compare their results with ours later.

We suppose that there are two security classes $\mathcal{L}$ (low) and $\mathcal{H}$ (high), and a partial order $\subseteq$ between security classes with $\mathcal{L} \subseteq \mathcal{H}$ ($\mathcal{L} \neq \mathcal{H}$) ($\mathcal{L}$ is a subtyping of $\mathcal{H}$). The types used in the type systems of [99, 91] are:

$$\begin{array}{llllllll} (\textit{data types}) & \tau & ::= & \mathcal{L} & | & \mathcal{H} & & \\ (\textit{phrase types}) & \rho & ::= & \tau & | & \tau\,\texttt{var} & | & \tau\,\texttt{cmd} \end{array}$$

Here type $\tau\,\texttt{var}$ is the type of a program variable. Type judgments are of the form $\gamma \vdash X : \tau\,\texttt{cmd}$ where X is an expression or a program and γ is a mapping from variables to types of variables, i.e. $\gamma(x) = \mathcal{L}\,\texttt{var}$ if $x \in \texttt{Var}_L$ and $\gamma(x) = \mathcal{H}\,\texttt{var}$ otherwise. The typing rules of [99] are given in Table 7.1. We assume that all constants have type $\mathcal{L}$ (rule (INT)). Furthermore, we omit typing rules for some expressions since they are similar to rule (SUM).

According to these typing rules, every test and every expression is well-typed. In particular, a test (or an expression) has type $\mathcal{H}$ if it contains a high variable, otherwise it has type $\mathcal{L}$.

Assignments of the form $x{:=}e$, where $x \in \texttt{Var}_L$ and e contains a high variable, are untypable in this type system because of the rule (ASSIGN). By this rule, $x{:=}e$ is accepted and has type $\mathcal{L}\,\texttt{cmd}$ if both x and e have type $\mathcal{L}$, or it is accepted and has type $\mathcal{H}\,\texttt{cmd}$ if x has type $\mathcal{H}$.

The meaning of $\gamma \vdash X : \tau\,\texttt{cmd}$ is that type τ is a lower bound for the security level of the assigned variables of X. Hence, if the condition of a well-typed conditional statement (or a well-typed while-loop) has type $\mathcal{H}$ then every assigned variable contained in its branches (or its body) has type $\mathcal{H}$ as well.

Example 7.18 We consider the insecure program X taken from Example 7.9.

```
X ::= IF h==1 THEN l:=1 ELSE l:=0 END IF
```

where $l \in \texttt{Var}_L$ and $h \in \texttt{Var}_H$. This program is untypable in the type system of [99]. Here the condition $h == 1$ has type $\mathcal{H}$. According to rule (IF), X is accepted only if both $l{:=}0$ and $l{:=}1$ have type $\mathcal{H}\,\texttt{cmd}$ or a lower one. However, these assignments have type $\mathcal{L}\,\texttt{cmd}$ which contains type $\mathcal{H}\,\texttt{cmd}$ (rule (CMD$^-$)).

(IDENT)	$\dfrac{\gamma(x) = \rho}{\vdash x : \rho}$
(INT)	$\vdash n : \mathcal{L}$
(R-VAL)	$\dfrac{\vdash e : \tau\ \texttt{var}}{\vdash e : \tau}$
(SUM)	$\dfrac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 + e_2 : \tau}$
(ASSIGN)	$\dfrac{\vdash x : \tau\ \texttt{var} \quad \vdash e : \tau}{\vdash x{:=}e : \tau\ \texttt{cmd}}$
(COMPOSE)	$\dfrac{\vdash c_1 : \tau\ \texttt{cmd} \quad \vdash c_2 : \tau\ \texttt{cmd}}{\vdash c_1; c_2 : \tau\ \texttt{cmd}}$
(IF)	$\dfrac{\vdash e : \tau \quad \vdash c_1 : \tau\ \texttt{cmd} \quad \vdash c_2 : \tau\ \texttt{cmd}}{\vdash \texttt{IF}\ e\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{END IF} : \tau\ \texttt{cmd}}$
(WHILE)	$\dfrac{\vdash e : \tau \quad \vdash c : \tau\ \texttt{cmd}}{\vdash \texttt{WHILE}\ e\ \texttt{DO}\ c\ \texttt{END WHILE} : \tau\ \texttt{cmd}}$
(BASE)	$\mathcal{L} \subseteq \mathcal{H}$
(REFLEX)	$\rho \subseteq \rho$
(CMD$^-$)	$\dfrac{\tau_1 \subseteq \tau_2}{\tau_1\ \texttt{cmd} \supseteq \tau_2\ \texttt{cmd}}$
(SUBTYPE)	$\dfrac{\vdash X : \rho_1 \quad \rho_1 \subseteq \rho_2}{\vdash X : \rho_2}$

Table 7.1: Typing and subtyping rules.

Example 7.19 Similar to the previous example, the following while-loop statement

```
WHILE h==1 DO
      l:=0;
END WHILE
```

is not well-typed, either.

We note that the typing rules in Table 7.1 respect termination-insensitive noninterference. That is to say, a well-typed program is secure by these typing rules if the

(WHILE-TSNI)	$\dfrac{\vdash e : \mathcal{L} \quad \vdash c : \mathcal{L}\,\mathtt{cmd}}{\vdash \mathtt{WHILE}\ e\ \mathtt{DO}\ c\ \mathtt{END}\ \ \mathtt{WHILE} : \mathcal{L}\,\mathtt{cmd}}$

Table 7.2: A typing rule for while-loop with respect to TSNI.

program terminates successfully. To deal with termination-sensitive noninterference, in [91] Smith and Volpano do not allow the guard of a while-loop holding high-security information and the while-loop itself can only get type $\mathcal{L}\,\mathtt{cmd}$. More precisely, they replace rule (WHILE) in Table 7.1 by rule (WHILE-TSNI) in Table 7.2.

In Section 7.5 we will present alternative definitions of noninterference based on program behaviors with certain advantages. We first characterize actions of program behaviors in Section 7.3, and define a bisimulation equivalence that identifies program behaviors with respect to security in Section 7.4.

7.3 Characterizing actions with respect to security

This section characterizes actions of a thread as *insecure, secure, invisible, low* and *high* actions with respect to security. We also define two *abstraction* operators that abstract from *internal* and high actions of a thread. Here the *internal* action $t \in \Sigma$ can have effect on the state space (but not on the low space) and can yield a negative reply.

7.3.1 Secure, low, invisible and high actions

To illustrate our definition of characterizing actions, we consider the following examples. The untypable action $[l{:=}h]$ with $l \in \mathtt{Var}_L$ and $h \in \mathtt{Var}_H$ is *insecure* in our approach because it reveals information of h. The assignment $[l{:=}1]$ is a *low* action because it has effect on the low subspace. The test $\langle h{==}1\rangle$ and the assignment $[h{:=}1]$ are regarded as *high* actions since they have something to do with the high subspace. Finally, the secure actions that have no effect on the low subspace such as tests and high actions are *invisible*. Formally:

Definition 7.20

1. *An action a is* **secure** *if it does not reveal any high-security information, i.e.* $\mathtt{effect}_a(s) =_{\mathbb{S}}^{L} \mathtt{effect}_a(t)$ *for all low equivalent states $s, t \in \mathbb{S}$ ($s =_{\mathbb{S}}^{L} t$).*

2. *A secure action a is* **low** *if it has effect on the low subspace, i.e.* $\mathtt{effect}_a(s) \neq_{\mathbb{S}}^{L} s$ *for some $s \in \mathbb{S}$.*

3. *A secure action a is* **invisible** *if it has no effect on the low subspace, i.e.* $\mathtt{effect}_a(s) =_{\mathbb{S}}^{L} s$ *for all states $s \in \mathbb{S}$.*

4. *An invisible action a is* **high** *if it has effect or yield on the high subspace, i.e.* $\texttt{effect}_a(s) \neq^H_{\mathbb{S}} s$ *for some state* $s \in \mathbb{S}$*, or* $y_a(s) \neq y_a(t)$ *for some states* $s, t \in \mathbb{S}$ *such that* $s \neq^H_{\mathbb{S}} t$*.*

We note that the terminology that actions are secure (or not) in Definition 7.20 is obtained from the standard terminology of secure information flow of [99].

Definition 7.20 allows us to accept certain assignments and programs which are rejected by most type systems (for instance that of [99]) as can be seen in the following examples.

Example 7.21 Let $l \in \texttt{Var}_L$ and $h \in \texttt{Var}_H$ be low and high program variables. Consider an action a of the form $[l{:=}h - h]$. Then the effect of a on $\mathbb{S}$ is given by $\texttt{effect}_a(s) = s \setminus \{\langle l, s.l.\texttt{value}\rangle\} \cup \{\langle l, 0\rangle\}$ for all $s \in \mathbb{S}$. By our definition, a is secure. However, a is untypable in the type system of [99] because the type of $h - h$ contains the type of l.

Example 7.22 Let $l \in \texttt{Var}_L$ and $h \in \texttt{Var}_H$ be low and high program variables. The program

```
X ::= WHILE l+h>h DO
            l:=1;
      END WHILE.
```

is insecure in type systems, since the while-loop guard has type $\mathcal{H}$ while the assignment within the while-loop has type $\mathcal{L}\,\texttt{cmd}$. In our approach, although the while-loop guard contains the high variable h, its value does not depend on the value of h at all, and therefore, it is a low action. Thus, this program can be shown to be secure in our approach (see Section 7.5).

Let Σ_S be the set of secure actions, Σ_I the set of invisible actions, Σ_L the set of low actions and Σ_H the set of high actions. Then $\Sigma_S = \Sigma_I \cup \Sigma_L \subseteq \Sigma$, $\Sigma_I \cap \Sigma_L = \emptyset$ and $\Sigma_H \subseteq \Sigma_I$.

Lemma 7.23 *Let a be an action in Σ, $a \notin \Sigma_H$. Then for all low equivalent states $s, t \in \mathbb{S}$, $y_a(s) = y_a(t)$.*

Proof: We distinguish three cases:

1. $s =^H_{\mathbb{S}} t$. Since $s =^L_{\mathbb{S}} t$, $s = t$. Thus, $y_a(s) = y_a(t)$.
2. $s \neq^H_{\mathbb{S}} t$ and a has some effect on the state space. Then $y_a(s) = y_a(t) = \texttt{true}$.
3. $s \neq^H_{\mathbb{S}} t$ and a has no effect on the state space. If $y_a(s) \neq y_a(t)$ then a has yield on the high subspace. By Definition 7.20, a is a high action. This contradicts the assumption that $a \notin \Sigma_H$. Therefore, $y_a(s) = y_a(t)$.

□

To choose secure, invisible, low and high actions, let $\Sigma_{\texttt{welltyped}}$ be the set of well-typed actions by the typing rules in Table 7.1, $\Sigma_{\mathcal{L}+\mathcal{H}}$ the set of tests, $\Sigma_{\mathcal{L}\,\texttt{cmd}}$ the set of assignments with type $\mathcal{L}\,\texttt{cmd}$, and $\Sigma_{\mathcal{H}+\mathcal{H}\,\texttt{cmd}}$ the set of actions with type $\mathcal{H}$ and $\mathcal{H}\,\texttt{cmd}$ (see Section 7.2.6). Then:

Lemma 7.24

1. $\Sigma_{\texttt{welltyped}} \subseteq \Sigma_S$.
2. $\Sigma_{\mathcal{H}+\mathcal{H}\,\texttt{cmd}} \cup \Sigma_{\mathcal{L}+\mathcal{H}} \subseteq \Sigma_I$.
3. $\Sigma_L \subseteq \Sigma_{\mathcal{L}\,\texttt{cmd}} \subseteq \Sigma_L \cup \Sigma_I = \Sigma_S$.
4. $\Sigma_H \subseteq \Sigma_{\mathcal{H}+\mathcal{H}\,\texttt{cmd}} \subseteq \Sigma_H \cup \Sigma_I = \Sigma_I$.

Hence one can choose the sets Σ_S of secure actions, Σ_L of low actions and Σ_H of high actions as the sets $\Sigma_{\texttt{welltyped}}$, $\Sigma_{\mathcal{L}\,\texttt{cmd}}$ and $\Sigma_{\mathcal{H}+\mathcal{H}\,\texttt{cmd}}$, respectively. The set Σ_I of invisible actions can be chosen as $\Sigma_I = \Sigma_H \cup \Sigma_{\mathcal{L}+\mathcal{H}}$. The previous lemma shows that these decisions are merely approximations of Σ_S, Σ_L, Σ_H and Σ_I. However, we can extend or restrict these sets with certain associated actions, in order to optimize them. The closer we get to the optimal solutions, the more secure programs can be accepted. For instance, the set Σ_S can be extended with actions of the form $[l{:=}h - h]$ as given in Example 7.21. The set Σ_H can be restricted by removing the actions $\langle l + h > h\rangle$ from Example 7.22.

7.3.2 The pre-abstraction operator

To define abstraction of high actions, we present a *pre-abstraction* operator that renames all actions of a set Φ to the internal action t. This operator was first introduced in [5].

Definition 7.25

1. *The* **pre-abstraction** *operator* $t_\Phi : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *for some* $\Phi \subseteq \Sigma$ *is defined as follows*

$$\begin{array}{lcl} t_\Phi(S) & = & S, \\ t_\Phi(D) & = & D, \\ t_\Phi(P \trianglelefteq a \trianglerighteq Q) & = & t_\Phi(P) \trianglelefteq t \trianglerighteq t_\Phi(Q) \text{ for } a \in \Phi, \\ t_\Phi(P \trianglelefteq a \trianglerighteq Q) & = & t_\Phi(P) \trianglelefteq a \trianglerighteq t_\Phi(Q) \text{ otherwise.} \end{array}$$

2. *For an infinite process* $P = (P_n)_{n\in\mathbb{N}}$ *with* $(P_n)_n$ *is a projective sequence, then* $t_\Phi(P) = (t_\Phi(P_n))_n$. *Thus* $t_\Phi(P) \in \mathrm{BTA}_\Sigma^\infty$.

7.3.3 Abstraction of internal actions

This section introduces an *abstraction* operator that omits occurrences of internal actions, assuming that a thread always behaves the same from the view of non-internal actions. This operator is similar to the abstraction operator that abstracts from concrete internal actions of [16].

Definition 7.26 *The abstraction operator* $\tau_t : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *is defined as follows:*

$$\begin{array}{lcll} \tau_t(S) & = & S, & \\ \tau_t(D) & = & D, & \\ \tau_t(P \trianglelefteq t \trianglerighteq Q) & = & \tau_t(P), & \\ \tau_t(P \trianglelefteq a \trianglerighteq Q) & = & \tau_t(P) \trianglelefteq a \trianglerighteq \tau_t(Q) & \text{for } a \neq t. \end{array}$$

One can show that this operator is monotone, i.e.:

Lemma 7.27 *For all* $P, Q \in \mathrm{BTA}_\Sigma$, $P \sqsubseteq Q \Rightarrow \tau_t(P) \sqsubseteq \tau_t(Q)$.

Lemma 7.27 suggests the abstraction of internal actions in an infinite process $P = (P_n)_{n\in\mathbb{N}}$ with $(P_n)_n$ a projective sequence as follows:

Definition 7.28 $\tau_t(P) = \bigsqcup_n \tau_t(P_n)$.

Since $\mathrm{BTA}_\Sigma^\infty$ is a cpo, $\tau_t(P) \in \mathrm{BTA}_\Sigma^\infty$ if $P \in \mathrm{BTA}_\Sigma^\infty$ (see [16, 104]).

Example 7.29 $\tau_t((a \circ S) \trianglelefteq t \trianglerighteq (b \circ S)) = a \circ S$, $\tau_t(t^\infty) = D$.

7.3.4 Abstraction of high actions

Assuming that a thread always behaves the same after the execution of high actions from the view of low actions, the abstraction of high actions simply removes occurrences of high actions of a thread.

Definition 7.30 *The abstraction operator* $\tau_H : \mathrm{BTA}_\Sigma^\infty \to \mathrm{BTA}_\Sigma^\infty$ *is defined by* $\tau_H(P) = \tau_t(t_{\Sigma_H}(P))$.

7.4 Labeled transition systems over BTA

In this section, we define a *labeled transition system* over BTA, and two bisimulation equivalences to give noninterference properties in the next section.

7.4.1 Labeled transition systems

A **labeled transition system** (LTS) with **termination** S and **deadlock** D is a pair $(\mathbb{P}, \rightarrow)$ with $\mathbb{P}$ a class of threads, and $\rightarrow \subseteq \mathbb{P} \times (\Sigma \times \{T, F\}) \times \mathbb{P}$ a set of **transitions**. We write $P \xrightarrow{a,\kappa} Q$ with $a \in \Sigma$ and $\kappa \in \{T, F\}$ for $(P, (a, \kappa), Q) \in \rightarrow$. An LTS is a **finite-state** (regular) LTS if both $\mathbb{P}$ and Σ are finite.

For a state P, if $P = S$ then it is a termination state. If $P = D$ then it is a deadlock. If $P = P_1 \trianglelefteq a \trianglerighteq P_2$ then $P \xrightarrow{a,T} P_1$ and $P \xrightarrow{a,F} P_2$.

We note that since program behaviors in the programming language `Lang` can be represented as *regular* threads, they can always be associated with a finite-state LTS.

7.4.2 Bisimulation

Bisimulation equivalence classifies threads behaving identically, defined as follows.

Definition 7.31 *A* **bisimulation** *relation $\mathcal{B}$ is a symmetric binary relation on threads satisfying:*

1. *If $(P, Q) \in \mathcal{B}$ and $P = S$, then $Q = S$.*
2. *If $(P, Q) \in \mathcal{B}$ and $P = D$, then $Q = D$.*
3. *If $(P, Q) \in \mathcal{B}$ and $P \xrightarrow{a,\kappa} P'$ then there exists Q' such that $Q \xrightarrow{a,\kappa} Q'$ and $(P', Q') \in \mathcal{B}$.*

Two threads P and Q are **bisimilar**, *denoted by $(P \leftrightarrows Q)$, if there is a bisimulation relation $\mathcal{B}$ such that $(P, Q) \in \mathcal{B}$.*

Proposition 7.32 *Bisimulation is an equivalence.*

7.4.3 Bisimulation up to I

Let $I \subseteq \Sigma$ be a set of actions. The relation *bisimulation up to I* identifies threads behaving the same from the view of the actions which are not in I. In other words, this bisimilarity is obtained by ignoring the presence of actions of I. We will use the following notion to define bisimulation up to I.

Definition 7.33 *Let P be a thread. A thread Q is a* **residual** *thread of P, written as $P \Rightarrow Q$, if there is a path of transitions $P = P_0 \xrightarrow{a_0,\kappa_0} P_1 \xrightarrow{a_1,\kappa_1} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} P_n = Q$ with $n \geq 0$. We write $P \overset{I}{\Rightarrow} Q$ if $a_i \in I$ for all $1 \leq i \leq n$.*

Definition 7.34 *A* **bisimulation up to** *I is a symmetric binary relation $\mathcal{B}$ on threads satisfying:*

1. *If $(S, Q) \in \mathcal{B}$ then there exists a path $Q \overset{I}{\Rightarrow} S$.*

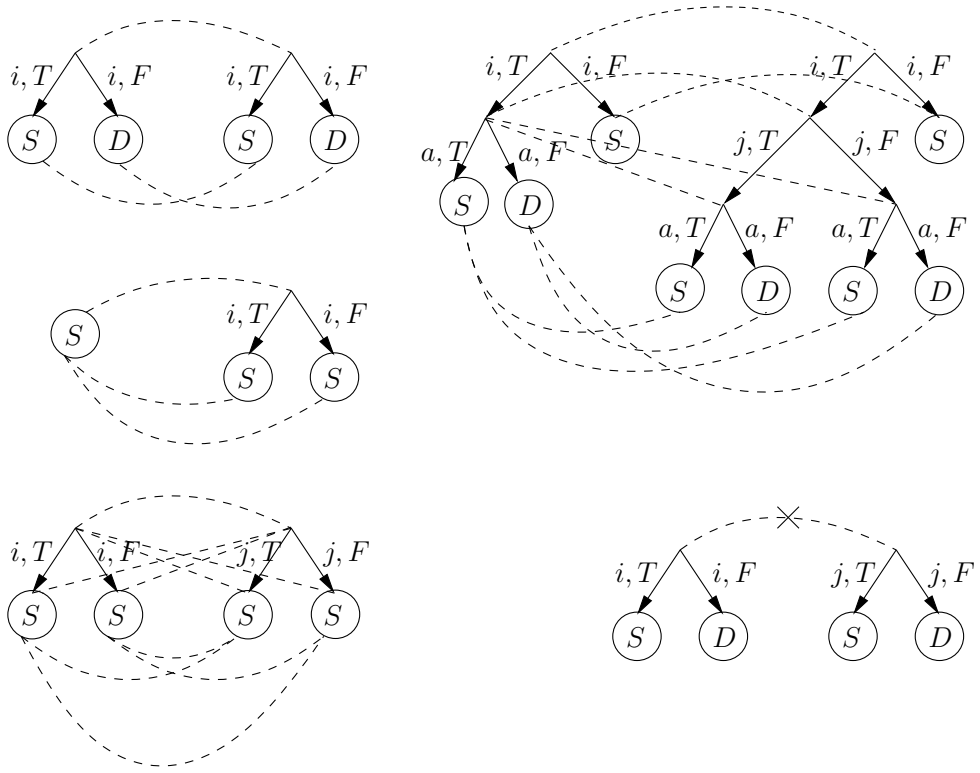

Figure 7.1: Examples of bisimulation with $I = \{i, j\}$ and $a \in \Sigma$. The dashed lines represent bisimulation up to I between threads.

2. *If* $(P, Q) \in \mathcal{B}$ *and* $P \xrightarrow{a,\kappa} P'$ *then either:*
   (a) $a \in I$ *and* $(P', Q) \in \mathcal{B}$*, or:*
   (b) *there exists a path* $Q \stackrel{I}{\Rightarrow} Q_1 \xrightarrow{a,\kappa} Q'$ *such that* $(P, Q_1) \in \mathcal{B}$ *and* $(P', Q') \in \mathcal{B}$.

Two threads P *and* Q *are* **bisimilar up to** I*, denoted by* $(P \leftrightarrows_I Q)$*, if there is a bisimulation up to* I *relation* $\mathcal{B}$ *such that* $(P, Q) \in \mathcal{B}$.

We note that in Clause 2(b) of the above definition, it is allowed that $a \in I$. Furthermore, in case $I = \emptyset$, bisimulation up to I coincides with bisimulation. Figure 7.1 illustrates the notion of bisimulation up to I between threads.

Proposition 7.35 *Bisimulation up to* I *is an equivalence.*

We omit the proof of the previous proposition, since it is similar to the proof for branching bisimulation [52] given in [12].

Proposition 7.36 *Let* $I \subseteq I' \subseteq \Sigma$*. Then* $\leftrightarrows_I \subseteq \leftrightarrows_{I'}$.

7.5 Noninterference based on behaviors

The approaches for checking the noninterference property based on type systems have been shown to be effective [88]. However, they cannot be applied to unstructured programming languages. Furthermore, these approaches are confusing when parallelism is introduced in the programming languages.

In this section, we define noninterference for program behaviors in the setting of thread algebra. We assume the existence of a set I of invisible actions that contains the set Σ_H of high actions ($\Sigma_H \subseteq I \subseteq \Sigma_I$). Following the existing notions from the literature, we propose *termination-insensitive noninterference up to* I ($\mathtt{TINI}_I$) and *termination-sensitive noninterference up to* I ($\mathtt{TSNI}_I$). Depending on certain features, I can be chosen to accept the most secure programs, while preserving other properties of threads. We also consider the worst-case scenario where an attacker may observe the timing of program execution, by providing the notion of *time-sensitive noninterference* (TISNI) for threads. We prove soundness for our definitions and show that we accept all secure programs that are typable in the type systems of [99, 91]. Our definitions can be applied to unstructured programming languages because they are based on program behaviors. Furthermore, they are also suitable for considering noninterference properties in multi-threaded languages since a multi-thread is also a single thread in thread algebra.

Let $\sigma(P)$ denote the set of actions occurring in a process P.

7.5.1 Termination-insensitive noninterference up to I

In this section, we present *termination-insensitive noninterference up to I* for threads. We consider the following programs.

```
X ::= l:=h
Y ::= IF h==1 THEN l:=0 ELSE l:=1 END IF
Z ::= IF h==1 THEN l:=0 ELSE l:=0 END IF
```

where $l \in \mathtt{Var}_L$ and $h \in \mathtt{Var}_H$ are low and high variables. Program X is insecure and is not accepted by the type systems of [99, 91] since the assignment $l{:=}h$ reveals the value of h. The insecure program Y is untypable in these type systems, too. It is because the type of the assigned variable l in the branches of the conditional statement is lower than the type of the condition $h{==}1$ (see Section 7.2.6). By the reason of the same fact, program Z is rejected by these type systems although it is secure. In our approach, we also reject X and Y, but accept Z. Because this program behaves the same after the execution of $\langle h{==}1\rangle$. Hence, an attacker cannot learn the value of h of program Z through branching on the condition $\langle h == 1\rangle$.

We then propose a notion of termination-insensitive noninterference for threads as follows. A thread is termination-insensitive noninterfering if its actions are secure. Furthermore, its behavior from the view of low actions, is always the same regardless of the returned boolean value after the execution of a high action. Formally:

Definition 7.37 *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. A thread P is* **termination-insensitive noninterfering up to** *I ($P \in \mathtt{TINI}_I$) if*

1. *all actions of P are secure, i.e. $\sigma(P) \subseteq \Sigma_S$,*

2. *for all residual threads Q of P such that $Q \xrightarrow{a,\kappa} Q'$ with $a \in \Sigma_H$, $Q \leftrightarrows_I Q'$.*

In the definition above, in case $I = \Sigma_I$, $\texttt{TINI}_I$ would accept the most secure programs, while in case $I = \Sigma_H$, $\texttt{TINI}_I$ would accept the least secure programs. For instance, consider the following example.

Example 7.38 Let U, V and W be programs given as follows.

```
U ::= IF h==1 THEN h:=h+1 ELSE h:=h-1 END IF.
V ::= l:=0;
      IF h==0 THEN
         IF l==1 THEN h:=1 ELSE h:=2 END IF;
      ELSE
         h:=3;
      END IF;
W ::= WHILE h<10 DO
            h:=h+1;
      END WHILE.
```

Program U and W are accepted in both cases $I = \Sigma_I$ and $I = \Sigma_H$. However, program V is only accepted in case $I = \Sigma_I$. This program is rejected by $\texttt{TINI}_{\Sigma_H}$ since it will proceed with the test $\langle l == 1\rangle \notin \Sigma_H$ if $h = 0$, while in the other case it will not.

Proposition 7.39 *Let $\Sigma_H \subseteq I \subseteq I' \subseteq \Sigma_I$. Then $\texttt{TINI}_I \subseteq \texttt{TINI}_{I'}$.*

Proof: This follows from Proposition 7.36. □

We now show that in case $I = \Sigma_I$, we accept all secure programs that are accepted by the type system of [99].

Theorem 7.40 *Let X be a program in the programming language* Lang*. If X is well-typed by the typing rules in Table 7.1 then $|X| \in \texttt{TINI}_{\Sigma_I}$.*

Proof: Since all actions in X are well-typed, they are secure. Let Q be a residual thread of X such that $Q \xrightarrow{a,\kappa} Q'$ with $a \in \Sigma_H$. By the typing rules of Table 7.1, if a is the condition of a conditional statement (or a while-loop) then all the assignments within the branches (or the body) of that statement are high. Hence there exists a residual thread P of Q satisfying the following property: for every path $Q = Q_0 \xrightarrow{a_0,\kappa_0} Q_1 \xrightarrow{a_1,\kappa_1} \cdots$ from Q, there exists $n \in \mathbb{N}$ such that $Q_n = P$ and $a_i \in \Sigma_I$ for all $i < n$. We define that $(Q_i, P) \in \mathcal{B}$ for all $i \leq n$. Then $\mathcal{B}$ is a bisimulation up to Σ_I. Hence $Q \leftrightarrows_{\Sigma_I} Q'$. By Definition 7.37, $|X| \in \texttt{TINI}_{\Sigma_I}$. □

It should be noticed that $\texttt{TINI}_I$ will accept *insecure* programs in certain cases as can be seen in Example 7.41.

Example 7.41 We consider the program below:

```
T ::= WHILE h>0 DO
            h:=h+1;
      END WHILE
```

where $h \in \mathtt{Var}_H$. It can be derived that $T \in \mathtt{TINI}_I$ for all sets $\Sigma_H \subseteq I \subseteq \Sigma_I$. However, $T \notin \mathtt{NI}$ since $|X| \bullet [\langle h, 0\rangle] = [\langle h, 0\rangle]$ while $|X| \bullet [\langle h, 1\rangle] = D$.

The program in Example 7.41 exemplifies a *termination-leak* insecure program. To preserve the noninterference property for $\mathtt{TINI}_I$ we impose a condition that *the program terminates successfully* as in [99], given as follows.

Definition 7.42 *A thread P* **terminates successfully** *if for all $s \in \mathbb{S}$, $P \bullet s \neq D$. Let BTA_Σ^T be the set of all threads that terminates successfully.*

Definition 7.42 implies that for a thread $P \in \mathrm{BTA}_\Sigma^T$ and a state $s \in \mathbb{S}$, there is a finite deterministic path $(P_0 = P, s_0 = s) \xrightarrow{a_0,\kappa_0} (P_1, s_1) \xrightarrow{a_1,\kappa_1} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} (P_n = S, s_n)$ satisfying that $P_i \xrightarrow{a_i,\kappa_i} P_{i+1}$, $\kappa_i = y_{a_i}(s_i)$ and $s_{i+1} = \mathtt{effect}_{a_i}(s_i)$ for all $0 \leq i < n$.

In the next theorem we show that if a thread is termination-insensitive noninterfering then it is secure under the condition that the thread terminates successfully.

Theorem 7.43 *(**Soundness of $\mathtt{TINI}_I$**). Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. Then $\mathtt{TINI}_I \cap \mathrm{BTA}_\Sigma^T \subseteq \mathtt{NI}$.*

Proof: Let $P \in \mathtt{TINI}_I \cap \mathrm{BTA}_\Sigma^T$. We show that for all low equivalent states $s, r \in \mathbb{S}$ ($s =_{\mathbb{S}}^L r$), $P \bullet s =_{\mathbb{S}}^L P \bullet r$. Since P terminates successfully, there are two finite deterministic paths obtained by the computations of P with s and r, given as in the following.

$$(P_0 = P, s_0 = s) \overset{I}{\Rightarrow} (P_0', s_0') \xrightarrow{a_0,\kappa_0} (P_1, s_1) \overset{I}{\Rightarrow} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} (P_n = S, s_n) \text{ and}$$
$$(Q_0 = P, r_0 = r) \overset{I}{\Rightarrow} (Q_0', r_0') \xrightarrow{b_0,\gamma_0} (Q_1, r_1) \overset{I}{\Rightarrow} \cdots \xrightarrow{b_{m-1},\gamma_{m-1}} (Q_m = S, r_m)$$

where $P_i \overset{I}{\Rightarrow} P_i'$, $P_i \leftrightarrows_I P_i'$ and $P_i' \not\leftrightarrows_I P_{i+1}$ for $0 \leq i < n$, and where $Q_j \overset{I}{\Rightarrow} Q_j'$, $Q_j \leftrightarrows_I Q_j'$ and $Q_j' \not\leftrightarrows_I Q_{j+1}$ for $0 \leq j < m$. We prove by induction on i that $P_i \leftrightarrows_I Q_i$ and $s_i =_{\mathbb{S}}^L r_i$ for all $0 \leq i \leq n$. We consider the following possibilities:

1. $i = 0$. Then $P_0 = Q_0 = P$. Thus, $P_0 \leftrightarrows_I Q_0$ and $s_0 = s =_{\mathbb{S}}^L r = r_0$.
2. Assume that $P_i \leftrightarrows_I Q_i$ and $s_i =_{\mathbb{S}}^L r_i$. We prove that $P_{i+1} \leftrightarrows_I Q_{i+1}$ and $s_{i+1} =_{\mathbb{S}}^L r_{i+1}$. One can derive that $P_i' \leftrightarrows_I Q_i'$. Moreover $a_i = b_i$ because of $P_i' \xrightarrow{a_i,\kappa_i} P_{i+1}$, $Q_i' \xrightarrow{b_i,\gamma_i} Q_{i+1}$, $P_i' \not\leftrightarrows_I P_{i+1}$ and $Q_i' \not\leftrightarrows_I Q_{i+1}$. Furthermore, by Definition 7.37, $a_i \notin H$. Since invisible actions do not have effect on the low space, $s_i' =_{\mathbb{S}}^L s_i =_{\mathbb{S}}^L r_i =_{\mathbb{S}}^L r_i'$. It follows from Lemma 7.23 and $s_i' =_{\mathbb{S}}^L r_i'$ that $\kappa_i = y_{a_i}(s_i') = y_{b_i}(r_i') = \gamma_i$. This implies that $P_{i+1} \leftrightarrows_I Q_{i+1}$ and $s_{i+1} = \mathtt{effect}_{a_i}(s_i') =_{\mathbb{S}}^L = \mathtt{effect}_{a_i}(r_i') = r_{i+1}$.

Hence $P_i \leftrightarrows_I Q_i$ and $s_i =^L_{\mathbb{S}} r_i$ for all $0 \leq i \leq n$. Since $P_n = Q_m = S$, $n = m$. Therefore, $P \bullet s =^L_{\mathbb{S}} s_n =^L_{\mathbb{S}} r_n =^L_{\mathbb{S}} P \bullet r$. By Definition 7.15, $P \in \mathtt{NI}$. □

Example 7.44 Consider the program:

```
X ::= IF h==1 THEN
              h:=h+1;
              l:=1;
      ELSE
              l:=1;
      END IF.
```

where $h \in \mathtt{Var}_H$ and $l \in \mathtt{Var}_L$. The behavior of X is determined by $|X| = ([h := h + 1] \circ [l{:=}1] \circ S) \trianglelefteq \langle h{==}1\rangle \trianglerighteq ([l{:=}1] \circ S)$. Let $I = \{\langle h{==}1\rangle, [h := h + 1]\}$. X is accepted by $\mathtt{TINI}_I$ since it behaves the same after the execution of the test $\langle h{==}1\rangle$ from the view of low actions. Note that X is rejected by the type system of [99].

7.5.2 Termination-sensitive noninterference up to I

In this section, we assume that an attacker can observe the termination of a program (termination leak). Hence the attacker may learn that the value of the high variable h in Example 7.41 is greater than 0 when the program does not terminate.

To circumvent this problem, we restrict the notion of termination-insensitive noninterference presented in the previous section to *termination-sensitive noninterference up to I* with the condition that threads cannot perform an infinite sequence of invisible actions from I. Formally:

Definition 7.45 *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. A thread P is* **termination-sensitive noninterfering up to** *I ($P \in \mathtt{TSNI}_I$) if*

1. *all actions of P are secure, i.e. $\sigma(P) \subseteq \Sigma_S$,*
2. *P cannot perform an infinite sequence of actions from I, i.e., P cannot have a residual thread Q such that $Q = Q_0 \xrightarrow{i_0,\kappa_0} Q_1 \xrightarrow{i_1,\kappa_1} \cdots$ with $i_j \in I$.*
3. *for all residual threads Q of P such that $Q \xrightarrow{a,\kappa} Q'$ with $a \in \Sigma_H$, $Q \leftrightarrows_I Q'$.*

Proposition 7.46 *$\mathtt{TSNI}_I \subseteq \mathtt{TINI}_I$ for all $\Sigma_H \subseteq I \subseteq \Sigma_I$.*

The previous definition of termination-sensitive noninterference seems strict. However, in case of structured programming languages, we accept all secure programs that are accepted by termination-sensitive noninterference treated by Smith and Volpano in [91] (in which loop guards are required to have type $\mathcal{L}$, i.e. they are not high actions). Here the set I is defined as the set of all high actions and all invisible actions occurring in a conditional statement with high guard.

Theorem 7.47 *Let X be a program in the programming language* `Lang` *such that X is well-typed by the typing rules in Table 7.1 with rule (WHILE) replaced by rule (WHILE-TSNI) in Table 7.2, and let I be the set of high actions together with all invisible actions occurring in a conditional statement with high guard. We assume that these actions are distinct from the remaining actions of X. Then $|X| \in \mathtt{TSNI}_I$.*

Proof: We only show that $|X|$ cannot perform an infinite sequence of actions from I. The rest of the proof is similar to the proof of Theorem 7.40. By rule (WHILE-TSNI) in Table 7.2, every while-loop of X has type $\mathcal{L}\,\mathtt{cmd}$. Since X is well-typed, it follows from rule (IF) in Table 7.1 that there is no while-loop occurring in a conditional statement with high guard. This implies that $|X|$ cannot perform an infinite sequence of actions from I. □

We note that unlike `TINI`, in case $I \subseteq I'$, $\mathtt{TSNI}_I \not\subseteq \mathtt{TSNI}_{I'}$ as can be seen in the following example.

Example 7.48 Let $\Sigma_H = I = \emptyset$, $I' = \{i\} \subseteq \Sigma_I$ and $P = i^\infty$. Then $P \in \mathtt{TSNI}_I$ while $P \notin \mathtt{TSNI}_{I'}$ since it can perform an infinite sequence of i-actions.

The condition that P cannot perform an infinite sequence of invisible actions from I suggests the following projective sequence of P based on the approximation operator π_n^I.

Definition 7.49 *The approximation operator $\pi_n^I : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ with $I \subseteq \Sigma$ is defined by*

$$\begin{array}{lcl}
\pi_0^I(P) & = & D, \\
\pi_{n+1}^I(S) & = & S, \\
\pi_{n+1}^I(D) & = & D, \\
\pi_{n+1}^I(P \trianglelefteq a \trianglerighteq Q) & = & \pi_{n+1}^I(P) \trianglelefteq a \trianglerighteq \pi_{n+1}^I(Q) \text{ with } a \in I, \\
\pi_{n+1}^I(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n^I(P) \trianglelefteq a \trianglerighteq \pi_n^I(Q) \text{ otherwise.}
\end{array}$$

A **projective sequence with respect to** *I is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$,*

$$\pi_n^I(P_{n+1}) = P_n.$$

For an infinite thread P represented by a projective sequence $(P_n)_n$ with respect to I, we write $\pi_n^I(P) = P_n$ for all $n \in \mathbb{N}$.

In order to show the soundness of $\mathtt{TSNI}_I$, we use the following auxiliary lemmas.

Lemma 7.50 *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and let P and Q be threads such that $P, Q \in \mathtt{TSNI}_I$ and $P \leftrightarrows_I Q$. Then $P \approx_{\mathbb{S}}^L Q$.*

Proof: We consider the following possibilities:

1. P and Q are finite threads. We prove this lemma by induction on the length of P. If $P \in \{S, D\}$ then this is trivial. Let $P = P_1 \trianglelefteq a \trianglerighteq P_2$. There are three cases:

   (a) $a \in \Sigma_H$. Since $P \in \texttt{TSNI}_I$, $P_1 \leftrightarrows_I P$ and $P_2 \leftrightarrows_I P$. Thus, $P_1 \leftrightarrows_I Q$ and $P_2 \leftrightarrows_I Q$. Then by the induction hypothesis, $P_1 \approx^L_{\mathbb{S}} Q$ and $P_2 \approx^L_{\mathbb{S}} Q$. Therefore, $P \approx^L_{\mathbb{S}} Q$.

   (b) $a \in I \setminus \Sigma_H$ and $P_1 \leftrightarrows_I Q$. It can be derived that $P_2 \leftrightarrows_I Q$. Then by the induction hypothesis, $P_1 \approx^L_{\mathbb{S}} Q$ and $P_2 \approx^L_{\mathbb{S}} Q$. Therefore, $P \approx^L_{\mathbb{S}} Q$.

   (c) There are finite paths $Q \overset{I}{\Rightarrow} Q' \xrightarrow{a,T} Q_1$ and $Q \overset{I}{\Rightarrow} Q' \xrightarrow{a,F} Q_2$ such that $P \leftrightarrows_I Q'$, $P_1 \leftrightarrows_I Q_1$ and $P_2 \leftrightarrows_I Q_2$. By the induction hypothesis, $P_1 \approx^L_{\mathbb{S}} Q_1$ and $P_2 \approx^L_{\mathbb{S}} Q_2$. Since $a \notin \Sigma_H$, $y_a(s) = y_a(t)$ for all states s, t of a state space $\mathbb{S}$ such that $s =^L_{\mathbb{S}} t$. This implies that $P \approx^L_{\mathbb{S}} Q'$. Since invisible actions have no effect on the low space, $Q \approx^L_{\mathbb{S}} Q'$. Therefore, $P \approx^L_{\mathbb{S}} Q$.

2. P and Q are infinite threads. Let $P_n = \pi^I_n(P)$ and $Q_n = \pi^I_n(Q)$ for all $n \in \mathbb{N}$. One can see that P_n and Q_n are finite, and $P_n \leftrightarrows_I Q_n$. It follows from the previous case that $P_n \approx^L_{\mathbb{S}} Q_n$. Therefore, $P \approx^L_{\mathbb{S}} Q$.

□

Lemma 7.51 *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$ and P be a thread that cannot perform an infinite sequence of invisible actions in I. Then $P \in \texttt{TSNI}_I \Leftrightarrow P \leftrightarrows_I \tau_H(P)$.*

Proof: Straightforward. □

The next theorem shows that if a thread is termination-sensitive noninterfering up to I then it is secure.

Theorem 7.52 *(***Soundness of $\texttt{TSNI}_I$***). Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. Then $\texttt{TSNI}_I \subseteq \texttt{NI}$.*

Proof: Let P be a thread such that $P \in \texttt{TSNI}_I$. It follows from Lemma 7.50 and Lemma 7.51 that $P \approx^L_{\mathbb{S}} \tau_H(P)$. Thus, $P \in \texttt{NI}$. □

7.5.3 Timing-sensitive noninterference

Finally, this section assumes the worst-case scenario, an attacker may observe the timing of program execution. Consider the following program:

```
X ::= WHILE h>0 DO
            h:=h-1;
      END WHILE.
```

This program is an example of a *timing leak* insecure program. Here, the attacker may learn the value of h from the timing behavior of the program. Timing leaks have been shown, for instance in [63], to leak sensitive information.

We assume the existence of the command `SKIP` in the programming language `Lang`. This command does nothing upon execution, but takes a single unit of time to perform and is represented by a special action, called `tick`. The action `tick` has no effect on the state space and always returns true after its executions.

We define another noninterference property called *timing-sensitive noninterference* for threads in order to avoid timing leaks. Following [87], we assume that the execution of each action takes a single unit of time. Our definition is based on the notion of bisimulation presented in Section 7.4. Informally speaking, a thread is timing-sensitive noninterfering if it behaves the same (in the sense that all high and tick actions are considered similarly) through branching on a high risk condition. Formally:

Definition 7.53 *A thread P is* **timing-sensitive noninterfering** *($P \in$ `TISNI`) if*

1. *all actions of P are secure, i.e. $\sigma(P) \subseteq \Sigma_S$,*
2. *for all residual threads Q of P such that $Q \xrightarrow{a,T} Q_1$ and $Q \xrightarrow{a,F} Q_2$ with $a \in \Sigma_H$, $t_{\Sigma_H \cup \{\texttt{tick}\}}(Q_1) \leftrightarrows t_{\Sigma_H \cup \{\texttt{tick}\}}(Q_2)$.*

According to the previous definition, the above program X is insecure. The following program is an example of a timing-sensitive secure program.

```
Y ::= IF h==1 THEN h:=h+1 ELSE SKIP END IF; l:=1.
```

One can derive that a timing-sensitive noninterfering thread is secure. Formally:

Theorem 7.54 *(**Soundness of TISNI**).* `TISNI` $\subseteq$ `NI`.

Proof: Straightforward. □

7.6 Compositionality of TISNI

In this section, we show that timing-sensitive noninterference satisfies compositionality, a fundamental property in programming languages, with respect to the cyclic interleaving operator.

Lemma 7.55 *(**Congruence**). Let P_i and Q_i ($1 \leq i \leq n$) be threads such that $P_i \leftrightarrows Q_i$. Then*

$$\|_{csi}\ (\langle P_1\rangle \curvearrowright \langle P_2\rangle \curvearrowright \cdots \curvearrowright \langle P_n\rangle) \leftrightarrows \|_{csi}\ (\langle Q_1\rangle \curvearrowright \langle Q_2\rangle \curvearrowright \cdots \curvearrowright \langle Q_n\rangle)$$

Proof: See Theorem 4.13. □

One can see that the pre-abstraction operator is closed under the cyclic interleaving operator, i.e.:

Lemma 7.56 *Let $\alpha = \alpha_1 \frown \cdots \frown \alpha_n$ be a thread vector. Then*

$$t_\Phi(\|_{csi}\ (\alpha)) = \|_{csi}\ (t_\Phi(\alpha_1) \frown t_\Phi(\alpha_2) \frown \cdots \frown t_\Phi(\alpha_n)).$$

It follows from Definition 7.53, Lemma 7.55 and Lemma 7.56 that timing-sensitive noninterference satisfies compositionality with respect to the cyclic interleaving operator.

Theorem 7.57 *(**Compositionality of TISNI**). Let $P_i \in \mathtt{TISNI}$ for all $1 \leq i \leq n$. Then*

$$\|_{csi}\ (\langle P_1 \rangle \frown \langle P_2 \rangle \frown \cdots \frown \langle P_n \rangle) \in \mathtt{TISNI}$$

7.7 An interleaving strategy with respect to noninterference

In the previous section, we have seen that timing-sensitive noninterference is compositional with respect to the cyclic interleaving operator. It would be natural if termination-insensitive noninterference and termination-sensitive noninterference also satisfy compositionality with respect to this operator. Unfortunately, it is shown in the following example that the compositionality property does not hold for the cases of $\mathtt{TINI}_I$ and $\mathtt{TSNI}_I$.

Example 7.58 Let $h \in \mathtt{Var}_H, l \in \mathtt{Var}_L$. Let α and β be two single threads defined as

$$\begin{array}{lcl} \alpha & = & (([h{:=}h+1] \circ [l{:=}0] \circ S) \trianglelefteq \langle h{==}1 \rangle \trianglerighteq ([l{:=}0] \circ S); \\ \beta & = & [l{:=}1] \circ [l{:=}2] \circ S. \end{array}$$

Let $I = \Sigma_H = \{[h{:=}h+1], \langle h{==}1 \rangle\}$. It can be checked that α and β are secure. However $\|_{csi}\ (\alpha \frown \beta)$ is not secure, since

$$\begin{array}{lcl} \|_{csi}\ (\alpha \frown \beta) = & & ([l{:=}1] \circ [h{:=}h+1] \circ [l{:=}2] \circ [l{:=}0] \circ S) \\ & & \trianglelefteq \langle h{==}1 \rangle \trianglerighteq \\ & & ([l{:=}1] \circ [l{:=}0] \circ [l{:=}2] \circ S) \end{array}$$

which produces $l = 0$ if $h = 1$, and $l = 2$ otherwise.

To preserve compositionality for $\mathtt{TSNI}_I$ and $\mathtt{TINI}_I$, we propose in this section a variant of the cyclic interleaving operator called the *cyclic strategic interleaving with persistence* operator ($\|_{csip}$) for thread algebra. This strategy is similar to the *current thread persistence* operator of [23] and will not invoke the rotation of a thread vector if the current action is *persistent*. We will show that bisimulation up to I is a congruence under this interleaving strategy with the assumption that all invisible actions of I are persistent. Later, in Section 7.8, it will be shown that $\mathtt{TINI}_I$ and $\mathtt{TSNI}_I$ satisfy compositionality with respect to the cyclic strategic interleaving with persistence operator.

7.7.1 The cyclic strategic interleaving with persistence operator

Let $\texttt{persistent} : \Sigma \to \{\texttt{true}, \texttt{false}\}$ be a boolean function defined on the set of actions Σ. Cyclic strategic interleaving with persistence $\|_{csip}$ is defined formally as follows.

Definition 7.59 *The axioms for the* **cyclic strategic interleaving with persistence** $\|_{csip}$ *operator are given for finite threads by*

$$\begin{array}{lcl}
\|_{csip}\ (\langle\rangle) & = & S \\
\|_{csip}\ (\langle S\rangle \frown \alpha) & = & \|_{csip}\ (\alpha) \\
\|_{csip}\ (\langle D\rangle \frown \alpha) & = & S_D(\|_{csip}\ (\alpha)) \\
\|_{csip}\ (\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) & = & \|_{csip}\ (\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \|_{csip}\ (\langle y\rangle \frown \alpha)\ \textit{if}\ \texttt{persistent}(a) \\
\|_{csip}\ (\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) & = & \|_{csip}\ (\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csip}\ (\alpha \frown \langle y\rangle)\ \textit{otherwise.}
\end{array}$$

For a thread vector α of arbitrary (finite or infinite) threads $\alpha = \alpha_1 \frown \cdots \frown \alpha_n$, $\|_{csip}\ (\alpha)$ is determined by its projective sequence:

$$\pi_n(\|_{csip}\ (\alpha)) = \pi_n(\|_{csip}\ (\pi_n(\alpha_1) \frown \cdots \frown \pi_n(\alpha_n))).$$

In order to preserve compositionality of $\texttt{TINI}_I$ and $\texttt{TSNI}_I$ with respect to cyclic interleaving with persistence, we define that for all invisible actions $a \in I$, $\texttt{persistent}(a) = \texttt{true}$. This assumption is to maintain the order of the invisible actions of I in the execution of a multi-thread, and therefore, the analysis can be made compositional.

Example 7.60 We now return to Example 7.58. We define that $\texttt{persistent}([h{:=}h{+}1]) = \texttt{persistent}([\langle h{==}1\rangle) = \texttt{true}$ and `persistent` yields false for all other actions. The multi-thread $\|_{csip}\ (\alpha \frown \beta)$ is secure, since

$$\begin{array}{lll}
\|_{csip}\ (\alpha \frown \beta) = & ([h{:=}h+1] \circ [l{:=}0] \circ [l{:=}1] \circ [l{:=}2] \circ S) \\
& \trianglelefteq \langle h{==}1\rangle \trianglerighteq \\
& ([l{:=}0] \circ [l{:=}1] \circ [l{:=}2] \circ S)
\end{array}$$

which always produces $l{=}2$.

We note that in the case that all actions are not persistent, the cyclic interleaving with persistence operator coincides with the cyclic interleaving operator ($\|_{csip}{=}\|_{csi}$).

The cyclic interleaving with persistence strategy with the assumption that all invisible actions of I are persistent may appear conservative in handing over the control to another thread in some cases. For instance, consider the following example.

Example 7.61 Let $P = \|_{csip}\ (\langle \texttt{tick}^\infty\rangle \frown a^\infty)$ and $I = \{\texttt{tick}\}$. According to our approach, $\texttt{persistent}(\texttt{tick}) = \texttt{true}$. By Definition 7.59, P will never have a chance to perform action a, since $P = \texttt{tick}^\infty$, which is considered disadvantagous for multi-threading.

To circumvent this issue, we could restrict I to the set Σ_H of high actions. This restriction would accept less secure threads as described in Section 7.5. However, it would increase the chance to hand over the control of multi-threading to another thread. We return to Example 7.61. In this example, the set Σ_H of high actions is empty. Let $I = \Sigma_H = \emptyset$. The multi-thread P now is determined by $P = \texttt{tick} \circ a \circ \texttt{tick} \circ a \cdots$.

7.7.2 Congruence with respect to $\texttt{TSNI}_I$

In this section, we show that bisimulation up to I is a congruence under the cyclic interleaving with persistence operator, with respect to the assumptions that a thread cannot perform an infinite sequence of invisible actions from I, and all invisible actions of I are persistent. We will use the following auxiliary lemma.

Lemma 7.62 *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and let P and Q be threads such that $P \leftrightarrows_I Q$. Then $S_D(P) \leftrightarrows_I S_D(Q)$.*

Lemma 7.63 *(**Congruence with respect to** $\texttt{TSNI}_I$). Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and $\texttt{persistent}(a) = \texttt{true}$ for all $a \in I$. Let P_i and Q_i $(1 \leq i \leq n)$ be threads such that $P_i \leftrightarrows_I Q_i$, and P_i and Q_i cannot perform an infinite sequence of actions from I. Then*

$$\|_{csip}\ (\langle P_1\rangle \frown \langle P_2\rangle \frown \cdots \frown \langle P_n\rangle) \leftrightarrows_I \|_{csip}\ (\langle Q_1\rangle \frown \langle Q_2\rangle \frown \cdots \frown \langle Q_n\rangle)$$

Proof: Let $\mathcal{B}$ be a binary relation defined as follows. For threads P and Q, $(P, Q) \in \mathcal{B}$ if there are sequences α and β with the same length n for some $n \in \mathbb{N}$ such that $P = \|_{csip}\ (\alpha)$, $Q = \|_{csip}\ (\beta)$, and for all components α_i and β_i of α and β, $\alpha_i \leftrightarrows_I \beta_i$ respectively. We show that $\mathcal{B}$ is a bisimulation up to I.

1. $P = S$. Then for all i, $1 \leq i \leq n$, $\alpha_i = S$. Since, $\beta_i \leftrightarrows_I \alpha_i$. It is not hard to see that there exists a finite path $Q \stackrel{I}{\Rightarrow} Q'$ with $Q' = S$.

2. $P \xrightarrow{a,\kappa} P'$. There exists i such that for all $j < i$, $\alpha_j \in \{S, D\}$ and $\alpha_i \xrightarrow{a,\kappa} x$. Let $P_1 = \|_{csip}\ (\alpha_i \frown \cdots \frown \alpha_n)$ and $P_1' = \|_{csip}\ (x \frown \alpha_{i+1} \frown \cdots \frown \alpha_n)$. Since for all $j < i$, $\alpha_j \leftrightarrows_I \beta_j$ and β_j cannot perform an infinite sequence of invisible actions, there exist finite paths $\beta_j \stackrel{I}{\Rightarrow} \beta_j'$ with $\beta_j' = \alpha_j$. Since $\alpha_i \leftrightarrows_I \beta_i$, there exists a finite path $\beta_i \stackrel{I}{\Rightarrow} \beta_i' \xrightarrow{a,\kappa} y$ with $\alpha_i \leftrightarrows_I \beta_i'$ and $x \leftrightarrows_I y$. Let $Q_1 = \|_{csip}\ (\beta_i' \frown \cdots \frown \beta_n)$ and $Q' = \|_{csip}\ (y \frown \beta_{i+1} \frown \cdots \frown \beta_n)$. Then $P_1 \leftrightarrows_I Q_1$ and $P_1' \leftrightarrows_I Q'$. If $\alpha_j = S$ for all $j < i$ then $P = P_1$, $P' = P_1'$ and $Q \stackrel{I}{\Rightarrow} Q_1 \xrightarrow{a,\kappa} Q'$ with $P \leftrightarrows_I Q_1$ and $P' \leftrightarrows_I Q'$, otherwise $P = S_D(P_1)$, $P' = S_D(P_1')$ and $Q \stackrel{I}{\Rightarrow} S_D(Q_1) \xrightarrow{a,\kappa} S_D(Q')$ with $P \leftrightarrows_I S_D(Q_1)$ and $P' \leftrightarrows_I S_D(Q')$ (because of Lemma 7.62).

Thus, $\mathcal{B}$ is a bisimulation up to I. □

One may expect that Lemma 7.63 also works on the case that threads can perform an infinite sequence of invisible actions from I. However, the following example shows that it is not the case.

Example 7.64 Let $P =\|_{csip} (\langle i^\infty \rangle \frown \langle a \circ S \rangle) = i^\infty$ and $Q =\|_{csip} (\langle D \rangle \frown \langle a \circ S \rangle) = a \circ D$ with $I = \{i\}$. It is obvious that P and Q are not bisimilar up to I although $i^\infty \leftrightarrows_I D$ and $a \circ S \leftrightarrows_I a \circ S$.

7.7.3 Congruence with respect to $\mathtt{TINI}_I$

We now show that Lemma 7.63 will hold for the case that a thread may perform an infinite sequence of invisible actions from I if all threads are deadlock-free.

Definition 7.65 *A thread is* **deadlock-free** *if it does not contain a residual deadlock D.*

Lemma 7.66 *(***Congruence with respect to** $\mathtt{TINI}_I$*).* *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and* $\mathtt{persistent}(a) = \mathtt{true}$ *for all $a \in I$. Let P_i and Q_i ($1 \leq i \leq n$) be deadlock-free threads such that $P_i \leftrightarrows_I Q_i$. Then*

$$\|_{csip} (\langle P_1 \rangle \frown \langle P_2 \rangle \frown \cdots \frown \langle P_n \rangle) \leftrightarrows_I \|_{csip} (\langle Q_1 \rangle \frown \langle Q_2 \rangle \frown \cdots \frown \langle Q_n \rangle)$$

Proof: Let $\mathcal{B}$ be a binary relation defined as follows. For threads P and Q, $(P, Q) \in \mathcal{B}$ if there are sequences α and β of the same length n for some $n \in \mathbb{N}$ such that $P =\|_{csip} (\alpha)$, $Q =\|_{csip} (\beta)$, and for all process components α_i and β_i of α and β, $\alpha_i \leftrightarrows_I \beta_i$ respectively. We show that $\mathcal{B}$ is a bisimulation up to I.

1. $P = S$. Then for all i, $1 \leq i \leq n$, $\alpha_i = S$. Since, $\beta_i \leftrightarrows_I \alpha_i$. It is not hard to see that there exists a finite path $Q \stackrel{I}{\Rightarrow} Q'$ with $Q' = S$.

2. $P \stackrel{a,\kappa}{\longrightarrow} P'$. Since $\alpha_j \neq D$ for all $1 \leq j \leq n$, there exists i such that for all $j < i$, $\alpha_j = S$ and $\alpha_i \stackrel{a,\kappa}{\longrightarrow} x$. Furthermore, $P =\|_{csip} (\alpha_i \frown \cdots \frown \alpha_n)$ and $P' =\|_{csip} (x \frown \alpha_{i+1} \frown \cdots \frown \alpha_n)$. Since for all $j < i$, $\alpha_j \leftrightarrows_I \beta_j$, there exist finite paths $\beta_j \stackrel{I}{\Rightarrow} S$. Since $\alpha_i \leftrightarrows_I \beta_i$, there exists a finite path $\beta_i \stackrel{I}{\Rightarrow} \beta_i' \stackrel{a,\kappa}{\longrightarrow} y$ with $\alpha_i \leftrightarrows_I \beta_i'$ and $x \leftrightarrows_I y$. Let $Q_1 =\|_{csip} (\beta_i' \frown \cdots \frown \beta_n)$ and $Q' =\|_{csip} (y \frown \beta_{i+1} \frown \cdots \frown \beta_n)$. Then $Q \stackrel{I}{\Rightarrow} Q_1 \stackrel{a,\kappa}{\longrightarrow} Q'$ with $P \leftrightarrows_I Q_1$ and $P' \leftrightarrows_I Q'$.

Thus, $\mathcal{B}$ is a bisimulation up to I. □

Example 7.64 is also a counter example of Lemma 7.66 for the case that threads contain deadlocks.

7.8 Compositionality of $\texttt{TSNI}_I$ and $\texttt{TINI}_I$

Assuming that invisible actions in the set $I \supseteq \Sigma_H$ are persistent and threads cannot perform an infinite sequence of invisible actions from I, we show that the abstraction of high actions is closed under the cyclic strategic interleaving with persistence. This implies that termination-sensitive noninterference satisfies compositionality with respect to this strategy. Furthermore, if we leave out the condition that threads cannot perform an infinite sequence of invisible actions from I, then for threads containing no deadlocks, termination-insensitive noninterference also satisfies compositionality with respect to the cyclic interleaving with persistence strategy.

7.8.1 Closure of abstraction of high actions

In order to show that the abstraction of high actions is closed under the cyclic interleaving with persistence operator, provided that all high actions are persistent, we use the following auxiliary lemma.

Lemma 7.67 *Let P be a finite thread. Let $\pi_m^{\Sigma_H} : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ be an approximation operator as defined in Definition 7.49 with I replaced by Σ_H. Then for all $m \in \mathbb{N}$,*

$$\tau_H(\pi_m^{\Sigma_H}(P)) = \pi_m(\tau_H(P)).$$

Theorem 7.68 *(***Closure of abstraction***). Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and $\texttt{persistent}(a) = \texttt{true}$ for all $a \in I$. Let $\alpha = \alpha_1 \frown \cdots \frown \alpha_n$ be a thread vector such that for $1 \leq i \leq n$, α_i cannot perform an infinite sequence of invisible actions of I. Then*

$$\tau_H(\|_{csip}\ (\alpha)) = \|_{csip}\ (\tau_H(\alpha_1) \frown \cdots \frown \tau_H(\alpha_n))$$

Proof: We consider two possibilities:

1. The thread α is finite. This implies that the threads α_i are finite for all $1 \leq i$. For a thread vector β of length n, we define that $\tau_H(\beta) = \tau_H(\beta_1) \frown \cdots \frown \tau_H(\beta_n)$. We prove this lemma by induction on the length of the threads and on the length of the thread vector. More precisely, we suppose that for all sequences β with $\texttt{length}(\beta) < n$, $\tau_H(\|_{csip}\ (\beta)) = \|_{csip}\ (\tau_H(\beta))$. Furthermore, for all threads Q with $\texttt{length}(Q) < \texttt{length}(P)$: $\tau_H(\|_{csip}\ (\langle Q \rangle \frown \beta)) = \|_{csip}\ (\langle \tau_H(Q) \rangle \frown \tau_H(\beta))$. We prove that $\tau_H(\|_{csip}\ (\langle P \rangle \frown \beta)) = \|_{csip}\ (\langle \tau_H(P) \rangle \frown \tau_H(\beta))$.

   (a) $P = S$. Then

   $$\tau_H(\|_{csip}\ (\langle S \rangle \frown \beta)) = \tau_H(\|_{csip}\ (\beta)) = \|_{csip}\ (\tau_H(\beta)) = \|_{csip}\ (\langle S \rangle \frown \tau_H(\beta)).$$

(b) $P = D$. Then

$$\begin{aligned}\tau_H(\|_{csip}\ (\langle D\rangle \curvearrowright \beta)) &= \tau_H(S_D(\|_{csip}\ (\beta))) = S_D(\|_{csip}\ (\tau_H(\beta)))\\ &= \|_{csip}\ (\langle D\rangle \curvearrowright \tau_H(\beta)).\end{aligned}$$

(c) $P = x \trianglelefteq a \trianglerighteq y$ with $a \in \Sigma_H$. Then

$$\|_{csip}\ (\langle P\rangle \curvearrowright \beta) = \|_{csip}\ (\langle x\rangle \curvearrowright \beta) \trianglelefteq a \trianglerighteq \|_{csip}\ (\langle y\rangle \curvearrowright \beta).$$

By the induction hypothesis,

$$\begin{aligned}\tau_H(\|_{csip}\ (\langle P\rangle \curvearrowright \beta)) &= \tau_H(\|_{csip}\ (\langle x\rangle \curvearrowright \beta))\\ &= \|_{csip}\ (\langle \tau_H(x)\rangle \curvearrowright \tau_H(\beta))\\ &= \|_{csip}\ (\langle \tau_H(P)\rangle \curvearrowright \tau_H(\beta)).\end{aligned}$$

(d) $P = x \trianglelefteq a \trianglerighteq y$ with $a \in I \setminus \Sigma_H$. Similar to the previous case,

$$\begin{aligned}&\tau_H(\|_{csip}\ (\langle P\rangle \curvearrowright \beta))\\ &\quad= \tau_H(\|_{csip}\ (\langle x\rangle \curvearrowright \beta)) \trianglelefteq a \trianglerighteq \tau_H(\|_{csip}\ (\langle y\rangle \curvearrowright \beta))\\ &\quad= \|_{csip}\ (\langle \tau_H(x)\rangle \curvearrowright \tau_H(\beta)) \trianglelefteq a \trianglerighteq \|_{csip}\ (\langle \tau_H(y)\rangle \curvearrowright \tau_H(\beta))\\ &\quad= \|_{csip}\ (\langle \tau_H(P)\rangle \curvearrowright \tau_H(\beta)).\end{aligned}$$

(e) $P = x \trianglelefteq a \trianglerighteq y$ with $a \notin I$. Then

$$\|_{csip}\ (\langle P\rangle \curvearrowright \beta) = \|_{csip}\ (\beta \curvearrowright \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{csip}\ (\beta \curvearrowright \langle y\rangle).$$

By the induction hypothesis,

$$\begin{aligned}&\tau_H(\|_{csip}\ (\langle P\rangle \curvearrowright \beta))\\ &\quad= \tau_H(\|_{csip}\ (\beta \curvearrowright \langle x\rangle)) \trianglelefteq a \trianglerighteq \tau_H(\|_{csip}\ (\beta \curvearrowright \langle y\rangle))\\ &\quad= \|_{csip}\ (\tau_H(\beta) \curvearrowright \langle \tau_H(x)\rangle) \trianglelefteq a \trianglerighteq \|_{csip}\ (\tau_H(\beta) \curvearrowright \langle \tau_H(y)\rangle)\\ &\quad= \|_{csip}\ (\langle \tau_H(P)\rangle \curvearrowright \tau_H(\beta)).\end{aligned}$$

2. The thread α is infinite. Let $P = \|_{csip}\ (\alpha)$ and

$$P_m = \pi_m^{\Sigma_H}(\|_{csip}\ (\pi_m^{\Sigma_H}(\alpha_1) \curvearrowright \pi_m^{\Sigma_H}(\alpha_2) \curvearrowright \cdots \curvearrowright \pi_m^{\Sigma_H}(\alpha_n))).$$

One can derive that $P = \bigsqcup_m P_m$. It follows from Lemma 7.67 and the previous case that

$$\begin{aligned}\tau_H(P_m) &= \pi_m(\tau_H(\|_{csip}\ (\pi_m^{\Sigma_H}(\alpha_1) \curvearrowright \pi_m^{\Sigma_H}(\alpha_2) \curvearrowright \cdots \curvearrowright \pi_m^{\Sigma_H}(\alpha_n))))\\ &= \pi_m(\|_{csip}\ (\tau_H(\pi_m^{\Sigma_H}(\alpha_1)) \curvearrowright \tau_H(\pi_m^{\Sigma_H}(\alpha_2)) \curvearrowright \cdots \curvearrowright \tau_H(\pi_m^{\Sigma_H}(\alpha_n))))\\ &= \pi_m(\|_{csip}\ (\pi_m(\tau_H(\alpha_1)) \curvearrowright \pi_m(\tau_H(\alpha_2)) \curvearrowright \cdots \curvearrowright \pi_m(\tau_H(\alpha_n))))\\ &= \pi_m(\|_{csip}\ (\tau_H(\alpha_1) \curvearrowright \tau_H(\alpha_2) \curvearrowright \cdots \curvearrowright \tau_H(\alpha_n)))\end{aligned}$$

This implies that $P = \|_{csip}\ (\tau_H(\alpha_1) \curvearrowright \tau_H(\alpha_2) \curvearrowright \cdots \curvearrowright \tau_H(\alpha_n))$.

□

7.8.2 Compositionality of termination-sensitive noninterference

In this section, we discuss the compositionality property of termination-sensitive noninterference. By assuming that all invisible actions of I are persistent, we show that $\texttt{TSNI}_I$ satisfies compositionality with respect to the cyclic interleaving with persistence strategy.

Theorem 7.69 *(**Compositionality of** $\texttt{TSNI}_I$**).* *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$ and for all actions $a \in I$, $\texttt{persistent}(a) = \texttt{true}$. Let $P_i \in \texttt{TSNI}_I$ for all $1 \leq i \leq n$. Then*

$$\|_{csip}\ (\langle P_1\rangle \frown \langle P_2\rangle \frown \cdots \frown \langle P_n\rangle) \in \texttt{TSNI}_I$$

Proof: This follows from Lemma 7.51, and Lemma 7.63 and Theorem 7.68. □

We note that without the assumption that all actions in I are persistent, the compositionality property for $\texttt{TSNI}_I$ does not hold as can be seen in Example 7.58. Furthermore, upon the execution of the thread vector, the current thread will switch to another thread since it cannot an infinite sequence of actions from I. Hence, every thread P_i in the thread vector will have a chance to perform its actions.

7.8.3 Compositionality of termination-insensitive noninterference

Similar to the previous section, we show that $\texttt{TINI}_I$ satisfies compositionality with respect to the cyclic interleaving with persistence, provided that all invisible actions of I are persistent, and that threads contain no deadlocks.

Theorem 7.70 *(**Compositionality of** $\texttt{TINI}_I$**).* *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$ and for all actions $a \in I$, $\texttt{persistent}(a) = \texttt{true}$. Let $P_i \in \texttt{TINI}_I$ be deadlock-free threads for all $1 \leq i \leq n$. Then*

$$\|_{csip}\ (\langle P_1\rangle \frown \langle P_2\rangle \frown \cdots \frown \langle P_n\rangle) \in \texttt{TINI}_I$$

Proof: Let $P = \|_{csip}\ (\langle P_1\rangle \frown \langle P_2\rangle \frown \cdots \frown \langle P_n\rangle)$. We show that for all residual threads Q of P, if $Q = Q' \trianglelefteq a \trianglerighteq Q''$ with $a \in \Sigma_H$ then $Q' \leftrightarrows_I Q''$. It can be derived that $Q = S$ or $Q = \|_{csip}\ (\langle Q_{i_1}\rangle \frown \langle Q_{i_2}\rangle \frown \cdots \frown \langle Q_{i_m}\rangle)$, where Q_k is a residual thread of P_k for $1 \leq k \leq n$. Let $Q_{i_1} = Q'_{i_1} \trianglelefteq a \trianglerighteq Q''_{i_1}$. Then

$$\begin{array}{lcl} Q' & = & \|_{csip}\ (\langle Q'_{i_1}\rangle \frown \langle Q_{i_2}\rangle \frown \cdots \frown \langle Q_{i_m}\rangle) \\ Q'' & = & \|_{csip}\ (\langle Q''_{i_1}\rangle \frown \langle Q_{i_2}\rangle \frown \cdots \frown \langle Q_{i_m}\rangle). \end{array}$$

Since $P_{i_1} \in \texttt{TINI}_I$, $Q'_{i_1} \leftrightarrows_I Q''_{i_1}$. It follows from Lemma 7.66 that $Q' \leftrightarrows_I Q''$. Therefore, $P \in \texttt{TINI}_I$. □

7.9 Concluding remarks

In this chapter, we have modelled various notions of noninterference in thread algebra. In particular, we have discussed termination-insensitive and termination-sensitive noninterference up to a set I which contains all high actions. These notions are based on bisimulation up to I. The former requires the condition that the program terminates successfully while the latter considers termination-leak information. Furthermore, we have defined timing-sensitive noninterference regarding timing-leak information. We have proven soundness of these noninterference properties and shown that we accept all secure programs that are accepted by the other approaches based on type systems discussed in [99, 91]. Hence, thread algebra is suitable as a process-algebraic framework for formalizing and analyzing security properties in multi-threaded languages. Furthermore, it is also an applicable framework for considering security properties for unstructured programs.

In the setting of multi-threading, it has been shown that timing-sensitive noninterference satisfies compositionality with respect to the cyclic interleaving strategy [23]. In order to preserve compositionality for termination-insensitive and termination-sensitive noninterference up to I, we have proposed the cyclic interleaving with persistence operator, and shown that these notions of noninterference satisfy compositionality, under the assumption that all actions of I are persistent.

For checking the noninterference properties presented in this chapter, one can apply existing techniques [78, 54] and existing tools [48, 49, 31] for checking process-equivalence to develop our security checkers.

In this chapter, we only cover a basic interleaving strategy namely the *cyclic interleaving* operator of [23] to illustrate our ideas. We have not considered plausible interleaving strategies dealing with other features of multi-threading such as forking and blocking in thread algebra. We leave these issues for future work. We hereby show that previous work on security for sequential and multi-threaded systems can be reconsidered in thread algebra.

Chapter 8

Goto elimination in program algebra

8.1 Introduction

Program algebra (PGA) is an algebraic framework for formalizing and analyzing sequential programming languages [22]. The virtue of PGA is that it is simple and easily memorized. Based on PGA, more complex programming languages such as Cobol, Java and C# can be developed and studied. A steady development of the core theory of PGA has been created and results on Maurer computers [26] and risk assessment [28] were achieved. This chapter discusses the topic of *goto removal* [45] in PGA.

Although goto removal has been studied for several decades, it is still important because of maintenance and redevelopment of legacy software systems. These systems have been growing and evolving over the years. Changing the code of legacy software systems is needed to keep up with requirements of real-life applications and technical evolution [32, 58, 70]. It requires a lot of time and effort for a programmer to maintain and modify unstructured source codes. Goto removal to aid the restructuring of the source code is a basic step in extracting business knowledge embedded in legacy applications [92]. Once the business logic from a legacy system has been extracted, the system is ready for modifications and integrations, for instance porting to Object Cobol [42]. Another example is to create web services from legacy host programs [93].

We distinguish two classes of approaches on goto removal. The first work considers gotos as a harmful statement [45]. It eliminates all gotos from programs by providing additional boolean variables, or by introducing a variety of loops with multi-level exits as studied in modern programming languages such as Java and C#. The latter is to remove certain types of gotos in order to restructure programs for modifiability and maintainability. This work has been studied exhaustively for legacy programming languages such as Cobol [90, 96]. In the following, we will briefly introduce these two classes.

8.1.1 Removing all gotos

Removing all gotos is involved with various notions of equivalence.

The most basic equivalence is *input-output equivalence* [67]. Two programs are input-output equivalent if given identical inputs, they produce identical outputs. Programs with gotos are considered as *flowcharts*. Goto removal with the use of additional variables is known as the *Folk theorem* (for an overview, see [55]) which states that every flowchart is equivalent to a while-program with only one occurrence of a while-loop under input-output equivalence, provided that additional variables are allowed. Many algorithms and transformation rules [33, 41, 76, 3, 39, 72] including that of Böhm and Jacopini [33] and Cooper [41] have been provided to prove this theorem. However, their correctness has not been discussed formally. Furthermore, goto removal described in the Folk theorem is treated under input-output equivalence, which is rather weak.

Goto removal is more complex under *path equivalence* [10]. Two flowcharts are path equivalent if they execute the same sequence of actions and tests. It is shown in [62, 3, 80, 61, 64] that conventional iterative constructs are not sufficient to replace gotos under this equivalence. To overcome this issue, Peterson et al. [80] introduce loops and multi-level exits, and show that a flowchart is path equivalent to a program written in this extension.

In [83], Ramshaw considers a stricter notion of *flow-graph* equivalence. Two programs are flow-graph equivalent if their flowcharts are the same. He provides a condition of *reducibility* to eliminate gotos under this equivalence. In particular, he shows that a program can be converted to another program which is free of *head-to-head crossings* under flow-graph equivalence if its flowchart is reducible. Furthermore, he defines the highest standard of equivalence, *structural* equivalence. Two programs are structurally equivalent if they are flow-graph equivalent, and we can convert the text of the source into the text of the target simply by replacing some components of the source without rearranging or altering any other statements. He then introduces two simple rules to remove all gotos and labels under structural equivalence: the *Forward Elimination Rule* and the *Backward Elimination Rule* for programs which are free of head-to-head crossings.

Both methods of Peterson et al. and Ramshaw aim to achieve structural equivalence. Therefore, they require strong conditions such as reducibility or being free of head-to-head crossings. In addition, the fact that path and flow-graph equivalence are defined for flowcharts, while structural equivalence is defined for program sources, causes inhomogeneity in reasoning about equivalences of programs. We will pick up where Peterson et al. and Ramshaw left off. Furthermore, we show that PGA provides a mathematical and systematic framework for reasoning about correctness and equivalence of goto removal with and without the use of additional variables.

8.1.2 Removing gotos for knowledge extraction

Removing all goto statements is not always a good solution in maintenance and redevelopment of legacy systems because the resulting programs cannot be expected

more transparent than the original ones [45]. In [90, 96], Sellink et al. and Veerman provide a collection of transformation rules using the ASF+SDF Meta-Environment [60, 37], which removes only certain gotos for restructuring programs in Cobol. These transformation rules have been shown to be very effective and were applied to several large industrial Cobol systems. However, no formal correctness proof for these rules is available. This is because of a lack of time and the several semantics defined for Cobol [97]. In particular, the semantics of the PERFORM statement in Cobol differs between Cobol dialects [98]. A lightweight approach to check correctness of these transformation rules has been proposed in [97]. But this is rather a circumstantial evidence for transformation correctness than a formal proof. We will define a restriction on Cobol that avoids the unexpected behaviors as studied in [98] and discuss correctness of these transformation rules in the setting of PGA. We also hint at an automatable method to prove correctness for most of them.

8.1.3 Our contributions and outline of the chapter

In this chapter, we show that:

1. Gotos can be eliminated in the setting of PGA by the use of additional boolean variables, or by introducing loops with multi-level exits. We formulate the algorithm of Cooper [41] for goto removal using additional boolean variables in the setting of PGA. Furthermore, we propose a technique to get rid of head-to-head crossings in order to subsequently employ the results of Peterson et al. and Ramshaw [80, 83] for goto removal without the use of additional variables. These are useful for the study of the program algebra itself.

2. PGA provides a mathematical and systematic framework for reasoning about and classifying the correctness and equivalence of various standard algorithms and transformation rules in goto removal. We show that the algorithm of Cooper is correct under *behavioral equivalence with respect to additional boolean variables*. This equivalence is finer than input-output equivalence in the approaches of [33, 41]. Furthermore, our algorithm for goto removal without the use of additional variables is correct under *behavioral equivalence*, an analogous notion of path equivalence defined for programs in PGA. We hereby explain goto removal with mathematical rigor. We also prove correctness for some industrial transformation rules [96] in restructuring Cobol programs.

The structure of this chapter is as follows. Section 8.2 recalls the basic concepts of PGA. Section 8.3 defines behaviors for programs containing gotos and structured programs. Section 8.4 eliminates gotos using additional variables. Section 8.5 eliminates gotos without using additional variables. Section 8.6 discusses correctness of transformation rules in [96]. The chapter ends with some concluding remarks in Section 8.7.

8.2 Technical concepts

In this section we recall the technical concepts of program algebra from [22, 27].

8.2.1 Program algebra (PGA)

Program algebra (PGA) [22] is an algebraic framework for the description of sequential programming languages. PGA is generated by a set of *primitive instructions* and two compositions *concatenation* and *repetition*. These primitive instructions are *basic instructions, termination, tests (positive/negative)* and *jumps*. Let Σ be a set of basic instructions. Each basic instruction returns a boolean value upon execution. Program expressions in PGA over Σ, denoted by PGA_Σ, are generated by a collection of primitive instructions and two composition constructs. These primitive instructions are:

Basic instruction All $a \in \Sigma$ are basic instructions. Upon the execution of a basic instruction, a boolean value is generated and a state may be modified. After execution, a program has to execute its subsequent instruction.

Termination instruction Termination instruction, denoted by !, indicates termination of the program.

Positive test instruction For each $a \in \Sigma$, there is a positive test instruction denoted by $+a$. If $+a$ is performed by a program, then first a is executed. The state is affected according to a. In case true is returned, the subsequent instruction is performed. In case false is returned, the next instruction is skipped and the execution continues with the following instruction.

Negative test instruction For each $a \in \Sigma$, there also exists a negative test instruction denoted by $-a$. If $-a$ is performed by a program, then first a is executed. The state is affected according to a. In case false is returned, the subsequent instruction is executed. In case true is returned, the next instruction is skipped and the execution proceeds with the following instruction.

Forward jump instruction For any natural number k, there is an instruction $\#k$ which denotes a jump of length k. The number k is the counter of the jump instruction.

- If $k = 0$, the jump is to itself (zero steps forward). In this case inaction will result.
- If $k = 1$, the instruction is skipped. The subsequent instruction is executed next.
- If $k > 1$, the execution skips the next $k - 1$ instructions. The instruction after that is performed.

If there is no instruction to be executed, inaction will occur.
The two composition constructs of PGA are:

Concatenation The concatenation of two programs X and Y in PGA_Σ, denoted by $X; Y$, is also in PGA_Σ.

Repetition The repetition of a program X in PGA_Σ, denoted by X^ω, is also in PGA_Σ.

The *unfolding* of a repetition can be derived: $X^\omega = X; X^\omega$.

8.2.2 Programming languages based on program algebra

Based on PGA, more complex programming languages can be developed and studied by providing simple and general constructions. This section introduces two variants of PGA: PGLE and PGLS. The program notation PGLE describes programming languages with labels and gotos while the program notation PGLS describes structured programming languages in general.

PGLE as a programming language with labels and gotos

PGLE is a modification of PGA by allowing the use of labels and gotos, and omitting the jump instructions and repetition. We recall the descriptions of labels and gotos from [22].

Label instruction For a natural number k, $\pounds k$ is a skip instruction that cannot modify a state. Upon execution, this instruction is simply passed.

Goto instruction For a natural number k, $\#\#\pounds k$ represents a jump to the leftmost occurrence of the label instruction $\pounds k$ in the program. If no such instruction can be found termination will occur.

The descriptions of the other primitive instructions in PGLE are given as in PGA (see Section 8.2.1). However, unlike PGA, in the execution of a program in PGLE, if there are no instructions to be performed, termination will occur. Programs in PGLE must satisfy the following restriction: each test instruction (positive or negative) must always be immediately followed by a goto instruction or a termination instruction. With this condition, in PGLE one can omit repeated occurrences of label instructions, while preserving the same behavior. Furthermore, one can simply replace a goto instruction by termination if a label with the same number does not exist, or simply leave out a label that has no associated goto instruction. For our purpose, we provide another restriction on programs in PGLE: labels of a program in PGLE must occur with different numbers. Furthermore, for a label instruction there must be at least an associated goto instruction, and vice versa, for a goto instruction there must be an associated label instruction.

Example 8.1 The typical program with labels and gotos in modern programming languages

```
L0:
statement1;
IF condition1 THEN GO L0 END IF;
EXIT;
statement2
```

can be formulated in PGLE as follows

$$\pounds 0; \texttt{statement1}; +\texttt{condition1}; \#\#\pounds 0; !; \texttt{statement2}.$$

Note that for simplicity, we only consider removing gotos for PGLE. Extensions of a PGLE program with IF-THEN-ELSE-END IF and WHILE-END WHILE statements can be transformed into PGLE.

PGLS as a structured programming language

PGLS is obtained from PGLE by adding conditional statements and while-loops, and leaving out termination, labels and gotos. A conditional statement in PGLE consists of three following instructions corresponding to three parts IF-THEN, ELSE, and END IF.

Conditional instruction For each basic instruction $a \in \Sigma$ the instructions $+a\{n$ and $-a\{n$ initiate the text of a conditional statement. The number n is the position of the corresponding separator.

Then/else separator The instruction $\}n\{$ connects two sections that are enclosed in braces. The number n is the position of the corresponding end brace.

End brace The instruction $\}$ serves as a closing brace of a conditional statement.

Similarly, a while-loop statement can be defined with the following instructions corresponding to WHILE and END WHILE, or LOOP and END LOOP.

Positive/negative while-loop header For an action $a \in \Sigma$ the instruction $+a\{*n$ and $-a\{*n$ initiate the text of a while-loop. The number n is the position of the corresponding closing brace in the while-loop.

Unconditional while-loop header The instruction $\{*$ initiates the text of an unconditional while-loop, i.e, a while-loop in which the loop condition is always true.

End of while-loop The instruction $*\}n$ serves as a closing brace of a while-loop in connection with its opening brace containing in a while-loop header at position n.

We note that the annotations n of these instructions can be left out if we do not want to emphasize the positions of their relative instructions in a conditional statement or a while-loop. Furthermore, a separator is considered as an opening brace or a closing brace in connection with its end brace or conditional instruction in a conditional statement.

Example 8.2 The following conditional and while-loop statements in modern programming languages

```
IF    condition1 THEN statement1 ELSE statement2 END IF;
WHILE condition2 DO   statement3 END WHILE;
```

are transformed into PGLS as

$$+\texttt{condition1}\{3;\texttt{statement1};\}5\{;\texttt{statement2};\};$$
$$+\texttt{condition2}\{*3;\texttt{statement3};*\}.$$

Definition 8.3 *A* PGLS *program* $X = u_1; \ldots; u_k$ *is* **well-formed** *if it satisfies the following conditions:*

1. *There is a one-to-one correspondence between opening braces and closing braces in* X*. If* (i, n) *is a pair of positions of an opening brace and its corresponding closing brace then* $i < n$*.*
2. *If* (i_j, n_j) $(j = 1, 2)$ *are pairs of positions of an opening brace and its corresponding closing brace such that* $i_1 < i_2$ *then either* $n_1 \leq i_2$ *or* $i_1 < i_2 < n_2 < n_1$*.*

Example 8.4 The program $X = +a\{3; +b\{3; \}4\{; \}$ is not well-formed since there are two opening braces $+a\{3$ and $+b\{3$ for the separator $\}4\{$.

8.2.3 Basic thread algebra (BTA)

The behaviors of programs in PGA and its generated programming languages can be defined in *basic thread algebra* (BTA) [22, 25], an algebraic theory about the semantics of sequential programming languages. We note that basic thread algebra was introduced as *basic polarized process algebra* (BPPA) in [22].

Primitives of BTA

The basic instructions in Σ are called *actions*.

Definition 8.5 *Let* BTA_Σ *be the set of finite threads over* Σ*. It is generated inductively by the following operators:*

- **Termination***:* $S \in \mathrm{BTA}_\Sigma$*.*
- **Inactive behavior***:* $D \in \mathrm{BTA}_\Sigma$*.*
- **Postconditional composition***:* $(-) \trianglelefteq a \trianglerighteq (-)$ *with* $a \in \Sigma$*. The thread* $P \trianglelefteq a \trianglerighteq Q$ *with* $P, Q \in \mathrm{BTA}_\Sigma$ *first performs* a *and then proceeds with* P *if true was produced and with* Q *otherwise. In case* $P = Q$*, we abbreviate this thread by the* **action prefix** *operator:* $a \circ (-)$*. In particular,* $a \circ P = P \trianglelefteq a \trianglerighteq P$*.*

We note that the inactive behavior D represents, for instance, the behavior of an infinite-loop containing no basic instructions or tests.

Threads can be infinite. To define an **infinite** thread in BTA, we require a projective sequence of its finite approximations.

Definition 8.6 *For every $n \in \mathbb{N}$, the* **approximation operator** $\pi_n : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ *is defined inductively by*

$$\begin{array}{lcl} \pi_0(P) & = & D, \\ \pi_{n+1}(S) & = & S, \\ \pi_{n+1}(D) & = & D, \\ \pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) & = & \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q). \end{array}$$

A **projective sequence** *is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$, $\pi_n(P_{n+1}) = P_n$.*

We say that two (finite or infinite) threads are equal exactly if for each $n \in \mathbb{N}$, their n-th approximations are equal. Let $\mathrm{BTA}_\Sigma^\infty$ be the set of all (finite and infinite) threads. *Regular* threads in $\mathrm{BTA}_\Sigma^\infty$ are well-defined (see [104]) and are given as follows.

Definition 8.7 *A thread P is* **regular** *over Σ if $P = E_1$, where E_1 is defined by a finite system of the form ($n \geq 1$):*

$$\{E_i = t_i | 1 \leq i \leq n, t_i = S \text{ or } t_i = D \text{ or } t_i = E_{il} \trianglelefteq a_i \trianglerighteq E_{ir}\}$$

with $E_{il}, E_{ir} \in \{E_1, \ldots, E_n\}$ and $a_i \in \Sigma$.

The finite system in the previous definition is called a *guarded recursive specification.*

Theorem 8.8 *A guarded recursive specification has a unique solution in $\mathrm{BTA}_\Sigma^\infty$.*

Proof: See Theorem 5.41. □

8.2.4 Assigning a thread in BTA to a program in PGA

This section assigns a regular thread in BTA to a program in PGA by means of *behavior extraction equations.* This suggests the notions of *behavioral equivalence* for programs and *program algebra transformations* that map programs of two different programming languages while preserving behavioral equivalence.

Behavior extraction equations

Program behaviors in PGA are given by means of an operator called the *behavior extraction operator* $|-|$ and *behavior extraction equations.*

Definition 8.9 *For a finite program X that does not contain repetition, its behavior is given by $|X| = |X; (\#0)^\omega|$.*

Definition 8.10 *The behavior $|X|$ of an (infinite) program X is determined recursively by the* **behavior extraction equations** *below:*

$$\begin{aligned}
|!;X| &= S,\\
|a;X| &= a \circ |X|,\\
|{+}a;u;X| &= |u;X| \trianglelefteq a \trianglerighteq |X|,\\
|{-}a;u;X| &= |X| \trianglelefteq a \trianglerighteq |u;X|,\\
|\#0;X| &= D,\\
|\#1;X| &= |X|,\\
|\#(k+2);u;X| &= |\#(k+1);X|.
\end{aligned}$$

where u is a primitive instruction, $a \in \Sigma$ and $k \in \mathbb{N}$.

By means of these equations, successive steps of the behavior of a program can be obtained. In the case that a program has a non-trivial loop in which no action occurs, its behavior will be identified with D. Phrased differently: if for a behavior $|X|$ the behavior extraction equations fail to prove $|X| = S$ or $\pi_1(|X|) = a \circ D$ for some $a \in \Sigma$, then $|X| = D$.

It is shown in [30] that every program in PGA can be specified by a regular thread in BTA, and vice versa.

Behavioral equivalence

Behavioral equivalence classifies programs whose behaviors are the same. This program equivalence resembles path equivalence [10] defined for flowcharts.

Definition 8.11 *Programs X and Y are* **behaviorally equivalent** *if $|X| = |Y|$.*

We note that one can provide similar notions of flow-graph and structural equivalence for programs in the setting of PGA (see Section 8.1.1).

Program algebra transformation

A program in a programming language can be transformed into one in another programming language. This transformation is correct if the obtained program does not change its behavior.

Definition 8.12 *A* **transformation** *is a mapping φ from a programming language L_1 to another programming language L_2. This transformation φ is* **correct** *if for every $X \in L_1$, $|X|_{L_1} = |\varphi(X)|_{L_2}$ where $|-|_L$ is an assignment of behaviors to elements of L. This transformation is called a* **projection** *if $L_2 \subseteq L_1$, and an* **embedding** *if $L_1 \subseteq L_2$.*

We can write $|X|$ instead of $|X|_L$ if L is fixed.

8.3 Behaviors for programs in PGLE and PGLS

The behavior extraction equations given in Definition 8.10 for programs in PGA enable single pass execution (each instruction in a PGA program can be dropped after having been processed). They, however, cannot be adapted to programs in PGLE and PGLS because of the complex semantics of labels, gotos, conditional instructions and while-loops. Thus, to determine behaviors for programs in PGLE and PGLS, we define behavior extraction equations based on positions of instructions of programs. Once these behaviors are given, we show that any flowchart can be represented by a PGLE program and vice versa. Furthermore, they can be combined with additional boolean variables.

8.3.1 Behavior extraction equations for labels and gotos

This section presents behavior extraction equations at each position of a program in PGLE. The behavior of this program is determined by the behavior starting at the first position of the program.

Definition 8.13 *Let $X = u_1; \ldots; u_k$ be a program in* PGLE*. The behavior $|X|$ of X is defined by $|X| = |1, X|$, where*

$$|i, X| = \begin{cases} S & \text{if } u_i =! \text{ or } i \notin [1..k], \\ a \circ |i+1, X| & \text{if } u_i = a, \\ |i+1, X| \trianglelefteq a \trianglerighteq |i+2, X| & \text{if } u_\sigma = +a, \\ |i+2, X| \trianglelefteq a \trianglerighteq |i+1, X| & \text{if } u_\sigma = -a, \\ |i+1, X| & \text{if } u_i = \pounds j, \\ |l, X| & \text{if } u_i = \#\#\pounds j,\ u_l = \pounds j. \end{cases}$$

The inactive behavior D will occur if the computation produces no result, i.e., if $|i, X| \neq S$ and there do not exist $n, m \in \mathbb{N}$ such that $|i, X| = |n, X| \trianglelefteq a \trianglerighteq |m, X|$ for some $a \in \Sigma$, then $|i, X| = D$.

We note that by this definition, one can prove that the transformation from PGLE to PGA defined in [22] is correct. This implies that PGLE is not more expressive than PGA.

Example 8.14 Consider the following program `Prog` taken from [27].

```
Prog ::=
br.set(T);
L0:
IF br.eq(T) THEN GO L1;
GO L2;
L1:
br.set(F);
Console.println(hello);
GO L0;
```

```
L2:
br.set(F);
Console.println(goodbye);
GO L0.
```

The program `Prog` can be formulated in PGLE as follows.

$$\begin{array}{rl} \texttt{Prog} = & \texttt{br.set(T)};\pounds 0;+\texttt{br.eq(T)};\#\#\pounds 1;\#\#\pounds 2; \\ & \pounds 1;\texttt{br.set(F)};\texttt{Console.println(hello)};\#\#\pounds 0; \\ & \pounds 2;\texttt{br.set(T)};\texttt{Console.println(goodbye)};\#\#\pounds 0. \end{array}$$

By applying behavior extraction equations, the behavior $|\texttt{Prog}|$ of `Prog` can be determined by

$$\begin{array}{lcl} |\texttt{Prog}| & = & \texttt{br.set(T)} \circ P, \\ P & = & Q \trianglelefteq \texttt{br.eq(T)} \trianglerighteq R, \\ Q & = & \texttt{br.set(F)} \circ \texttt{Console.println(hello)} \circ |\texttt{Prog}|, \\ R & = & \texttt{br.set(T)} \circ \texttt{Console.println(goodbye)} \circ |\texttt{Prog}|. \end{array}$$

We note that in the example above some actions are of the form $f.a(x)$ where f is called a *focus* and $a(x)$ a *co-action*. We will give further explainations to the nature of these co-actions in Section 8.3.4.

8.3.2 Behavior extraction equations for conditional statements and while-loops

This section defines behaviors for programs in the structured program notation PGLS.

Definition 8.15 *Let $X = u_1;\ldots;u_k$ be a* PGLS *program. The behavior extraction equations for conditional statements and while-loops are defined as follows.*

$$|i,X| = \begin{cases} |i+1,X| \trianglelefteq a \trianglerighteq |n+1,X| & \textit{if} \quad u_i = +a\{n, \\ |n+1,X| \trianglelefteq a \trianglerighteq |i+1,X| & \textit{if} \quad u_i = -a\{n, \\ |n+1,X| & \textit{if} \quad u_i = \}n\{, \\ |i+1,X| & \textit{if} \quad u_i = \}, \\ |i+1,X| \trianglelefteq a \trianglerighteq |n+1,X| & \textit{if} \quad u_i = +a\{*n, \\ |n+1,X| \trianglelefteq a \trianglerighteq |i+1,X| & \textit{if} \quad u_i = -a\{*n, \\ |i+1,X| & \textit{if} \quad u_i = \{*, \\ |n,X| & \textit{if} \quad u_i = *\}n, \end{cases}$$

It is shown in [22] that the structured program notation PGLS is strictly weaker than PGLE, i.e., there exists a PGLE program that cannot be transformed into PGLS with the same behavior. This agrees with the "well-known" result of [62, 3, 80, 61, 64] that we cannot replace goto statements with the use of conditional statements and while-loops without additional variables. Definition 8.13 and Definition 8.15 suggest the notions of *reached, reachable, live* and *dead* positions of a program as follows.

Definition 8.16 *Let $X = u_1; \ldots; u_k$ be a program, and i, j two positions in X ($1 \leq i, j \leq k$).*

1. *Position j is* **reached** *from position i (or instruction u_j is reached from instruction u_i), denoted by $i \to j$, if j occurs in the right-hand side of a behavior extraction equation defined for X at position i.*

2. *Position j is* **reachable** *from position i (or instruction u_j is reachable from instruction u_i), if there is a path $i = i_0 \to i_1 \to \cdots \to i_n = j$ of zero or more steps from i to j.*

3. *u_i is* **live** *if i is reachable from 1, otherwise it is* **dead**.

Example 8.17 Consider the program given in Example 8.1. Then the instructions $\#\#£0$ and ! are reached from the instruction +`condition1`. Furthermore, the instruction `statement2` is dead, while the other instructions are live.

8.3.3 Representing flowcharts in PGLE

Flowcharts as illustrated in Figure 8.1 are used to represent programs when dealing with goto removal in literature.

Definition 8.18 *A* **flowchart** *is a finite binary directed graph which has a starting node, labeled by $\downarrow$. It may have a* **successful termination** *node S and an* **unsuccessful termination** *node D. If a node is not a termination node then either it represents an action, and has exactly one outgoing branch, or it represents a test, and has exactly two outgoing branches (negated and unnegated), labeled by $-$ and $+$, respectively.*

Given a flowchart F, a PGLE program X representing F can be defined as follows.

$$X = \psi(£1); \ldots; \psi(£k),$$

where $£1, \ldots, £k$ are labels of the nodes in F, and $£1$ is the label of the starting node. Let $£i$ be the label of a node s in F. Then $\psi(£i)$ is given below.

- If s is the successful termination node S then $\psi(£i) = £i; !$.
- If s is the unsuccessful termination node D then $\psi(£i) = £i; \#\#£i$.
- If s is a test node a that has two outgoing branches labeled by $+$ and $-$ to s_1 labeled by $£j$, and s_2 labeled by $£l$, respectively, then $\psi(£i) = £i; +a; \#\#£j; \#\#£l$.
- Otherwise s represents an action a. It has only one outgoing branch to node s' labeled by $£j$. Then $\psi(£i) = £i; a; \#\#£j$.

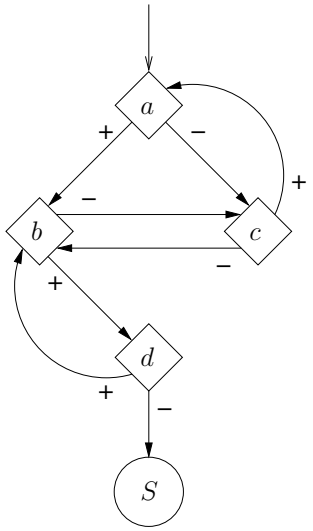

Figure 8.1: Example of a flowchart.

Example 8.19 The flowchart given in Figure 8.1 can be represented in PGLE by

$$\begin{aligned} X = \quad & £1; +a; \#\#£2; \#\#£3; \\ & £2; +b; \#\#£4; \#\#£3; \\ & £3; +c; \#\#£1; \#\#£2; \\ & £4; +d; \#\#£2; \#\#£5; \\ & £5; !. \end{aligned}$$

Conversely, given a PGLE program X, we can produce a corresponding flowchart F as the graphical representation of $|1, X|$.

8.3.4 Combining threads with additional boolean variables

In order to eliminate gotos, some standard algorithms such as [33, 76, 55] introduce new boolean variables. The question is how to combine the behavior of a program with the use of these additional boolean variables. We adapt the interaction between threads and services of [27] to answer this question.

We assume that the value of an additional boolean variable is stored in a memory device called *boolean register* [27] that supports the program in its execution. A boolean register B has four *co-actions*

$$\{\mathtt{set}(T), \mathtt{set}(F), \mathtt{eq}(T), \mathtt{eq}(F)\}$$

and two *states* B_T and B_F representing the truth values T (true) and F (false). The co-actions $\mathtt{set}(b)$ with $b \in \{T, F\}$ always return true after their execution. The co-actions $\mathtt{eq}(b)$ return true after its execution whenever b is the current value of the

register, and false otherwise. Furthermore, the state of the register is flipped whenever it receives a 'set co-action' to the opposite truth value. Initially, $B = B_T$. We note that in [27], a function called *reply function* is also given to define the returned boolean values of the co-actions. For simplicity, we ignore the presence of this reply function in this chapter.

We use the *use-operator* $/_f$ to combine a program behavior P and a boolean register B with *focus* f producing a behavior, written $P/_f B$. This operator is to support a program X in its operation, which will express its value by executing actions that are not processed by B. Upon termination of the execution of X, B is forgotten and so is the state it is in. For a basic action a of a program behavior that does not have a focus, we assume that $a = \epsilon.a$ where ϵ is the *empty* focus.

Definition 8.20 *The semantic equations for the use-operator that combines program behaviors and boolean registers are defined as follows.*

$$\begin{array}{lcll}
S/_f B & = & S & \\
D/_f B & = & D & \\
(P \trianglelefteq g.a \trianglerighteq Q)/_f B & = & P/_f B \trianglelefteq g.a \trianglerighteq Q/_f B & \text{if } g \neq f \\
(P \trianglelefteq f.\mathtt{set}(T) \trianglerighteq Q)/_f B & = & P/_f B_T & \\
(P \trianglelefteq f.\mathtt{set}(F) \trianglerighteq Q)/_f B & = & P/_f B_F & \\
(P \trianglelefteq f.\mathtt{eq}(b) \trianglerighteq Q)/_f B & = & P/_f B & \text{if } B = B_b \\
(P \trianglelefteq f.\mathtt{eq}(b) \trianglerighteq Q)/_f B & = & Q/_f B & \text{if } B \neq B_b \\
(P \trianglelefteq f.c \trianglerighteq Q)/_f B & = & D &
\end{array}$$

where $b \in \{T, F\}$ *and* $c \notin \{\mathtt{set}(T), \mathtt{set}(F), \mathtt{eq}(T), \mathtt{eq}(F)\}$.

Lemma 8.21 *Let* B_1 *and* B_2 *be boolean registers with foci* f_1 *and* f_2*, respectively. Let* P *be a program behavior. Then* $P/_{f_1} B_1/_{f_2} B_2 = P/_{f_2} B_2/_{f_1} B_1$.

Proof: See [27]. □

Example 8.22 Consider the program behavior $|\mathtt{Prog}|$ and the boolean register B with focus `br` in Example 8.14.

$$\begin{array}{lcl}
|\mathtt{Prog}| & = & \mathtt{br.set(T)} \circ P, \\
P & = & Q \trianglelefteq \mathtt{br.eq(T)} \trianglerighteq R, \\
Q & = & \mathtt{br.set(F)} \circ \mathtt{Console.println(hello)} \circ |\mathtt{Prog}|, \\
R & = & \mathtt{br.set(T)} \circ \mathtt{Console.println(goodbye)} \circ |\mathtt{Prog}|.
\end{array}$$

The interaction between $|\mathtt{Prog}|$ and B via the focus `br` produces the behavior $|\mathtt{Prog}|/_{\mathtt{br}} B$ abbreviated by P_B. It can be derived that

$$P_B = \mathtt{Console.println(hello)} \circ \mathtt{Console.println(goodbye)} \circ P_B.$$

8.3.5 Behavioral equivalence with respect to additional variables

Definition 8.20 suggests a program equivalence called *behavioral equivalence with respect to additional variables* that classifies programs whose behaviors in combination with additional variables are the same. This equivalence is coarser than behavioral equivalence, but finer than input-output equivalence of [33, 76, 55].

Definition 8.23 *Programs X and Y are* **behaviorally equivalent with respect to additional variables** *if there are boolean registers $B_1, \ldots, B_n$ and foci $f_1, \ldots, f_n$ such that $|X|/_{f_1}B_1 \ldots /_{f_n}B_n = |Y|/_{f_1}B_1 \ldots /_{f_n}B_n$.*

8.4 Eliminating gotos using additional variables

As mentioned earlier, many algorithms and transformation rules [33, 41, 76, 3, 39, 72] have been proposed to prove the Folk theorem [55] which states that every flowchart is equivalent to a while-program with only one occurrence of a while-loop under input-output equivalence, provided that additional variables are allowed. However, the correctness proofs of these algorithms and transformation rules have not been discussed formally. Furthermore, they are treated under the coarsest equivalence, input-output equivalence. In this section, we show that PGA provides a mathematical framework for reasoning about the correctness of the Folk theorem under a finer equivalence: behavioral equivalence with respect to additional variables. In particular, we formulate the algorithm of Cooper [41] by transforming a program in PGLE (instead of flowcharts) into a structured program in PGLS, and show its correctness under behavioral equivalence with respect to additional variables.

8.4.1 The algorithm

The algorithm of [41] for transforming a program X with labels and gotos to a structured program using additional variables works as follows. The whole program X is put within a while-loop which executes one code fragment belonging to a label each time around the loop. Assume that X contains n labels, we will need $n + 1$ boolean variables for this transformation. The first variable is to handle the conditional of the while-loop. Whenever X terminates, the value of this variable is set to true. The control of the structured program then exits from the while-loop. The other n boolean variables correspond to the n labels in X. Upon the execution of a fragment, the value of the associated variable to the next label is set to true. The control then returns to the beginning of the while-loop. This transformation is illustrated in the following example.

Example 8.24 Consider the program `ProgA` containing labels and gotos written in a syntactic sugared version of PGLE with modern programming features to enhance readability in the left-hand side of Table 8.1. `ProgA` has three labels, and therefore, in order to eliminate all gotos of `ProgA`, we need four additional boolean variables.

```
ProgA ::=

statement0;
GO L1;

L1:

statement1;
GO L3;

L2:

EXIT;

L3:

IF condition THEN
   GO L2;
END IF;
GO L1;
```

```
ProgB ::=
x0.set(FALSE);
x1.set(FALSE);
x2.set(FALSE);
x3.set(FALSE);
statement0;
x1.set(TRUE);
WHILE not x0.eq(TRUE) DO
    IF x1.eq(TRUE) THEN
       x1.set(FALSE);
       statement1;
       x3.set(TRUE);
    ELSE
       IF x2.eq(TRUE) THEN
          x2.set(FALSE);
          x0.set(TRUE);
       ELSE
          IF x3.eq(TRUE) THEN
             x3.set(FALSE);
             IF condition THEN
                x2.set(TRUE);
             ELSE
                x1.set(TRUE);
             END IF;
          ELSE
          END IF;
       END IF;
    END IF;
END WHILE;
```

Table 8.1: Example of goto elimination with additional variables.

Program `ProgA` is transformed into program `ProgB` containing one while-loop in the right-hand side of the figure.

We now assume that the value of an additional boolean variable is stored in a boolean register. We will show that the algorithm above is correct under behavioral equivalence with respect to additional variables by formulating it as a transformation from PGLE to PGLS. In order to reduce the target code, we perform a preprocessing consisting of two transformations `Removing_empty_blocks` and `Adding_implicit_gotos`. The former transformation removes all *redundant* labels and dead instructions, while the latter adds *implicit* gotos to the program. We then eliminate gotos by providing additional variables under behavioral equivalence with respect to these variables. In short, the goto removal using additional variables `Gte_using_variables` from PGLE to PGLS is the composition of three transformations: `Removing_empty_blocks`,

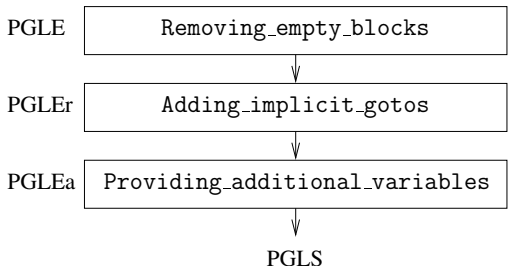

Figure 8.2: Goto elimination using additional variables.

`Adding_implicit_gotos` and `Providing_additional_variables`, illustrated in Figure 8.2. In the following sections, we will formally define and prove correctness of these three transformations.

8.4.2 Removing empty blocks

This section removes redundant labels and dead instructions of a program in PGLE to a program in PGLEr. Here, a redundant label is a label that precedes another label or another goto with a different number. The projection `Removing_empty_blocks` from PGLE to PGLEr works as follows. For each program X in PGLE we perform the following repeatedly:

1. For a label instruction $u_i = £l$ such that $u_{i+1} = £l'$ or $u_{i+1} = \#\#£l'$ with $l \neq l'$, replace it and all goto instructions $\#\#£l$ with the instruction $\#\#£l'$.
2. Remove all dead instructions.

Example 8.25 Consider the program `Prog` in the left-hand side of Table 8.2. Then `Prog` can be refined into `ProgR` in the right-hand side by removing `L4` and `statement2`.

One can see that the target program is behaviorally equivalent to the subject program, i.e.:

Lemma 8.26 *The projection* `Removing_empty_blocks` *from* PGLE *to* PGLEr *is correct.*

Furthermore, since this transformation simply removes certain instructions without altering other instructions, it preserves structural equivalence (see Section 8.1.1).

8.4.3 Adding implicit gotos

The preprocessing `Adding_implicit_gotos` is a crucial step in goto elimination using additional variables. The labels of the obtained program after applying this transformation can only be reached by a goto statement. This transformation adds gotos

```
Prog ::=                              ProgR ::=
statement0;                           statement0;
L1:                                   L1:
statement1;                           statement1;
GO L4;                                GO L3;
L2:                                   L2:
EXIT;                                 EXIT;
L4:
L3:                                   L3:
IF condition THEN GO L2 END IF;       IF condition THEN GO L2 END IF;
GO L1;                                GO L1;
statement2;
```

Table 8.2: Example of removing empty blocks.

```
ProgR ::=                             ProgA ::=
statement0;                           statement0;
                                      GO L1;
L1:                                   L1:
statement1;                           statement1;
GO L3;                                GO L3;
L2:                                   L2:
EXIT;                                 EXIT;
L3:                                   L3:
IF condition THEN GO L2 END IF;       IF condition THEN GO L2 END IF;
GO L1;                                GO L1;
```

Table 8.3: Example of adding implicit gotos.

that are considered *implicit* into a PGLEr program. Here, an implicit goto is a goto to the next instruction (which is the associated label instruction). Furthermore, if the last instruction of the program is neither a goto nor a termination then the program is ended with an extra termination. The resulting program notation is called PGLEa.

Example 8.27 Consider the program `ProgR` in the left-hand side of Table 8.3. This program can be transformed to the program `ProgA` (in the right-hand side) by adding an implicit goto immediately before label `L1`.

Definition 8.28 *The projection from* PGLEr *to* PGLEa *is defined by:*

$$\texttt{Adding\_implicit\_gotos}(u_1;\ldots;u_k) = \psi_1(u_1);\ldots;\psi_k(u_k)$$

where

$$\begin{array}{lcll} \psi_i(\pounds l) & = & \#\#\pounds l;\pounds l & \textit{if } i=1 \textit{ or } u_{i-1}=a \textit{ or } u_{i-2}=\pm a,\\ \psi_k(u) & = & u;! & \textit{if } u_k \neq \#\#\pounds l \textit{ and } u_k \neq !,\\ \psi_i(u) & = & u & \textit{otherwise.} \end{array}$$

Lemma 8.29 *The projection* `Adding_implicit_gotos` *from* PGLEr *to* PGLEa *is correct.*

Proof: Straightforward. □
We note that this projection indeed preserves structural equivalence, since it does not rearrange any instructions of the programs.

8.4.4 Goto elimination by providing additional variables

This section defines a transformation of a program X from PGLEa to PGLS, working as the algorithm described in Section 8.4.1. Assume that X contains n labels, we use $n+1$ boolean registers $B_0, \ldots, B_n$, and $n+1$ foci $x_0, \ldots, x_n$ representing $n+1$ additional variables. The transformation from PGLEa to PGLS is described below.

Definition 8.30 *Let X be a program in* PGLEa. *Without loss of generality (Wlog) we assume that*

$$\begin{array}{rll} X = & & u_{0,1}; \ldots; u_{0,k_0}; \\ & \pounds 1; & u_{1,1}; \ldots; u_{1,k_1}; \\ & & \vdots \\ & \pounds n; & u_{n,1}; \ldots; u_{n,k_n} \end{array}$$

where $u_{i,j}$ are not label instructions. Let $B_0, \ldots, B_n$ be boolean registers with foci $x_0, \ldots, x_n$. The transformation `Providing_additional_variables` *from* PGLEa *to* PGLS *is defined as follows.*

$$\begin{aligned} &\texttt{Providing\_additional\_variables}(X) \\ &\quad = x_0.\texttt{set}(F); x_1.\texttt{set}(F); \ldots; x_n.\texttt{set}(F); \psi_{0,1}(u_{0,1}); -x_0.\texttt{eq}(T)\{*; \Phi_1; *\} \end{aligned}$$

where

$$\begin{array}{lcl} \Phi_i & = & +x_i.\texttt{eq}(T)\{; x_i.\texttt{set}(F); \psi_{i,1}(u_{i,1}); \}\{; \Phi_{i+1}; \}; \text{ for } i < n, \\ \Phi_n & = & +x_n.\texttt{eq}(T)\{; x_n.\texttt{set}(F); \psi_{n,1}(u_{n,1}); \}\{; \}; \end{array}$$

and where

$$\begin{array}{lcl} \psi_{i,j}(!) & = & x_0.\texttt{set}(T), \\ \psi_{i,j}(a) & = & a; \psi_{i,j+1}(u_{i,j+1}), \\ \psi_{i,j}(\pm a) & = & \pm a\{; \psi_{i,j+1}(u_{i,j+1}); \}\{; \psi_{i,j+2}(u_{i,j+2}); \}, \\ \psi_{i,j}(\#\#\pounds l) & = & x_l.\texttt{set}(T). \end{array}$$

We now show that the transformation `Providing_additional_variables` from PGLEa to PGLS preserves behavioral equivalence with respect to additional variables.

Lemma 8.31 *Let X be a program in* PGLEa. *Then*

$$|\texttt{Providing\_additional\_variables}(X)|/_{x_0}B_0/_{x_1}B_1 \ldots /_{x_n}B_n = |X|.$$

Proof: Let $P = |X|$. It can be derived that $P = P_{0,1}$, where $P_{i,j}$ ($0 \leq i \leq n$, $0 \leq j \leq k_i$) are determined by the following system

$$P_{i,j} = \begin{cases} S & \textit{if} \quad u_{i,j} =!, \\ a \circ P_{i,j+1} & \textit{if} \quad u_{i,j} = a, \\ P_{i,j+1} \trianglelefteq a \trianglerighteq P_{i,j+2} & \textit{if} \quad u_{i,j} = +a, \\ P_{i,j+2} \trianglelefteq a \trianglerighteq P_{i,j+1} & \textit{if} \quad u_{i,j} = -a, \\ P_{i,1} & \textit{if} \quad j = 0, \\ P_{l,1} & \textit{if} \quad u_{i,j} = \#\#\pounds l. \end{cases}$$

Now let $Q = |\texttt{Providing\_additional\_variables}(X)|$. By Definition 8.15,

$$\begin{array}{lcl} Q & = & x_0.\texttt{set}(F) \circ x_1.\texttt{set}(F) \circ \cdots \circ x_n.\texttt{set}(F) \circ M_{0,1}, \\ M & = & S \trianglelefteq x_0.\texttt{eq}(T) \trianglerighteq M_{1,0} \end{array}$$

where

$$M_{i,j} = \begin{cases} x_0.\texttt{set}(T) \circ M & \textit{if } u_{i,j} =!, \\ a \circ M_{i,j+1} & \textit{if } u_{i,j} = a, \\ M_{i,j+1} \trianglelefteq a \trianglerighteq M_{i,j+2} & \textit{if } u_{i,j} = +a, \\ M_{i,j+2} \trianglelefteq a \trianglerighteq M_{i,j+1} & \textit{if } u_{i,j} = -a, \\ (x_i.\texttt{set}(F) \circ M_{i,1}) \trianglelefteq x_i.\texttt{eq}(T) \trianglerighteq M_{i+1,0} & \textit{if } j = 0, \\ x_l.\texttt{set}(T) \circ M & \textit{if } u_{i,j} = \#\#\pounds l. \end{cases}$$

Let $Q_{i,j} = x_0.\texttt{set}(F) \circ x_1.\texttt{set}(F) \circ \cdots \circ x_n.\texttt{set}(F) \circ M_{i,j}/_{x_0}B_0/_{x_1}B_1 \ldots /_{x_n}B_n$. Then $Q/_{x_0}B_0/_{x_1}B_1 \ldots /_{x_n}B_n = Q_{0,1}$. It follows from Definition 8.20 and Lemma 8.21 that

$$Q_{i,j} = \begin{cases} S & \textit{if} \quad P_{i,j} = S, \\ D & \textit{if} \quad P_{i,j} = D, \\ Q_{i,l} \trianglelefteq a \trianglerighteq Q_{i,r} & \textit{if} \quad P_{i,j} = P_{i,l} \trianglelefteq a \trianglerighteq P_{i,r}. \end{cases}$$

with $j, l, r \in [1..k_i]$. It follows from Theorem 8.8 that $Q_{i,j} = P_{i,j}$ for all $0 \leq i \leq n$ and $0 \leq j \leq k_i$. Therefore,

$$|\texttt{Providing\_additional\_variables}(X)|/_{x_0}B_0/_{x_1}B_1 \ldots /_{x_n}B_n = Q_{0,1} = P_{0,1} = |X|.$$

□

We have shown correctness and equivalence of the transformations `Removing_empty_blocks`, `Adding_implicit_gotos` and `Providing_additional_variables`. This implies that goto statements can be eliminated from the setting of PGA using additional boolean variables under behavioral equivalence with respect to these variables, a finer equivalence than input-output equivalence of [33, 55].

Theorem 8.32 *The projection* `Gte_using_variables` *from PGLE to PGLS is correct under behavioral equivalence with respect to additional variables.*

Proof: This follows from Lemma 8.26, Lemma 8.29 and Lemma 8.31. □

We hereby have shown that the algorithm of Cooper for the Folk theorem is correct under behavioral equivalence with respect to additional variables.

8.5 Eliminating gotos without the use of additional variables

In this section, we consider goto removal without the use of additional variables. We intend to apply the forward and backward elimination rules of Peterson et al. and Ramshaw [80, 83] to eliminate gotos, by introducing loops with multi-level exits into the language. Since these rules preserve structural equivalence, it is required that the program must be free of so-called *head-to-head crossings*. Hence, in order to eliminate gotos without the use of additional variables under behavioral equivalence, we perform a preprocessing that gets rid of head-to-head crossings for a program containing labels and gotos, and subsequently employ the result of Peterson et al. and Ramshaw.

We introduce the program notation PGLM which is a modification of PGLE by replacing labels and gotos with loops and multi-level exits. The first step is to *remove empty blocks* and *add implicit gotos* to the program as described in Section 8.4.2 and Section 8.4.3. We then *reorder* labels in the program in a reasonable way. Note that this step can be omitted, however, this would double certain code fragments after getting rid of head-to-head crossings (see Example 8.53). We *get rid of head-to-head crossings* by copying certain blocks of code in the program. In order to use the result of Peterson et al. and Ramshaw, we *remove all implicit gotos* and *add extra labels* to the program. Finally, we *apply the Elimination rules* of [80, 83] to remove all gotos. The goto elimination without the use of additional variables `Gte_without_variables` from PGLE to PGLM is the composition of the transformations above and is depicted in Figure 8.3.

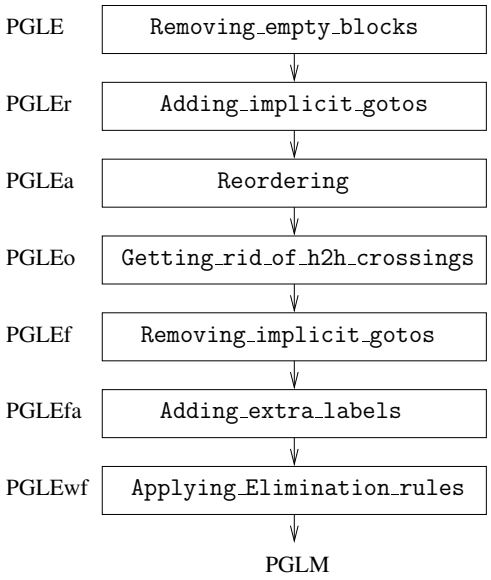

Figure 8.3: Goto elimination without additional variables.

Our algorithm is illustrated by two examples taken from Example 8.25 and Example 8.19 running throughout every step of the goto removal: one is written in a syntactic sugared version (using modern programming features to enhance readability), and one is written in PGLE, respectively.

8.5.1 The program notation PGLM with loops and multi-level exits

Loops and multi-level exits are introduced in modern programming languages such as Ada, Java and C# to replace the use of goto statements. In this section we provide the program notation PGLM in the setting of PGA which is a variant of PGLE by leaving out labels and gotos and adding *loops with multi-level exits*. This program notation consists of the following instructions:

Basic instruction Basic instructions are given as in PGA.

Test instruction Test instructions consist of positive tests and negative tests given as in PGA. Each test instruction must be immediately followed by an exit instruction.

Loop with multi-level exit For any natural number k, the instructions $k\{$ and $k\}n$, where n is the position of $k\{$, serve as an opening and closing braces of a loop labeled by k. The exit instruction $\mathrm{break}\, k^m$ is inside the loop with label k, and serves as an exit statement from the loop with label k to the instruction at position $m+1$ in the program, where m is the position of the instruction $k\}n$.

We note that the notations n and m can be left out if we do not want to emphasize the positions of opening braces and closing braces of loops with multi-level exits in a PGLM program.

Example 8.33 A typical program in Java with loops and multi-level exits

```
X::= outerloop:
     LOOP
        innerloop:
        LOOP
          IF something_really_bad_happened THEN
             break outerloop;
          END IF;
        END innerloop;
     END outerloop;
```

can be formulated in PGLM as

$$X ::= 1\{; 2\{; +\texttt{something\_really\_bad\_happened}; \mathrm{break}\, 1^6; 2\}2; 1\}1.$$

Similar to Definition 8.3, we define the notion of well-formed programs in PGLM as follows.

Definition 8.34 *A* PGLM *program is* **well-formed** *if it satisfies the following conditions:*

1. *There are one-to-one correspondences between opening braces and closing braces, and between exits and closing braces of loops. Furthermore, if* $u_n = k\{$, $u_i = \mathrm{break}\, k^m$ *and* $u_m = k\}n$ *form a loop with multi-level exit then* $n < i < m$.

2. *If* (n_j, m_j), $j = 1, 2$, *are positions of opening braces and their corresponding closing braces of two loops such that* $n_1 < n_2$ *then either* $m_1 < n_2$ *or* $n_1 < n_2 < m_2 < m_1$.

8.5.2 Behavior extraction equations for loops with multi-level exits

In this section, we give behavior extraction equations for the instructions of loops with multi-level exits in order to determine the behavior of a program in PGLM.

Definition 8.35 *Let* $X = u_1; \ldots; u_k$ *be a program in* PGLM. *The behavior extraction equations for loops with multi-level exits in* X *are given by*

$$|i, X| = \begin{cases} |i+1, X| & \text{if} \quad u_i = k\{, \\ |n, X| & \text{if} \quad u_i = k\}n, \\ |m+1, X| & \text{if} \quad u_i = \mathrm{break}\, k^m. \end{cases}$$

Example 8.36 Consider the program X given in Example 8.33. Then its behavior can be determined as a regular thread given below.

$$\begin{aligned} |X| &= |1, X| = |2, X| = |3, X| = |4, X| \trianglelefteq \texttt{something\_really\_bad\_happened} \trianglerighteq |5, X| \\ &= |7, X| \trianglelefteq \texttt{something\_really\_bad\_happened} \trianglerighteq |2, X| \\ &= S \trianglelefteq \texttt{something\_really\_bad\_happened} \trianglerighteq |X|. \end{aligned}$$

8.5.3 Reordering the labels in programs

The projection `Reordering` from PGLEa (see Section 8.4.3) to PGLEo is a preprocessing for goto elimination in order to get rid of head-to-head crossings in a program. It rearranges the labels of the program respecting the *dominator order* while preserving behavioral equivalence. Here the dominator order means that a label of a goto program which is executed earlier than some other labels should occur first in the program. This rearrangement is to reduce copies of certain code fragments of a subject program in PGLEa when eliminating head-to-head crossings.

```
ProgA ::=                          | ProgO ::=
statement0;                        | statement0;
GO L1;                             | GO L1;
L1:                                | L1:
statement1;                        | statement1;
GO L3;                             | GO L3;
L2:                                | L3:
EXIT;                              | IF condition THEN GO L2 END IF;
L3:                                | GO L1;
IF condition THEN GO L2 END IF;    | L2:
GO L1;                             | EXIT;
```

Table 8.4: Example of reordering.

Example 8.37 Consider the program `ProgA` written in a syntactic sugared version of PGLE in Example 8.27. Then `L1<L3<L2` is a dominator order between the labels of `ProgA`. The program `ProgA` can be transformed to the program `ProgO` given in Table 8.4.

In order to define the dominator order and the transformation `Reordering` in a formal way, we provide some supporting definitions.

Definition 8.38 *Let X be a program in* PGLEa*, and $£i$ a label of X. A label $£n$ ($\neq £i$) of X* **dominates** *$£i$ if every path from the first instruction of X to $£i$ goes through $£n$. A label $£l$ is the* **immediate dominator** *of $£i$, denoted by* $\texttt{immediate\_dominator}(£i) = £l$*, if it dominates $£i$, and there does not exist another dominator $£n$ of $£i$ such that $£l$ dominates $£n$.*

There could be many labels that are immediately dominated by the same label. In Figure 8.4, label $£l$ is the immediate dominator of labels $£i$ and $£j$, while label $£i$ is the immediate dominator of label $£m$. We group these labels in the following definition.

Definition 8.39 *Let X be a program in* PGLEa*, and $£l$ be a label of X. The group* $\texttt{immediately\_dominated\_by}(£l)$ *is defined by*

$$\texttt{immediately\_dominated\_by}(£l) = \{£i |\ \texttt{immediate\_dominator}(£i) = £l\}.$$

Let $£i$ and $£j$ be two labels in $\texttt{immediately\_dominated\_by}(£l)$*. Then $£i$ is* **reachable** *from $£j$* **through** *the group* $\texttt{immediately\_dominated\_by}(£l)$ *if there is a path $i_1 \to \cdots \to i_n$ with $u_{i_1}, \ldots, u_{i_m} \in$* $\texttt{immediately\_dominated\_by}(£l)$ *such that $u_{i_1} = £i$, $u_{i_n} = £j$. Two labels $£i$ and $£j$ are* **connected through** *the group* $\texttt{immediately\_dominated\_by}(£l)$ *if both are reachable from the other through this group.*

We now provide the notion of dominator orders between labels of a program.

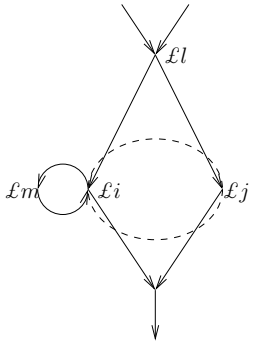

Figure 8.4: An example of dominating between labels in programs. Here $£l$ dominates $£i$ and $£j$, and $£i$ dominates $£m$.

Definition 8.40 *Let X be a program in* PGLEa. *A* **dominator order** $<$ *between the labels occurring in X is defined as follows. For all elements $£i$, $£j$ in* `immediately_dominated_by`$(£l)$ *in X:*

1. $£l < £i$.
2. *If $£i$ is reachable from $£j$ and $£j$ is not reachable from $£i$, through the group* `immediately_dominated_by`$(£l)$, *then $£i < £j$.*
3. *If there is no connection between $£i$ and $£j$ or they are connected through* `immediately_dominated_by`$(£l)$, *then either $£i < £j$ or $£j < £i$.*
4. *If $£i < £j$ then for all $£m$ in X such that $£i$ dominates $£m$, $£m < £j$.*

For a program X in PGLEa, there are several ways to define a dominator order between the labels of X.

Example 8.41 Consider the program X given in Example 8.19. Then $£1 < £3 < £2 < £4 < £5$ and $£1 < £2 < £4 < £3 < £5$ are two dominator orders which can be defined for the labels in X.

Definition 8.42 *A* PGLEa *program X is* **in dominator order** *when the textual order of its labels is a dominator order.*

Let PGLEo be the set of PGLEa programs that are in dominator order.

Definition 8.43 *Let X be a program in* PGLEa. *Wlog we assume that*

$$\begin{array}{rl} X = & u_{0,1}; \ldots; u_{0,k_0}; \\ £1; & u_{1,1}; \ldots; u_{1,k_1}; \\ & \vdots \\ £n; & u_{n,1}; \ldots; u_{n,k_n} \end{array}$$

where $u_{i,j}$ are not label instructions. The projection `Reordering` *from* PGLEa *to* PGLEo *is defined by*

$$\texttt{Reordering}(X) = u_{0,1}; \ldots; u_{0,k_0}; \psi(£_{i_1}); \ldots; \psi(£_{i_n})$$

where $£i_1 < \ldots < £i_n$ is a dominator order between the labels of X, and where

$$\psi(£i) = £i; u_{i,1}; \ldots; u_{i,k_i}$$

Lemma 8.44 *The projection* `Reordering` *from* PGLEa *to* PGLEo *is correct.*

Proof: Straightforward. □
Indeed, it can be derived that the projection `Reordering` preserves flow-graph equivalence.

Example 8.45 Consider the program X from Example 8.19 and Example 8.41. Then we can project X into a program in PGLEo as in Table 8.5.

$X =$	`Reordering`$(X)=$
$£1; +a; \#\#£2; \#\#£3;$	$£1; +a; \#\#£2; \#\#£3;$
$£2; +b; \#\#£4; \#\#£3;$	$£3; +c; \#\#£1; \#\#£2;$
$£3; +c; \#\#£1; \#\#£2;$	$£2; +b; \#\#£4; \#\#£3;$
$£4; +d; \#\#£2; \#\#£5;$	$£4; +d; \#\#£2; \#\#£5;$
$£5; !$	$£5; !$

Table 8.5: Example of reordering.

8.5.4 Getting rid of head-to-head crossings

This section defines a projection from PGLEo into PGLEf to get rid of *head-to-head crossings*.

Definition 8.46 *Let $X = u_1; \ldots; u_k$ be a program in* PGLE*, and $u_i = £l$ a label instruction of X. The instruction $u_j = \#\#£l$ of X is a* **backward** *goto, written as* `bw`$(\#\#£l, j)$ *if $i < j$, and a* **forward** *goto, written as* `fw`$(\#\#£l, j)$ *otherwise. The label $£l$ is a* **backward goto label***, written as* `bw`$(£l)$*, if it is the label of some backward goto. The label $£m$ is a* **forward goto label***, written as* `fw`$(£m)$*, if it is the label of some forward goto.*

We note that a label instruction of a PGLE program can be both backward and forward goto.

Definition 8.47 *Let X be a program in* PGLE*. A backward goto instruction $u_i = \#\#£n$ causes a* **head-to-head crossing** *in X (see Figure 8.5), denoted by* `head2head`$(\#\#£n, i)$*, if there is a forward goto instruction $u_j = \#\#£m$ in X such that $j < i' < j' < i$, where i' and j' are positions of $£n$ and $£m$, respectively.*

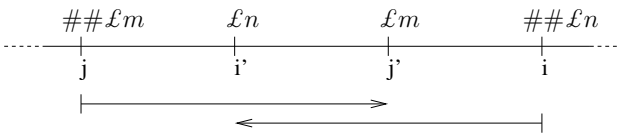

Figure 8.5: A head-to-head crossing.

Let PGLEf be the set of PGLE programs that are free of head-to-head crossings. To project a program in PGLEo to a program in PGLEf, we represent the number in a label instruction in $\texttt{Getting\_rid\_of\_h2h\_crossings}(X)$ by a pair (i, m). The number m is the position of this label and the number i is the position of the equivalent label instruction in X to emphasize that from these positions the two programs $\texttt{Getting\_rid\_of\_h2h\_crossings}(X)$ and X behave indentically. We denote these labels in $\texttt{Getting\_rid\_of\_h2h\_crossings}(X)$ as $£^i_m$. Therefore, the gotos are denoted by $\#\#£^i_m$ as well.

The projection `Getting_rid_of_h2h_crossings` of a program X from PGLEo to PGLEf works as follows. Let $Y = \texttt{Getting\_rid\_of\_h2h\_crossings}(X)$. At the beginning, the program Y is similar to X, except that its labels and gotos are represented as explained in the previous paragraph. Assume that at position n of the program, a head-to-head crossing occurs because of a goto instruction $\#\#£^i_m$. We then look for a label $£^i_{m'}$ with $m' < n$ such that there is no head-to-head crossing at n if we replace $\#\#£^i_m$ by $\#\#£^i_{m'}$. If such position m' does not exist, we copy a part of the program X from position i to the instruction standing before the next label in X and replace the goto $\#\#£^i_m$ by this part. This step is performed repeatedly until the target program is free of head-to-head crossings as described in the next definition.

Definition 8.48 *Let $X = u_1; \ldots; u_k$ be a program in* PGLEo*. The projection* `Getting_rid_of_h2h_crossings` *from* PGLEo *to* PGLEf *is defined as follows.*

$$Y = \texttt{Getting\_rid\_of\_h2h\_crossings}(X) = \psi_1(u_1); \ldots; \psi_k(u_k).$$

We denote $[Y]_i$ as the instruction at position i of Y. The auxiliary functions ψ_σ are given by

$$\begin{array}{lcl}
\psi_i(£l) & = & \psi_i(£^i_i) \\
\psi_\sigma(\#\#£l) & = & \psi_\sigma(\#\#£^i_i) \text{ where } u_i = £l, \\
\psi_\sigma(\#\#£^i_m) & = & \begin{cases} \#\#£^i_m & \text{if } \neg\texttt{head2head}(\#\#£^i_m, n), \\ \#\#£^i_{m'} & \text{if } \exists m' < n \wedge \neg\texttt{head2head}(\#\#£^i_{m'}, n), \\ \texttt{copy}(\sigma, i, n) & \text{otherwise}, \end{cases} \\
\psi_\sigma(u) & = & u \text{ otherwise}
\end{array}$$

where n is the position of $\psi_\sigma(\#\#£^i_m)$. The function $\texttt{copy}(\sigma, i, n)$ is given as follows.

$$\texttt{copy}(\sigma, i, n) = \begin{cases} £^i_n; \psi_{\sigma,i+1}(u_{i+1}); \ldots; \psi_{\sigma,i+m_i}(u_{i+m_i}) & \text{if } [Y]_{n-1} \neq \pm a \\ \#\#£^i_{n+2}; \#\#£n; & \\ £^i_{n+2}; \psi_{\sigma,i+1}(u_{i+1}); \ldots; \psi_{\sigma,i+m_i}(u_{i+m_i}); £n & \text{otherwise} \end{cases}$$

with m_i a minimal number satisfying $u_{i+m_i+1} = £l$ for some l.

We note that in the previous definition we have a label of the form $£n$. This label is to mark the end of an inserted code fragment. The projection above is illustrated by the following example.

Example 8.49 Consider the PGLEo program given in Example 8.45. This program can be projected to a program in PGLEf as follows.

$$
\begin{aligned}
&\texttt{Getting\_rid\_of\_h2h\_crossings}(\\
&\qquad £1; +a; \#\#£2; \#\#£3;\\
&\qquad £3; +c; \#\#£1; \#\#£2;\\
&\qquad £2; +b; \#\#£4; \#\#£3;\\
&\qquad £4; +d; \#\#£2; \#\#£5;\\
&\qquad £5; !)\\
&= \psi_1(£1); \psi_2(+a); \psi_3(\#\#£2); \psi_4(\#\#£3);\\
&\quad \psi_5(£3); \psi_6(+c); \psi_7(\#\#£1); \psi_8(\#\#£2);\\
&\quad \psi_9(£2); \psi_{10}(+b); \psi_{11}(\#\#£4); \psi_{12}(\#\#£3);\\
&\quad \psi_{13}(£4); \psi_{14}(+d); \psi_{15}(\#\#£2); \psi_{16}(\#\#£5);\\
&\quad \psi_{17}(£5); \psi_{18}(!)\\
&= \psi_1(£^1_1); +a; \psi_3(\#\#£^9_9); \psi_4(\#\#£^5_5);\\
&\quad \psi_5(£^5_5); +c; \psi_7(\#\#£^1_1); \psi_8(\#\#£^9_9);\\
&\quad \psi_9(£^9_9); +b; \psi_{11}(\#\#£^{13}_{13}); \psi_{12}(\#\#£^5_5);\\
&\quad \psi_{13}(£^{13}_{13}); +d; \psi_{15}(\#\#£^9_9); \psi_{16}(\#\#£^{17}_{17});\\
&\quad \psi_{17}(£^{17}_{17}); !\\
&= £^1_1; +a; \#\#£^9_9; \#\#£^5_5;\\
&\quad £^5_5; +c; \#\#£^1_1; \#\#£^9_9;\\
&\quad £^9_9; +b; \#\#£^{13}_{13}; \texttt{copy}(12, 5, 12);\\
&\quad \psi_{13}(£^{13}_{13}); +d; \psi_{15}(\#\#£^9_9); \psi_{16}(\#\#£^{17}_{17});\\
&\quad \psi_{17}(£^{17}_{17}); !\\
&= £^1_1; +a; \#\#£^9_9; \#\#£^5_5;\\
&\quad £^5_5; +c; \#\#£^1_1; \#\#£^9_9;\\
&\quad £^9_9; +b; \#\#£^{13}_{13}; £^5_{12}; \psi_{12,6}(+c); \psi_{12,7}(\#\#£^1_1); \psi_{12,8}(\#\#£^9_9);\\
&\quad \psi_{13}(£^{13}_{13}); +d; \psi_{15}(\#\#£^9_9); \psi_{16}(\#\#£^{17}_{17});\\
&\quad \psi_{17}(£^{17}_{17}); !\\
&= £^1_1; +a; \#\#£^9_9; \#\#£^5_5;\\
&\quad £^5_5; +c; \#\#£^1_1; \#\#£^9_9;\\
&\quad £^9_9; +b; \#\#£^{13}_{13}; £^5_{12}; +c; \#\#£^1_1; \#\#£^9_9;\\
&\quad £^{13}_{13}; +d; \#\#£^9_9; \#\#£^{17}_{17};\\
&\quad £^{17}_{17}; !
\end{aligned}
$$

In the following, we prove well-definedness and correctness for the projection `Getting_rid_of_h2h_crossings`.

Lemma 8.50 *The projection* `Getting_rid_of_h2h_crossings` *is well-defined.*

Proof: We prove that when projecting a PGLEo program X to a program in PGLEf at position n, the function $\texttt{copy}(\sigma, i, n)$ is terminating. Suppose there is a label $\pounds_p^j$ occurring in $\texttt{copy}(\sigma, i, n)$. This means that at position p, a head-to-head crossing occurs because of some goto instruction $\#\#\pounds_q^j$ for some q. It follows from Definition 8.48 that $j \neq i$ otherwise we would have replaced $\#\#\pounds_q^j$ by $\#\#\pounds_n^i$ (or $\#\#\pounds_{n+2}^i$) which does not cause a head-to-head crossing. Since i and j are positions of some labels in X, the labels occurring in $\texttt{copy}(\sigma, i, n)$ are finite, or the function $\texttt{copy}(\sigma, i, n)$ is terminating. Thus, the projection `Getting_rid_of_h2h_crossings` is well-defined. □

Lemma 8.51 *Let X be a program in* PGLEo.
Then $\texttt{Getting\_rid\_of\_h2h\_crossings}(X)$ *is free of head-to-head crossings.*

Proof: This follows from Definition 8.48 and Lemma 8.50. □

Theorem 8.52 *The projection* `Getting_rid_of_h2h_crossings` *from* PGLEo *to* PGLEf *is correct.*

Proof: Let X and Y be defined as in Definition 8.48. We prove that $|X| = |Y|$. Wlog we assume that if $u_i = \pounds l$ then $l = i$. Let i_σ denote the position of $\psi_{\sigma,i}(u_i)$ in Y, and let $P = |i, X|$ and $Q = |i_\sigma, Y|$. We distinguish the following cases:

1. $u_i =!$. Then $P_i = S$ and $\psi_{\sigma,i}(u_i) =!$. Hence $Q_{i_\sigma} = P_i = S$.
2. $u_i = \#\#\pounds j$. Then $P_i = P_j$ and $\psi_{\sigma,i}(u_i) = \#\#\pounds_m^j$ for some m or $\psi_{\sigma,i}(u_i) = \texttt{copy}((\sigma, i), j, i_\sigma)$. It follows from Definition 8.48 and Definition 8.15 that $Q_{i_\sigma} = Q_{j_\delta}$ for some δ.
3. $u_i = a$. Then $P_i = a \circ P_{i+1}$ and $\psi_{\sigma,i}(u_i) = a$. This implies that $Q_{i_\sigma} = a \circ Q_{i+1_{\sigma,i+1}}$.
4. $u_i = +a$. Then $P_i = P_{i+1} \trianglelefteq a \trianglerighteq P_{i+2}$. There are two cases:
   (a) $\psi_{\sigma,i+1}(u_{i+1})$ is an instruction. Then $Q_{i_\sigma} = Q_{i+1_{\sigma,i+1}} \trianglelefteq a \trianglerighteq Q_{i+2_{\sigma,i+2}}$.
   (b) $\psi_{\sigma,i+1}(u_{i+1}) = \#\#\pounds_{n+2}^j; \#\#\pounds n; \pounds_{n+2}^j; \ldots; \pounds n$ with $n = i_\sigma + 1$ and j some position of X. By Definition 8.15, we can also get $Q_{i_\sigma} = Q_{i+1_{\sigma,i+1}} \trianglelefteq a \trianglerighteq Q_{i+2_{\sigma,i+2}}$.
5. $u_i = -a$. Then $P_i = P_{i+2} \trianglelefteq a \trianglerighteq P_{i+1}$. Similar to the previous case, $Q_{i_\sigma} = Q_{i+2_{\sigma,i+2}} \trianglelefteq a \trianglerighteq Q_{i+1_{\sigma,i+1}}$.

Hence for all positions i and i_σ of X and Y,

1. if $P_i = S$ then $Q_{i_\sigma} = S$;
2. if $P_i = P_n$ then $Q_{i_\sigma} = Q_{n_\delta}$ for some δ;
3. if $P_i = P_m \trianglelefteq a \trianglerighteq P_n$ then $Q_{i_\sigma} = Q_{m_\delta} \trianglelefteq a \trianglerighteq Q_{n_\gamma}$ for some δ and γ

```
ProgA ::=
statement0;
GO L1;
L1:
statement1;
GO L3;
L2:
EXIT;
L3:
IF condition THEN GO L2 END IF;
GO L1;
```

```
ProgF ::=
statement0;
GO L1;
L1:
statement1;
GO L3;
L2:
EXIT;
L3:
IF condition THEN GO L4 END IF;
GO L5;
L4:
EXIT;
L5:
GO L1;
```

Table 8.6: Example of getting rid of head-to-head crossings.

and vice versa. It follows from Theorem 8.8 that $P_i = Q_{i_\sigma}$ for all positions i and i_σ of X and Y. Therefore $|X| = |1, X| = |1, Y| = |Y|$. □

We note that the projection `Getting_rid_of_h2h_crossings` also works on PGLEa, i.e., we can apply this projection to get rid of head-to-head crossings even in the case that the program is not in dominator order. However, this would double certain code fragments of the program as can be seen in the following example.

Example 8.53 Consider the PGLEa program `ProgA` in Example 8.37. We can eliminate head-to-head crossings of `ProgA` by applying the projection `Getting_rid_of_h2h_crossings` directly on `ProgA` as can be seen in Table 8.6. The code fragment `L2;EXIT` of `ProgA` is copied to `L4;EXIT` in the resulting program `ProgF`. This would not be necessary if we apply the projection `Getting_rid_of_h2h_crossings` on the reordering `ProgO` of `ProgA` which is already free of head-to-head crossings. We note that the numbers occurring in the label and goto statements of `ProgF` are simplified.

8.5.5 Removing implicit gotos

Our programs now are free of head-to-head crossings, a condition that is sufficient to replace all labels and gotos with loops and multi-level exits. To reduce the target code of the replacement, we remove all implicit gotos of the program. This procedure is a reverse of the projection `Adding_implicit_gotos` given in Section 8.4.3, and is defined on PGLE. We note that it can happen that *implicit* labels which have no associated gotos will occur after removing certain implicit gotos. These labels should also be left out.

The projection `Removing_implicit_gotos` from PGLE to PGLE$^-$ works as follows. For a program $X = u_1; \ldots; u_k$ in PGLE, we perform the following repeatedly:

ProgO=	ProgF=
statement0;	statement0;
GO L1;	
L1:	L1:
statement1;	statement1;
GO L3;	
L3:	
IF condition THEN GO L2;	IF condition THEN GO L2;
GO L1;	GO L1;
L2:	L2:
EXIT;	EXIT;

Table 8.7: Example of removing implicit gotos.

1. For an implicit goto $\#\#\pounds l$ at position i, if $i = 1$ or u_{i-1} is not a test ($u_i \neq \pm a$) then remove u_i;
2. Remove all implicit labels.

Lemma 8.54 *The projection* `Removing_implicit_gotos` *from* PGLE *to* PGLE$^-$ *is correct.*

Proof: Straightforward. □

We note that one can obtain that the projection `Removing_implicit_gotos` preserves structural equivalence. Let PGLEfa be the set of PGLEf programs that are free of implicit gotos.

Example 8.55 Consider the program `ProgO` in Example 8.37. This program is free of head-to-head crossings, and therefore, it is in PGLEf. We can remove implicit labels and gotos of `ProgO` to the program `ProgF` as described in Table 8.7.

Example 8.56 Let F be the program obtained from the program in Example 8.49 by replacing the labels (δ, i) with concrete natural numbers. We can remove all implicit gotos from F as in Table 8.56.

$F =$	`Removing_implicit_gotos`$(F) =$
$\pounds 1; +a; \#\#\pounds 2; \#\#\pounds 3;$	$\pounds 1; +a; \#\#\pounds 2;$
$\pounds 3; +c; \#\#\pounds 1; \#\#\pounds 2;$	$+c; \#\#\pounds 1;$
$\pounds 2; +b; \#\#\pounds 4;$	$\pounds 2; +b; \#\#\pounds 4;$
$\pounds 31; +c; \#\#\pounds 1; \#\#\pounds 2;$	$+c; \#\#\pounds 1; \#\#\pounds 2;$
$\pounds 4; +d; \#\#\pounds 2; \#\#\pounds 5;$	$\pounds 4; +d; \#\#\pounds 2;$
$\pounds 5; !$	$!;$

Table 8.8: Example of removing implicit gotos.

8.5.6 Adding extra labels

In the previous section, we have defined a projection to get rid of head-to-head crossings in a goto program. As explained in [83], this is a sufficient condition to replace all labels and gotos in a program under structural equivalence. However, in order to achieve Elimination rules, it is required that PGLEfa programs must be *well-formed*, meaning that a label of a PGLEfa program must be either a forward goto label or a backward goto label (see Definition 8.46). Let PGLEwf be the set of well-formed programs in PGLEfa. This section provides a projection from PGLEfa to PGLEwf that transforms PGLEfa programs to well-formed programs in PGLEwf by adding an extra label next to the label of both forward and backward gotos, and replacing its forward gotos with the gotos associated with the new label. Formally:

Definition 8.57 *Let* $X = u_1; \ldots; u_k$ *be a program in* PGLEfa. *The projection* `Adding_extra_labels` *is defined as follows.*

$$\texttt{Adding\_extra\_labels}(X) = \psi_1(u_1); \ldots; \psi_k(u_k)$$

where the auxiliary functions $\psi_i(u_i)$ *are defined by*

$$\begin{array}{lll} \psi_i(\#\#\pounds l) = & \#\#\pounds l + n; & \textit{if } \texttt{fw}(\#\#\pounds l, i) \wedge \texttt{bw}(\pounds l), \\ \psi_i(\pounds l) = & \pounds l + n; \pounds l; & \textit{if } \texttt{fw}(\pounds l) \wedge \texttt{bw}(\pounds l), \\ \psi_i(u) = & u; & \textit{otherwise} \end{array}$$

and where $n = m + 1$, *and* m *is the maximum number in the labels of* X.

Lemma 8.58 *The projection* `Adding_extra_labels` *from* PGLEfa *to* PGLEwf *is correct.*

Proof: Straightforward. □

Again, it can be shown that the projection `Adding_extra_labels` preserves structural equivalence.

Example 8.59 Consider the program F_a in PGLEfa Example 8.56. This program is not well-formed. We project it into PGLEwf as in Table 8.9.

$F_a =$	$\texttt{Adding\_extra\_labels}(F_a)=$
$\pounds 1; +a; \#\#\pounds 2; +c; \#\#\pounds 1;$	$\pounds 1; +a; \#\#\pounds 7; +c; \#\#\pounds 1;$
$\pounds 2; +b; \#\#\pounds 4; +c; \#\#\pounds 1; \#\#\pounds 2;$	$\pounds 7; \pounds 2; +b; \#\#\pounds 4; +c; \#\#\pounds 1; \#\#\pounds 2;$
$\pounds 4; +d; \#\#\pounds 2; !$	$\pounds 4; +d; \#\#\pounds 2; !$

Table 8.9: Example of adding extra labels.

```
ProgF=

statement0;




L1:

statement1;
IF condition THEN
   GO L2;
END IF;
GO L1;


L2:

EXIT;
```

```
ProgM=
mainloop:
LOOP
   statement0;
   loop2:
   LOOP
      aux_loop1:
      LOOP
         loop1:
         LOOP
            statement1;
            IF condition THEN
               BREAK loop2;
            END IF;
            BREAK loop1;
         END loop1;
      END aux_loop1;
      BREAK loop2;
   END loop2;
   BREAK mainloop;
END mainloop;
```

Table 8.10: Example of applying the Elimination rules.

8.5.7 Replacing labels and gotos by loops with multi-level exits

Finally, in this section, we apply the Elimination rules of [83, 80] to replace all labels and gotos in a PGLEwf program by loops with multi-level exits. The whole program is put within a loop. Whenever the program terminates, the control of the program exits from that loop. Given a label $£l$ of the program to which there are only forward gotos, the Forward Elimination rule [80] is a transformation that eliminates all gotos by replacing $\#\#£l$ with break l and $£l$ with break $l; l\}$. The opening brace $l\{$ is inserted immediately before or after some label that precedes all the gotos $\#\#£l$. Similarly, given a label $£m$ to which there are only backward gotos, the Backward Elimination rule [83] replaces $\#\#£m$ with break m and $£m$ with $m'\{m\{$. The phrase break $m'; m\}; m'\}$ is inserted immediately before or after some label that stands behind all the gotos $\#\#£m$ in the program. The application of the Elimination rules can be seen in the following example.

Example 8.60 Consider the program `ProgF` in Example 8.55. Then the label `L1` in the program is of backward gotos, while the label `L2` is of forward gotos. We can eliminate goto statements in `ProgF` with the Elimination rules to the program `ProgM` given in Table 8.10.

To determine where to insert the opening braces $l\{$ and the phrases break $m'; m\}; m'\}$, we use the notions of *lower bounds* and *upper bounds* of labels (see Figure 8.6) in a

program.

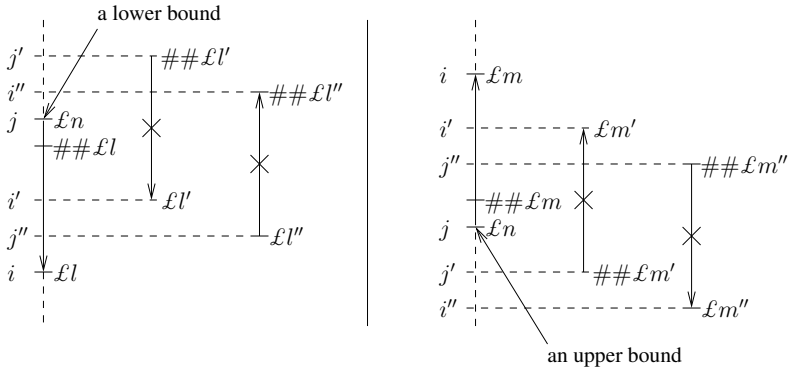

Figure 8.6: Example of lower bounds and upper bounds.

Definition 8.61 *Let $X = u_1; \ldots; u_k$ be a program in* PGLEwf*.*

1. *Let $u_i = \pounds l$ be a label to which there are only forward gotos. The position $j < i$ of a label instruction $\pounds n$ is a* **lower bound** *of $\pounds l$ if all positions j' of $\#\#\pounds l$ satisfy $j < j' < i$. Furthermore, there does not exist a label $u_{i'} = \pounds l'$ of a forward goto $u_{j'} = \#\#\pounds l'$ and there does not exist a label $u_{i''} = \pounds l''$ of a backward goto $u_{j''} = \#\#\pounds l''$ such that $j' < j < i' < i$ and $i'' < j < j'' < i$. If there is no such position j then $j = 0$. We denote* $\texttt{greatest\_lower\_bound}(\pounds l)$ *as the greatest lower bound of $\pounds l$.*

2. *Similarly, let $u_i = \pounds m$ be a label to which there are only backward gotos. The position $j < i$ of a label instruction $\pounds n$ is an* **upper bound** *of $\pounds l$ if all positions j' of $\#\#\pounds l$ satisfy $i < j' < j$. Furthermore, there does not exist a label $u_{i'} = \pounds m'$ of a backward goto $u_{j'} = \#\#\pounds m'$ and there does not exist a label $u_{i''} = \pounds m''$ of a forward goto $u_{j''} = \#\#\pounds m''$ such that $i < i' < j < j'$ and $i < j'' < j < i''$. If there is no such position j then $j = k + 1$. We denote* $\texttt{lowest\_upper\_bound}(\pounds m)$ *as the lowest upper bound of $\pounds m$.*

The transformation from PGLEwf to PGLM by applying the Elimination rules is given below.

Definition 8.62 *Let $X = u_1; \ldots; u_k$ be a program in* PGLEwf*, and m the maximum number in labels and gotos of X. Let $n = m + 2$. The transformation* `Applying_Elimination_rules` *from* PGLEwf *to* PGLM *is defined by*

$\texttt{Applying\_Elimination\_rules}(X) =$
$m + 1\{; \texttt{o\_braces}(\texttt{before}, 0); \psi_1(u_1); \ldots; \psi_k(u_k); \texttt{c\_braces}(\texttt{after}, k + 1);\ m + 1\}$

where the auxiliary functions $\psi_i(u_i)$ are defined by

$\psi_i(\pounds l) = \texttt{c\_braces}(\texttt{before}, i); \text{break}\, l; l\}; \texttt{c\_braces}(\texttt{after}, i); \texttt{o\_braces}(\texttt{after}, i)$
if $\texttt{fw}(\pounds l)$,
$\psi_i(\pounds l) = \texttt{c\_braces}(\texttt{before}, i); \texttt{o\_braces}(\texttt{before}, i); l + n\{; l\{; \texttt{o\_braces}(\texttt{after}, i)$
if $\texttt{bw}(\pounds l)$,
$\psi_i(\#\#\pounds l) = \text{break}\, l$,
$\psi_i(!) = \text{break}\, m + 1$,
$\psi_i(u) = u$ *otherwise.*

The auxiliary function $\texttt{o\_braces}(x, i)$ with $\texttt{pos}(\pounds l, X) = i$ and $x \in \{\texttt{before}, \texttt{after}\}$ determines a sequence of opening braces inserted immediately before or immediately after position i. Here $\texttt{pos}(\pounds l, X)$ determines the position of $\pounds l$ in X. If there is no such position, it returns 0. The opening braces given by $\texttt{o\_braces}(x, i)$ correspond to the closing braces that are replacements of the labels whose greatest lower bounds are i. Formally:

$$\texttt{o\_braces}(x, i) = i_1\{; \ldots; i_l\{$$

where $\pounds i_1, \ldots, \pounds i_l$ are labels of all forward goto instructions in X satisfying the condition that $\texttt{greatest\_lower\_bound}(\#\#\pounds i_j) = i$ for all $1 \leq j \leq l$ and $\texttt{pos}(\pounds i_{j+1}, X) < \texttt{pos}(\pounds i_j, X)$ for all $1 \leq j < l$. If $x = \texttt{before}$ and $\pounds l$ is a label of backward gotos then $\texttt{pos}(\pounds i_j, X) \leq \texttt{lowest\_upper\_bound}(\pounds l)$ for all $1 \leq j \leq l$. If $x = \texttt{after}$ and $\pounds l$ is a label of backward gotos then $\texttt{pos}(\pounds i_j, X) > \texttt{lowest\_upper\_bound}(\pounds l)$ for all $1 \leq j \leq l$.

Similarly, the auxiliary function $\texttt{c\_braces}(x, i)$ with $\texttt{pos}(\pounds l, X) = i$ and $x \in \{\texttt{before}, \texttt{after}\}$ determines a sequence of closing braces inserted immediately before or immediately after position i. These closing braces correspond to the opening braces that are replacements of the labels whose lowest upper bounds are i. Formally:

$$\texttt{c\_braces}(x, i) = \text{break}\, i_1 + n; i_1\}; i_1 + n\}; \ldots; \text{break}\, i_l + n; i_l\}; i_l + n\}$$

where $\pounds i_1, \ldots, \pounds i_l$ are labels of all backward goto instructions in X satisfying the condition that $\texttt{lowest\_upper\_bound}(\#\#\pounds i_j) = i$ for all $1 \leq j \leq l$ and $\texttt{pos}(\pounds i_{j+1}, X) < \texttt{pos}(\pounds i_j, X)$ for all $1 \leq j < l$. If $x = \texttt{before}$ and $\pounds l$ is a label of forward gotos then $\texttt{pos}(\pounds i_j, X) < \texttt{greatest\_lower\_bound}(\pounds l)$ for all $1 \leq j \leq l$. If $x = \texttt{after}$ and $\pounds l$ is a label of forward gotos then $\texttt{pos}(\pounds i_j, X) \geq \texttt{greatest\_lower\_bound}(\pounds l)$ for all $1 \leq j \leq l$.

Lemma 8.63 *Let X be a program in* PGLEwf. *Then the program* $\texttt{Applying\_Elimination\_rules}(X)$ *is well-formed.*

Proof: Omitted. □

Lemma 8.64 *The transformation* `Applying_Elimination_rules` *from* PGLEwf *to* PGLM *is correct.*

Proof: Similar to the proof of Theorem 8.52. □
Furthermore, the transformation `Applying_Elimination_rules` preserves structural equivalence since it does not rearrange or alter any instruction of the program except for the label, goto and termination instructions.

Example 8.65 Let X be the PGLEwf program defined in Example 8.59. We can transform X to a program in PGLM as in Table 8.65.

$X =$	`Applying_Elimination_rules`$(X) =$
	$8\{;$
$\pounds 1;$	$10\{; 1\{;$
	$7\{;$
$+a; \#\#\pounds 7; +c; \#\#\pounds 1;$	$+a;$ break $7; +c;$ break $1;$
	break $7;$
$\pounds 7;$	$7\};$
$\pounds 2;$	$11\{; 2\{;$
	$4\{;$
$+b; \#\#\pounds 4; +c; \#\#\pounds 1; \#\#\pounds 2;$	$+b;$ break $4; +c;$ break $1;$ break $2;$
	break $4;$
$\pounds 4;$	$4\};$
$+d; \#\#\pounds 2; !$	$+d;$ break $2;$ break $8;$
	break $11;$
	$2\}; 11\};$
	break $10;$
	$1\}; 10\};$
	$8\}$

Table 8.11: Example of applying the Elimination rules.

We have shown correctness and equivalence of the transformations `Reordering`, `Getting_rid_of_h2h_crossings`, `Adding_extra_labels` and `Applying_Elimination_rules`. Hence, labels and gotos can be eliminated from the setting of PGA by introducing loops and multi-level exits under behavioral equivalence.

Theorem 8.66 *The transformation* `Gte_without_variables` *from* PGLE *to* PGLM *is correct.*

Proof: This follows from Lemma 8.26, Lemma 8.29, Lemma 8.44, Theorem 8.52, Lemma 8.58 and Lemma 8.64. □
The inverse of the transformation `Gte_without_variables` can be given as follows.

Definition 8.67 *Let* $X = u_1; \ldots; u_k$ *be a program in* PGLM. *The transformation from* PGLM *to* PGLE *is defined by*

$$\texttt{Eliminating\_loops\_with\_multi\_level\_exits} = \psi_1(u_1); \ldots; \psi_k(u_k)$$

where the auxiliary functions $\psi_k(u_i)$ is determined by the following rewrite rules

$$\begin{array}{lcl} \psi_i(j\{) & = & \pounds i, \\ \psi_i(j\}n) & = & \#\#\pounds n; \pounds i, \\ \psi_i(\mathrm{break}\, j\, n) & = & \#\#\pounds n, \\ \psi_i(u) & = & u \textit{ otherwise.} \end{array}$$

This transformation together with the transformation `Gte_without_variables` indicates that the program notation PGLM with loops and multi-level exits is as expressive as the program notation PGLE with labels and gotos.

8.6 Restructuring Cobol programs

Removing all goto statements in a program is not always a good solution in the maintenance of legacy systems such as ones written in Cobol. As can be seen in the previous sections, the programs after eliminating all gotos are often very different from the original ones. This makes it difficult for a maintenance programmer to recognize and modify programs [40]. To circumvent this issue, in [96], Veerman presents a collection of transformation rules using the ASF+SDF Meta-Environment [60] to revitalize Cobol programs while preserving the track of the original ones. These transformation rules remove certain goto statements in order to extract business logic, and have been applied to several large industrial Cobol systems. However no formal correctness proofs of them have been provided due to a lack of time and the different semantics defined for Cobol as discussed in [65, 66]. In this section, we show that PGA provides a mathematical framework for reasoning about formal correctness proofs for these transformation rules. In order to do this, we first introduce a program notation namely CoPA (Cobol in program algebra) that interprets the programming language Cobol in PGA. We then define a restriction on CoPA in order to avoid the unexpected behaviors as studied in [98] (see Section 8.6.2). Finally, we formulate some transformation rules of [96] in CoPA and prove their correctness. We also hint at an automatable method to prove correctness for most of transformation rules in [96]. We note that the transformation rules presented in this chapter are obtained directly from the author of [96].

8.6.1 The program notation CoPA

In this section, we will introduce the program notation CoPA in the setting of PGA that is used to interpret the programming language Cobol.

First of all, we will explain a few basic concepts of Cobol. A Cobol program consists of four divisions: one for a description of the program, one for external dependencies, one for variable declarations, and one for the programming logic. The division for the logic is ordered similar to a text in a natural language. We will concentrate only on this division. It consists of sections, a section is divided into paragraphs, a paragraph consists of sentences, and a sentence consists of statements. Sections and paragraphs usually start with a label, which can be used for reference from elsewhere

in the program. Unlike other simple program notations in PGA, a program in Cobol may contain procedure calls using *out-of-line* PERFORM statements. There are two types of out-of-line PERFORM statements in Cobol:

- The first type of PERFORM statements is of the form PERFORM $\pounds l$. When the instruction PERFORM $\pounds l$ is performed, the control-flow jumps to the label $\pounds l$ and executes the paragraph belonging to this label. As soon as the control flow reaches the last statement of the paragraph, it is passed back to the statement following the PERFORM statement.

- The second type PERFORM statements is of the form PERFORM $\pounds i$ THRU $\pounds j$, assuming that the label $\pounds i$ precedes the label $\pounds j$ in the program. When this statement is performed, the control-flow jumps to label $\pounds i$ and executes the subsequence following $\pounds i$. As soon as the control flow reaches the end of the paragraph belonging to label $\pounds j$, the control-flow is passed back to the statement following the PERFORM THRU statement.

Our purpose is to show correctness of some transformation rules in [96]. Thus, for simplicity, we will construct CoPA with a least collection of primitive statements, a subset of the language, but rich enough for important applications.

The structure of a program in CoPA

A program in CoPA consists of paragraphs, and a paragraph consists of statements. We do not provide sections and sentences into CoPA, since they can be constructed as sequences of paragraphs or statements. In particular, a program in CoPA must have the following form:

$$\pounds l_1; u_1; \ldots; \pounds l_k; u_k$$

where u_i $(1 \leq i \leq k)$ are *paragraphs* containing no labels, and the numbers l_i contained in the labels are all distinct.

Primitive statements of the program notation CoPA

The program notation CoPA is constructed from the following primitive statements:

Basic instruction Basic instructions in Σ represent basis statements such as DISPLAY statements which are executed sequentially in Cobol.

Termination instruction Termination instruction ! represents the STOP RUN statement in Cobol.

Label instruction Labels given as in PGLE represent labels in Cobol. In CoPA, a label is always placed immediately before a paragraph.

Goto instruction Gotos given as in PGLE represent gotos in Cobol. Like in PGLE, for each goto in a CoPA program there must be an associated label.

Conditional statement Conditional statements given as in PGLS represent conditional statements in Cobol.

While-loop While-loops given as in PGLS represent *inline* PERFORM statements in Cobol. For instance, a phrase of inline PERFORM statements

```
PERFORM TEST BEFORE UNTIL condition
        statement
END PERFORM
```

in Cobol can be written in CoPA as $\mathbf{u}(-\texttt{condition}\{*;\texttt{statement};*\})$.

PERFORM statement The statement PERFORM $\pounds l_i$ THRU $\pounds l_j$ with $i \leq j$ represent out-of-line PERFORM statements in Cobol. The statement PERFORM $\pounds l$ in Cobol is represented as PERFORM $\pounds l$ THRU $\pounds l$ in CoPA.

Unit Finally, we allow the use of the *unit instruction operator* $\mathbf{u}(-)$ (see [82] for details) which takes a part of a program and wraps it into a unit of length one, in order to represent paragraphs of Cobol. These units are also used to keep track of instructions of a conditional statement or a while-loop. We impose the following restriction on CoPA: Every conditional statement and while-loop of a CoPA program is wrapped in a unit. For instance, consider the typical Cobol program below

```
L1.
   IF A>B
      DISPLAY '1'
   ELSE
      DISPLAY '2'
   END IF.
L2.
   PERFORM TEST BEFORE UNTIL NOT (A>B)
      DISPLAY '3'
   END PERFORM.
```

The program above can be written in CoPA as

$$\pounds 1; \mathbf{u}(+(A{>}B)\{;\texttt{DISPLAY '1';}\}\{;\texttt{DISPLAY '2';}\});$$
$$\pounds 2; \mathbf{u}(+(A{>}B)\{*;\texttt{DISPLAY '3';}*\}).$$

We note that a program in CoPA may contain an empty unit, denoted by $\mathbf{u}()$, to represent an empty paragraph. This unit takes place in the program but does not contain any statement. Furthermore, the condition of a conditional statement or an in-line PERFORM statement can be a *boolean expression* such as (condition1 AND condition2) or (condition1 OR condition2). For simplicity, we will not consider these expressions in this chapter.

8.6.2 The behavior of a program in CoPA

In this section, we impose a restriction on CoPA that avoids the unexpected behaviors explained in [98], and project programs in CoPA to PGLSu (PGLS with units), a sublanguage of CoPA without PERFORM statements, in order to determine behaviors of programs in CoPA.

Cobol mines

To define program behaviors in CoPA, one needs a precise semantics of PERFORM statements. However, out-of-line PERFORM statements in Cobol can be programmed in ways that lead to unexpected behaviors. We consider the following Cobol programs taken from [98].

```
P1::=L1.
         DISPLAY '1'
         PERFORM L2 THRU L3
         STOP RUN.
     L2.
         DISPLAY '2'
         PERFORM L3 THRU L4.
     L3.
         DISPLAY '3'.
     L4.
         DISPLAY '4'.
```

Program P1 has a *nested overlapping* PERFORM statement. In the paragraph following L1, a perform statement performs L2 through L3. Then in the paragraph following L2, a second perform statement references L3 through L4, thereby passing control through the exit of the first perform statement.

```
P2::=L1.
         DISPLAY '1'
         a=1
         PERFORM L3
         DISPLAY 'END'
         STOP RUN.
     L2.
         DISPLAY '2'.
     L3.
         DISPLAY '3'
         IF a=1 THEN
           a=0
           GO L2
         END IF.
```

Program P2 has an *external goto* that jumps out of a performed paragraph. The first time the paragraph following L3 is entered, variable a has value 1 and thus a goto

statement jumps to L2 before the end is reached. After L2, the control-flow can fall through to L3. The second time this paragraph is entered, a has value 0 and thus the end of paragraph is reached. When the perform statement in L1 terminates, the program displays 'END'.

```
P3::=L1.
         a=1
         PERFORM L2
         STOP RUN.
     L2.
         DISPLAY a
         IF a<3 THEN
           a=a+1
           PERFORM L2
         END IF.
         DISPLAY 'END'.
```

Program P3 has a *recursive* PERFORM *statement*. In L1, a is initialized with value 1 and L2 is performed. In L2, the value of a is displayed. Then, if the value of a is less than 3, it is increased by one, and the recursive perform of L2 is made. If $a \leq 3$, the program prints 'END'.

It is shown in [98] that with a number of different compilers on different platforms, the outputs of the three programs above are different. This means that the semantics of PERFORM statement differs between Cobol dialects. A code containing structures as in programs P1, P2 and P3 is regarded as *a Cobol mine*. The behaviors of programs containing Cobol mines are often unexpected. We will use the following definition to define Cobol mines in CoPA in a formal way.

Definition 8.68 *For a statement* PERFORM $£l_i$ THRU $£l_j$ *in a paragraph* u_k *of a* CoPA *program, we say that labels* $£l_n$ *for all* $i \leq n \leq j$ *are* **performed** *and are* **contained** *in this statement. In particular,* $£l_i$ *is the* **first** *performed label while* $£l_j$ *is the* **last** *performed label. Furthermore, for all* $i \leq n \leq j$ *paragraphs* u_n *(following labels* $£l_n$*) are* **performed**, *and label* $£l_k$ *(preceding paragraph* u_k*) is a* **predecessor** *of all labels* $£l_n$*. A label* $£l_j$ *is a* **descendant** *of a label* $£l_i$ *in the program if there is a sequence* $i_0, \ldots, i_n$ *with* $n \geq 0$ *such that* $i_0 = i$, $i_n = j$, *and* $£l_{i_k}$ *is a predecessor of* $£l_{i_{k+1}}$ *for all* $0 \leq k < n$.

Definition 8.69

1. *A* PERFORM *statement is* **recursive** *if it contains a performed label that is a descendant of itself.*

2. *A* PERFORM $£l_i$ THRU $£l_j$ *statement is* **nested overlapping** *if it contains a performed label* $£l_n$ *such that* $£l_j$ *is a descendant of* $£l_n$.

3. *A goto* $\#\#£l_n$ *in a performed paragraph* u_k *whose label contained in the statement* PERFORM $£l_i$ THRU $£l_j$ *is* **external** *if* $n \notin [i..j]$.

Hence, to avoid unexpected behaviors, we propose a restriction on programs in CoPA: Programs in CoPA may not contain recursive PERFORM statements, nested overlapping PERFORM statements, or external gotos in their performed paragraphs.

The proposed restriction on PERFORM statements avoids unexpected program behaviors in CoPA. However, it is still difficult to determine the behavior of a CoPA program because of the complex semantics of procedure calls. To circumvent this problem, we project CoPA to the program notation PGLSu (PGLS with units), a sub-language of CoPA by leaving out all PERFORM statements. The behavior of a CoPA program is determined by the behavior of its transformation in PGLSu.

In order to remove PERFORM statements in CoPA, we locate instructions of a program by sequence of natural numbers that keeps track of the relative position in a unit and that of all encompassing units. The empty sequence is written as ϵ, and "," is used as a separator between the natural numbers occurring in a sequence. We explain this location with the following example.

Example 8.70 Let X be a program in CoPA, and $[X]_\sigma$ denote the instruction at position σ of X.

$$X = \pounds 1; \mathbf{u}(a; \mathbf{u}(+b\{; \text{PERFORM } \pounds 2; \}\{; d; \})); \pounds 2; \mathbf{u}(g).$$

We locate all instructions of X as follows:

$$\begin{array}{lcl}
[X]_1 & = & \mathbf{u}(a; \mathbf{u}(+b\{; \text{PERFORM } \pounds 2; \}\{; d; \})), \\
[X]_{1,1} & = & a, \\
[X]_{1,2} & = & \mathbf{u}(+b\{; \text{PERFORM } \pounds 2; \}\{; d; \}), \\
[X]_{1,2,1} & = & +b\{, \\
[X]_{1,2,2} & = & \text{PERFORM } \pounds 2, \\
[X]_{1,2,3} & = & \}\{, \\
[X]_{1,2,4} & = & d, \\
[X]_{1,2,5} & = & \}, \\
[X]_2 & = & \mathbf{u}(g), \\
[X]_{2,1} & = & g.
\end{array}$$

In this location notation, labels are locationless. We assume that for each instruction $[X]_\sigma$ at position σ of a program X in CoPA or PGLSu there is a label instruction $\pounds X_\sigma$ (explicitly or hiddenly) placed before $[X]_\sigma$ in the program. Furthermore, this label X_σ can be represented by a sequence of natural numbers.

Definition 8.71 *Given two positions σ and δ of a program X in* CoPA. *We say that σ is a* **predecessor** *of δ if $\delta = \sigma, i$ for some $i \in \mathbb{N}$, or $[X]_\sigma = \text{PERFORM } \pounds l_i \text{ THRU } \pounds l_j$ and $\delta = n$ with $i \leq n \leq j$. Furthermore, δ is a* **descendant** *of σ if there is a sequence $\sigma_0, \ldots, \sigma_n$ such that $\sigma = \sigma_0$, $\sigma_n = \delta$, and σ_i is a predecessor of σ_{i+1} for all $0 \leq i < n$.*

Example 8.72 Position $(2, 1)$ of program X in Example 8.70 is a descendant of position $(1, 2, 2)$. Hence it is a descendant of the first position of X.

The restriction on CoPA (that does not allow recursive PERFORM statements) ensures that a position is not a descendant of itself.

The projection from CoPA to PGLSu

Let X be a program in CoPA. Wlog we assume that $X = £1; [X]_1; \ldots; £k; [X]_k$. The projection $\texttt{Perform\_removal}(X)$ from CoPA to PGLSu replaces a statement PERFORM $£i$ THRU $£i$ at position σ of the current program by the code fragment $£\sigma; v_i$, where the unit v_i is the same as $[X]_i$ except that the goto $\#\#£i$ in $[X]_i$ is replaced by the goto $\#\#£\sigma$ in v_i. Similarly, the projection $\texttt{Perform\_removal}(X)$ from CoPA to PGLSu replaces a statement PERFORM $£i$ THRU $£j$ with $i < j$ at position σ of the current program by the code fragment $\mathbf{u}(£\sigma, 1; v_i; \ldots; £\sigma, j-i+1; v_j)$, where the units v_l $(i \leq l \leq j)$ are the same as $[X]_l$ except that the gotos $\#\#£n$ in $[X]_l$ with $i \leq n \leq j$ are replaced by the goto $\#\#£\sigma, n-i+1$ in v_l. These replacements ensure that the control flow does not jump out of the procedure call. In order to keep track of the labels $£\sigma$ and $£\sigma, n-i+1$ with $i \leq n \leq j$, we use a sequence χ with $\chi = \chi_1; \ldots; \chi_k$, where k is the number of labels in X. Initially, $\chi = 1; \ldots; k$. Whenever a statement PERFORM $£i$ THRU $£i$ at position σ is considered, the value of χ_i is updated to σ. Similarly, whenever a statement PERFORM $£i$ THRU $£j$ with $i < j$ at position σ is considered, the values of χ_n for $i \leq n \leq j$ are updated to $\sigma, n-i+1$. This update is formally defined by:

$$\begin{array}{lcl} \texttt{update}(i, i, \sigma, \chi) & = & \chi_1; \ldots; \chi_{i-1}; \sigma; \chi_{i+1}; \ldots; \chi_k \\ \texttt{update}(i, j, \sigma, \chi) & = & \chi_1; \ldots; \chi_{i-1}; (\sigma, 1); \ldots; (\sigma, j-i+1); \chi_{j+1}; \ldots; \chi_k. \end{array}$$

The projection `Perform_removal` described above is given formally as follows.

Definition 8.73 *Let X be a program in* CoPA. *Wlog we assume that for every instruction at position σ of the current program there is a label $£\sigma$ placed hiddenly before it, and initially $X = u_1; \ldots; u_k$. The transformation* `Perform_removal` *from* CoPA *to* PGLSu *is defined by*

$$\texttt{Perform\_removal}(X) = \mathcal{P}_1^{1;\ldots;k}(u_1); \ldots; \mathcal{P}_k^{1;\ldots;k}(u_k)$$

where

$$\begin{array}{l} \mathcal{P}_\sigma^\chi(\text{PERFORM } £i \text{ THRU } £i) = \mathcal{P}_\sigma^{\texttt{update}(i,i,\sigma,\chi)}(u_i), \\ \mathcal{P}_\sigma^\chi(\text{PERFORM } £i \text{ THRU } £j) = \\ \qquad \mathbf{u}(\mathcal{P}_{\sigma,1}^{\texttt{update}(i,j,\sigma,\chi)}(u_i); \ldots; \mathcal{P}_{\sigma,j-i+1}^{\texttt{update}(i,j,\sigma,\chi)}(u_j)), \\ \mathcal{P}_\sigma^\chi(\mathbf{u}(s_1; \ldots; s_n)) = \mathbf{u}(\mathcal{P}_{\sigma,1}^\chi(s_1); \ldots; \mathcal{P}_{\sigma,n}^\chi(s_n)), \\ \mathcal{P}_\sigma^\chi(\#\#£i) = \#\#£\chi_i, \\ \mathcal{P}_\sigma^\chi(u) = u \text{ otherwise} \end{array}$$

with $i, j, l, m, n \in \mathbb{N}$.

The previous definition is illustrated in the following example.

Example 8.74 Consider the program X containing a PERFORM statement in the left-hand side of Table 8.12. The program Y in the right-hand side is obtained by removing PERFORM statements from X. We note that in these two programs the labels of paragraphs and the labels of units replacing PERFORM statements are displayed explicitly. Program X is formulated in CoPA as

```
X::=L1.
       PERFORM L3.




    L2.
       a.
    L3.
       IF b THEN
          GO L3
       ELSE
          c
       END IF.
```

```
Y::=L1.
       L1,1.
          IF b THEN
             GO L1,1
          ELSE
             c
          END IF.
    L2.
       a.
    L3.
       IF b THEN
          GO L3
       ELSE
          c
       END IF.
```

Table 8.12: Example of PERFORM removal.

$$X = \pounds 1; \mathbf{u}(\text{PERFORM}\ \pounds 3\ \text{THRU}\ \pounds 3); \pounds 2; \mathbf{u}(a); \pounds 3; \mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \})).$$

The PERFORM removal from X to Y is determined by

$$\begin{aligned}
&\texttt{Perform_removal}(X)\\
&\quad = \pounds 1; \mathcal{P}_1^{1;2;3}(\mathbf{u}(\text{PERFORM}\ \pounds 3\ \text{THRU}\ \pounds 3));\\
&\qquad \pounds 2; \mathcal{P}_2^{1;2;3}(\mathbf{u}(a));\\
&\qquad \pounds 3; \mathcal{P}_3^{1;2;3}(\mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \})))\\
&\quad = \pounds 1; \mathbf{u}(\mathcal{P}_{1,1}^{1;2;3}(\text{PERFORM}\ \pounds 3\ \text{THRU}\ \pounds 3));\\
&\qquad \pounds 2; \mathbf{u}(a);\\
&\qquad \pounds 3; \mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \}))\\
&\quad = \pounds 1; \mathbf{u}(\pounds 1, 1; \mathcal{P}_{1,1}^{1;2;(1,1)}(\mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \}))));\\
&\qquad \pounds 2; \mathbf{u}(a);\\
&\qquad \pounds 3; \mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \})).\\
&\quad = \pounds 1; \mathbf{u}(\pounds 1, 1; \mathbf{u}(+b\{; \#\#\pounds 1, 1; \}\{; c; \}));\\
&\qquad \pounds 2; \mathbf{u}(a);\\
&\qquad \pounds 3; \mathbf{u}(\mathbf{u}(+b\{; \#\#\pounds 3; \}\{; c; \})).
\end{aligned}$$

Since programs in CoPA do not contain recursive PERFORM statements, the transformation `Perform_removal` terminates. This is equivalent to the following lemma.

Lemma 8.75 *The transformation* `Perform_removal` *from* CoPA *into* PGLSu *is well-defined.*

The previous lemma suggests the definition of behaviors of programs CoPA as follows.

Definition 8.76 *The behavior* $|X|$ *of a program* X *in* CoPA *is given by*

$$|X| = |\texttt{Perform\_removal}(X)|.$$

Behavior extraction equations for PGLSu

The behavior of a PGLSu program is determined by the behavior extraction equations for PGLSu given below. We note that for a position σ of a PGLSu program X, the computation $\sigma \oplus 1$ denotes the position of the next instruction of $[X]_\sigma$ in the program.

Definition 8.77 *Let* X *be a program in* PGLSu. *The behavior* $|X|$ *of* X *is defined by* $|X| = |1, X|$, *where*

$$|\sigma, X| = \begin{cases} S & \text{if } [X]_\sigma =! \text{ or } \sigma \text{ is not a position in } X, \\ a \circ |\sigma \oplus 1, X| & \text{if } [X]_\sigma = a, \\ |\delta, X| & \text{if } [X]_\sigma = \#\#\pounds X_\delta, \\ |(\sigma, 1), X| & \text{if } [X]_\sigma = \mathbf{u}(U), \\ |\sigma \oplus 1, X| \trianglelefteq a \trianglerighteq |\delta \oplus 1, X| & \text{if } [X]_\sigma = +a\{\delta, \\ |\delta \oplus 1, X| \trianglelefteq a \trianglerighteq |\sigma \oplus 1, X| & \text{if } [X]_\sigma = -a\{\delta, \\ |\delta \oplus 1, X| & \text{if } [X]_\sigma = \}\delta\{, \\ |\sigma \oplus 1, X| & \text{if } [X]_\sigma = \}, \\ |\sigma \oplus 1, X| \trianglelefteq a \trianglerighteq |\delta \oplus 1, X| & \text{if } [X]_\sigma = +a\{*\delta, \\ |\delta \oplus 1, X| \trianglelefteq a \trianglerighteq |\sigma \oplus 1, X| & \text{if } [X]_\sigma = -a\{*\delta, \\ |\sigma \oplus 1, X| & \text{if } [X]_\sigma = \{*, \\ |\delta, X| & \text{if } [X]_\sigma = *\}\delta, \end{cases}$$

Inactive behavior D *will occur if the computation produces no result.*

Example 8.78 The behavior of program X in Example 8.74 is determined by $|X| = |\texttt{Perform\_removal}(X)| = P$ where

$$\begin{array}{lcl} P & = & P \trianglelefteq b \trianglerighteq Q, \\ Q & = & c \circ R, \\ R & = & a \circ T, \\ T & = & T \trianglelefteq b \trianglerighteq U, \\ U & = & c \circ S. \end{array}$$

8.6.3 Correctness of transformation rules for goto removal

The transformation rules of [96] are written in the ASF+SDF Meta-Environment [60, 37]. Intuitively, most of them preserve structural equivalence. Despite the understandable intuition of these transformation rules, it is quite difficult to prove their correctness because of the procedure calls via PERFORM statements. In this section, we formulate and prove correctness of some transformation rules in [96] in the setting of CoPA. These transformation rules remove certain types of goto statements in order to revitalize programs while preserving their original shapes. The correctness of the remaining rules can be shown in the same way.

```
conditions
⟹
left-hand side                    ⟶  right-hand side
```

Table 8.13: Notation for a transformation rule in the ASF+SDF Meta-Environment.

```
⟹
Goto_elimination(
  L1.                             ⟶  L1.
    S1                                   S1.
    GO L2.                               L2.
  L2.
,Last-performed-labels
)
```

Table 8.14: The `Goto_elimination` rule in the ASF+SDF Meta-Environment.

Transformation rules in the ASF+SDF Meta-Environment

In order to formulate the transformation rule eliminate-go in the setting of PGA, first of all, we give a brief introduction of transformation rules written in the ASF+SDF Meta-Environment. Here SDF stands for Syntax Definition Formalism and supports the definition of both lexical and context-free syntax (see [56]), while ASF stands for Algebraic Specification Formalism and supports the definition of conditional rewrite rules (see [18]). A transformation rule (or a rewrite rule) in the ASF+SDF Meta-Environment as illustrated in Table 8.13, consists of a left-hand side pattern and a right-hand side pattern with abstract and concrete syntax, and may have a condition. Whenever a transformation rule is applied on a source code, a parser generated by SDF will transform all the codes matching the left-hand side pattern to the codes matching the the right-hand side pattern from left to right, provided that their conditions are successfully evaluated.

The `Goto_elimination` rule

The `Goto_elimination` rule removes implicit goto statements standing immediately before their associated labels (see Table 8.14). We note that since CoPA programs do not contain Cobol mines, label L1 of the left-hand side pattern is not the last performed label otherwise GO L2 would be an external goto. We formulate the transformation rule `Goto_elimination` in the extension of CoPA with PGLSu, so that it can be applied on PGLSu programs as well.

Definition 8.79 *Let X be a program in* CoPA $\cup$ PGLSu. *Wlog we assume that for every instruction at position σ of the current program, there is a label £σ placed hiddenly before it, and initially $X = u_1; \ldots; u_k$. The* `Goto_elimination` *rule is defined by*

$$\texttt{Goto\_elimination}(X) = \mathcal{G}_1(u_1); \ldots; \mathcal{G}_k(u_k)$$

where

$$\begin{array}{lcl} \mathcal{G}_{\delta,i}(\mathbf{u}(s;\#\#\pounds\delta,i+1)) & = & \mathbf{u}(\mathcal{G}_{\delta,i,1}(s)), \\ \mathcal{G}_{\sigma}(\mathbf{u}(v_1;\ldots;v_n)) & = & \mathbf{u}(\mathcal{G}_{\sigma,1}(v_1);\ldots;\mathcal{G}_{\sigma,n}(v_n)), \\ \mathcal{G}_{\sigma}(u) & = & u \textit{ otherwise.} \end{array}$$

One can see that the `Goto_elimination` rule terminates.

Lemma 8.80 *The transformation rule* `Goto_elimination` *is well-defined.*

Furthermore, if a program does not contain PERFORM statements then the correctness of this transformation rule on this program is straightforward.

Lemma 8.81 *The transformation rule* `Goto_elimination` *on* PGLSu *is correct.*

Finally, the composition result of two transformations `Perform_removal` and `Goto_elimination` on CoPA does not depend on their order, i.e.:

Theorem 8.82 *Let X be a* CoPA *program defined as in Definition 8.73 and Definition 8.79. Then*

$$\texttt{Perform\_removal}(\texttt{Goto\_elimination}(X)) = \texttt{Goto\_elimination}(\texttt{Perform\_removal}(X)).$$

Proof: Let $Y = \texttt{Goto\_elimination}(X)$ and $Z = \texttt{Perform\_removal}(X)$. We show that $\texttt{Perform\_removal}(Y) = \texttt{Goto\_elimination}(Z)$. By Definition 8.73 and Definition 8.79, let

$$\begin{array}{rcl} Y & = & \mathcal{G}_1([X]_1);\ldots;\mathcal{G}_k([X]_k), \\ Z & = & \mathcal{P}_1^{1;\ldots;k}([X]_1);\ldots;\mathcal{P}_k^{1;\ldots;k}([X]_k), \\ \texttt{Perform\_removal}(Y) & = & P_1^{1;\ldots;k}(\mathcal{G}_1([X]_1));\ldots;P_k^{1;\ldots;k}(\mathcal{G}_k([X]_k)), \\ \texttt{Goto\_elimination}(Z) & = & G_1(\mathcal{P}_1^{1;\ldots;k}([X]_1));\ldots;G_k(\mathcal{P}_1^{1;\ldots;k}([X]_k)), \end{array}$$

where G_σ and P_σ^χ are defined as $\mathcal{G}_\sigma$ and $\mathcal{P}_\sigma^\chi$ in Definition 8.79 and Definition 8.73 for programs Y and Z, respectively. For a position γ of X and a position σ of Y and Z, we prove by induction on the descendants of σ that

$$G_\sigma(\mathcal{P}_\sigma^\chi([X]_\gamma)) = P_\sigma^\chi(\mathcal{G}_\gamma([X]_\gamma)).$$

where γ, σ and χ satisfy following condition: If $\gamma = i,\gamma'$ then $\sigma = \sigma',i',\gamma'$ and $\chi_i = \sigma',i'$ for some sequences γ',σ' and natural numbers i,i'. Furthermore, if $[X]_\gamma = \#\#\pounds i \pm n$ then $\chi_{i\pm n} = \sigma',i' \pm n,\gamma'$ for some natural number n. This condition is to avoid external gotos in a program. We assume for all descendants σ' of σ, positions γ' of X and sequences χ' satisfying the condition above that $G_{\sigma'}(\mathcal{P}_{\sigma'}^{\chi'}([X]_{\gamma'})) = P_{\sigma'}^{\chi'}(\mathcal{G}_{\gamma'}([X]_{\gamma'}))$. We consider the following possibilities:

1. $[X]_\gamma = u$ with $u \in \Sigma$. Then $G_\sigma(\mathcal{P}^\chi_\sigma(u)) = P^\chi_\sigma(\mathcal{G}_\gamma(u)) = u$.
2. $[X]_\gamma = \#\#\pounds i$. Then $G_\sigma(\mathcal{P}^\chi_\sigma(\#\#\pounds i)) = P^\chi_\sigma(\mathcal{G}_\gamma(\#\#\pounds i)) = \#\#\pounds\chi_i$.
3. $\gamma = i$, $[X]_i = \mathbf{u}(s; \#\#\pounds i + 1)$ and $\sigma = \delta, i'$. Then $\chi_{i+1} = \delta, i' + 1$ since X does not contain external gotos. By Definition 8.73, Definition 8.79 and the induction hypothesis,

$$\begin{array}{rcl} G_{\delta,i'}(\mathcal{P}^\chi_{\delta,i'}(\mathbf{u}(s; \#\#\pounds i + 1))) & = & G_{\delta,i'}(\mathbf{u}(\mathcal{P}^\chi_{\delta,i',1}(s); \#\#\pounds\delta, i' + 1)) \\ & = & \mathbf{u}(G_{\delta,i',1}(\mathcal{P}^\chi_{\delta,i',1}(s))) \\ & = & \mathbf{u}(P^\chi_{\delta,i',1}(\mathcal{G}_{i,1}(s))) \\ & = & P^\chi_{\delta,i}(\mathcal{G}_i(\mathbf{u}(s; \#\#\pounds i + 1))) \end{array}$$

4. $[X]_\gamma = \mathbf{u}(v_1; \ldots; v_n)$. Then

$$\begin{array}{rcl} G_\sigma(\mathcal{P}^\chi_\sigma(\mathbf{u}(v_1; \ldots; v_n))) & = & \mathbf{u}(G_{\sigma,1}(\mathcal{P}^\chi_{\sigma,1}(v_1)); \ldots; G_{\sigma,n}(\mathcal{P}^\chi_{\sigma,n}(v_n))) \\ & = & \mathbf{u}(P^\chi_{\sigma,1}(\mathcal{G}_{\gamma,1}(v_1)); \ldots; P^\chi_{\sigma,n}(\mathcal{G}_{\gamma,n}(v_n))) \\ & = & P^\chi_\sigma(\mathcal{G}_\gamma(\mathbf{u}(v_1; \ldots; v_n))). \end{array}$$

5. $[X]_\gamma$ = PERFORM $\pounds i$ THRU $\pounds i$. It follows from Definition 8.73, Definition 8.79 and the induction hypothesis that

$$\begin{array}{rcl} G_\sigma(\mathcal{P}^\chi_\sigma([X]_\gamma)) & = & G_\sigma(\mathcal{P}^{\mathtt{update}(i,i,\sigma,\chi)}_\sigma([X]_i)) \\ & = & P^{\mathtt{update}(i,i,\sigma,\chi)}_\sigma(\mathcal{G}_i([X]_i)) \\ & = & P^{\mathtt{update}(i,i,\sigma,\chi)}_\sigma([X]_i) \\ & = & P^\chi_\sigma(\text{PERFORM } \pounds i \text{ THRU } \pounds i) = P^\chi_\sigma(\mathcal{G}_\gamma([X]_\gamma)). \end{array}$$

6. $[X]_\gamma$ = PERFORM $\pounds i$ THRU $\pounds j$. Similar to the previous case, we also have

$$G_\sigma(\mathcal{P}^\chi_\sigma([X]_\gamma)) \quad = \quad P^\chi_\sigma(\mathcal{G}_\gamma([X]_\gamma)).$$

Hence $G_i(\mathcal{P}^{1;\ldots;k}_i([X]_i)) = P^{1;\ldots;k}_i(\mathcal{G}_i([X]_i))$ for all $1 \leq i \leq k$. This implies that $\mathtt{Goto\_elimination}(Z) = \mathtt{Perform\_removal}(Y)$. □
Therefore:

Theorem 8.83 *The transformation rule* `Goto_elimination` *is correct.*

Proof: Let X be a CoPA defined as in Definition 8.79. It follows from Definition 8.76, Lemma 8.81 and Theorem 8.82 that

$$\begin{array}{rcl} |\mathtt{Goto\_elimination}(X)| & = & |\mathtt{Perform\_removal}(\mathtt{Goto\_elimination}(X)| \\ & & \text{(by Definition 8.76)} \\ & = & |\mathtt{Goto\_elimination}(\mathtt{Perform\_removal}(X))| \\ & & \text{(by Theorem 8.82)} \\ & = & |\mathtt{Perform\_removal}(X)| \quad \text{(by Lemma 8.81)} \\ & = & |X| \quad \text{(by Definition 8.76).} \end{array}$$

```
⟹
Loop_reformulation(
  L1.                    ⟶L1.
    IF condition THEN        PERFORM TEST BEFORE UNTIL NOT condition
      S1                       S1
      GO L1                  END PERFORM
    ELSE                     S2
      S2                     S3.
    END IF
    S3.
)
```

Table 8.15: The `Loop_reformulation` rule in the ASF+SDF Meta-Environment.

□

We note that the transformation rule `Goto_elimination` indeed preserves structural equivalence (see Section 8.1.1), since it does not rearrange any other instructions of a program.

The `Loop_reformulation` rule

The `Loop_reformulation` rule eliminates *local* gotos while preserving behavioral equivalence (see Table 8.15). Like the `Goto_elimination` rule, we define the `Loop_reformulation` rule in the extension of CoPA with PGLSu, so that it can be applied on programs in PGLSu as well.

Definition 8.84 *Let X be a program in* CoPA $\cup$ PGLSu. *Wlog we assume that for every instruction at position σ of the current program there is a label $£\sigma$ placed hiddenly before it, and initially $X = u_1; \ldots; u_k$. The* `Loop_reformulation` *rule is defined by*

$$\texttt{Loop\_reformulation}(X) = \mathcal{L}_1(u_1); \ldots; \mathcal{L}_k(u_k)$$

where

$$\begin{array}{ll} \mathcal{L}_\sigma(\mathbf{u}(\mathbf{u}(\pm a\{; s_1; \#\#£\sigma; \}\{; s_2; \};); s_3)) & = \mathbf{u}(\mathbf{u}(\pm a\{*; \mathcal{L}_\alpha(s_1); *\};); \mathcal{L}_\beta(s_2); \mathcal{L}_\gamma(s_3)), \\ \mathcal{L}_\sigma(\mathbf{u}(v_1; \ldots; v_n)) & = \mathbf{u}(\mathcal{L}_{\sigma,1}(v_1); \ldots; \mathcal{L}_{\sigma,n}(v_n)), \\ \mathcal{L}_\sigma(u) & = u \textit{ otherwise} \end{array}$$

with $\alpha = \sigma, 1, 2$; $\beta = \sigma, 1, 5$ and $\gamma = \sigma, 2$ the positions of the units s_1, s_2 and s_3 in X.

Since the number of gotos decreases by one after each step, the transformation rule `Loop_reformulation` terminates successfully.

Lemma 8.85 *The transformation rule* `Loop_reformulation` *is well-defined.*

Furthermore, if a program does not contain PERFORM statements then the correctness of this transformation rule on this program is straightforward.

Lemma 8.86 *The transformation rule* `Loop_reformulation` *on* PGLSu *is correct.*

Finally like the `Goto_elimination` rule, the composition result of the two transformations `Perform_removal` and `Loop_reformulation` does not depend on their order. In other words, we can first remove all PERFORM statements, then reformulate the program, or we can transform the program conversely.

Theorem 8.87 *Let* X *be a* CoPA *program Definition 8.73 and Definition 8.84. Then*

$$\texttt{Perform\_removal}(\texttt{Loop\_reformulation}(X)) = \texttt{Loop\_reformulation}(\texttt{Perform\_removal}(X)).$$

Proof: Similar to the proof of Theorem 8.82. □
The previous results imply that:

Theorem 8.88 *The transformation rule* `Loop_reformulation` *is correct.*

Proof: Similar to the proof of Theorem 8.83, this follows from Definition 8.76, Lemma 8.86 and Theorem 8.87. □

The `Switch_paragraph` rule

The `Switch_paragraph` rule transforms repeatedly from left to right any code fragment of a program having the form as the left-hand side pattern in Table 8.16 to the code fragment of the form as the right-hand side pattern. In this figure, the goto statement GO L2 is eliminated, since it becomes an implicit goto after exchanging labels L2 and L3 together with their paragraphs. The condition imposed on this transformation rule is that labels L4 and L5 are different from labels L2 and L3, respectively. Furthermore, in order to avoid Cobol mines after the transformation, label L2 is not the first label of a PERFORM statement and label L3 is not the last label of a PERFORM statement in the original program. We note that since programs in CoPA do not contain Cobol mines, label L3 is not the first label of a PERFORM statement.

Let `First_performed_labels` and `Last_performed_labels` be the sets of first performed labels and last performed labels, respectively. We use an auxiliary transformation called `Sp_gte` to define the `Switch_paragraph` rule in CoPA.

Definition 8.89 *Let* X *be a program in* CoPA*. Wlog we assume that* $X = £1; u_1; \ldots; £k; u_k$*. The transformation* `Sp_gte` *is defined as follows. Let* i *be the first position of* X *satisfying that the code fragment* X_i^{i+2} *is of the form as the*

```
L4 != L2,
L5 != L3,
L2 in First-performed-labels == false,
L3 in Last-performed-labels == false
⟹
Switch_paragraph(
  L1.                              ⟶    L1.
    S1                                     S1
    GO L4.                                 GO L4.
  L2.                                    L3.
    S2                                     S3.
    GO L5.                               L2.
  L3.                                      S2
    S3                                     GO L5.
    GO L2.
,Last-performed-labels
)
```

Table 8.16: The `Switch_paragraph` rule in the ASF+SDF Meta-Environment.

left-hand side pattern in Table 8.16, where $X_i^j = £i; u_i; \ldots; £j; u_j$ $(i \leq j)$*. If there is no such position* i *then*

$$\texttt{Sp\_gte}(X) = X$$

otherwise let

$$X_i^{i+2} = £i; \mathbf{u}(s_1; \#\#£m); £i+1; \mathbf{u}(s_2; \#\#£n); £i+2; \mathbf{u}(s_3; \#\#£i+1)$$

with $i+1 \notin$ `First_performed_labels`*,* $i+2 \notin$ `Last_performed_labels`*,* $m \neq i+1$ *and* $n \neq i+2$*. Then*

$$\texttt{Sp\_gte}(X) = X_1^{i-1}; £i; \mathbf{u}(s_1; \#\#£m); £i+2; \mathbf{u}(s_3); £i+1; \mathbf{u}(s_2; \#\#£n); X_{i+3}^k.$$

Definition 8.90 *The* `Switch_paragraph` *rule is obtained by applying the transformation* `Sp_gte` *repeatedly from left to right.*

Since the number of gotos decreases by one after applying the transformation `Sp_gte`, the system-specific rule terminates, i.e.:

Lemma 8.91 *The* `Switch_paragraph` *rule is well-defined.*

We note that unlike the `Goto_elimination` rule and the `Loop_reformulation` rule, the composition result of the `Perform_removal` and the `Switch_paragraph` rule in different orders is significant as can be seen in the following example.

Example 8.92 Consider the program below:

```
L0.
     PERFORM L1 THRU L2.
L1.
     GO L1.
L2.
     GO L1.
L3.
     a
     GO L2.
```

Then the left-hand side program of Table 8.17 is obtained from this program by first applying the `Switch_paragraph` rule and then removing PERFORM statements, while the right-hand the program is obtained conversely. It is obvious that these two

```
L0.                    |    L0.
     L1'.              |         L1'.
         GO L1'.       |             GO L1'.
     L3'.              |         L2'.
         a.            |             GO L1'.
     L2'.              |    L1.
         GO L1'.       |         GO L1.
 L1.                   |    L3.
     GO L1.            |         a.
 L3.                   |    L2.
     a.                |         GO L1.
 L2.                   |
     GO L1.            |
```

Table 8.17: Example of applying the `Perform_removal` and the `Switch_paragraph` rule.

programs are not the same.

Lemma 8.93 *The transformation* `Sp_gte` *is correct.*

Proof: Let Y be obtained from program X in CoPA by applying the transformation `Sp_gte` given as below.

$$\begin{array}{rcl} X & = & X_1^{i-1}; \pounds i; \mathbf{u}(s_1; \#\#\pounds m); \pounds i+1; \mathbf{u}(s_2; \#\#\pounds n); \pounds i+2; \mathbf{u}(s_3; \#\#\pounds i+1); X_{i+3}^k, \\ Y & = & X_1^{i-1}; \pounds i; \mathbf{u}(s_1; \#\#\pounds m); \pounds i+2; \mathbf{u}(s_3); \pounds i+1; \mathbf{u}(s_2; \#\#\pounds n); X_{i+3}^k \end{array}$$

Let $X' = \texttt{Perform\_removal}(X)$ and $Y' = \texttt{Perform\_removal}(Y)$. We show that $|X'| = |Y'|$. To do this, we define a binary relation $\sim$ between positions of X' and Y' inductively as follows.

1. $j \sim j$ for all $1 \leq j \leq k$, $j \notin \{i+1, i+2\}$.

2. $i+1 \sim i+2$.

3. $i+2,1 \sim i+1,1$.

4. $i+2,2 \sim i+2$.

5. If $\sigma \sim \delta$ and $[X']_\sigma$ and $[Y']_\delta$ are the replacements of the same unit $u = \mathbf{u}(t_1;\ldots;t_p)$ of X and Y then $\sigma, j \sim \delta, j$ for all $1 \leq j \leq p$.

6. If $\sigma \sim \delta$ and $[X']_\sigma$ and $[Y']_\delta$ are the replacements of the statement PERFORM $\pounds p$ THRU $\pounds q$ then by Definition 8.6.3, $p \neq i+1$ and $q \neq i+2$. Furthermore, $p \neq i+2$ otherwise $\#\#\pounds i+1$ is an external goto of X. We consider the following cases:

   (a) $q < i+1$ or $p > i+2$. We define that $\sigma, j \sim \delta, j$ for $1 \leq j \leq q-p+1$.

   (b) $p < i+1 < i+2 < q$. We define that

      i. $(\sigma, j) \sim (\delta, j)$ for all $1 \leq j \leq q-p+1$, $j \notin \{i+2-p, i+3-p\}$. Here $[X']_{\sigma,i+2-p}$ and $[X']_{\sigma,i+3-p}$ are the replacements of $[X]_{i+1} = \mathbf{u}(s_2;\#\#\pounds n)$ and $[X]_{i+2} = \mathbf{u}(s_3;\#\#\pounds l_{i+2})$, while $[Y']_{\delta,i+2-p}$ and $[Y']_{\delta,i+3-p}$ are the replacements of $[Y]_{i+1} = \mathbf{u}(s_3)$ and $[Y]_{i+2} = \mathbf{u}(s_2;\#\#\pounds n)$.

      ii. $\sigma, i+2-p \sim \delta, i+3-p$.

      iii. $\sigma, i+3-p, 1 \sim \delta, i+2-p, 1$.

      iv. $\sigma, i+3-p, 2 \sim \delta, i+3-p$.

   (c) $q = i+1$. Since X does not contain external gotos, the units $[X]_p, \ldots, [X]_{i+1}$ do not contain any goto $\#\#\pounds i+2$. Thus the units $[Y]_p, \ldots, [Y]_i, [Y]_{i+2}$ do not contain any goto $\#\#\pounds i+2$ whose associated label precedes $[Y]_{i+1}$. This implies that the units $[Y']_{\delta,1}, \ldots, [Y]_{\delta,i+1-p}, [Y']_{\delta,i+3-p}$ do not contain any goto $\#\#\pounds\delta, i+3-p$ whose associated label precedes $[Y']_{\delta,i+2-p}$. Hence $[Y']_{\delta,i+2-p}$ is a dead instruction in Y'. We then define that $\sigma, j \sim \delta, j$ for all $1 \leq j \leq i+1-p$ and $\sigma, i+2-p \sim \delta, i+3-p$.

Let $P_\sigma = |\sigma, X'|$ and $Q_\delta = |\delta, Y'|$ for all positions σ of X' and δ of Y'. Similar to the proof of Theorem 8.52, we can derive that $P_\sigma = Q_\delta$ if $\sigma \sim \delta$. Therefore $|X'| = |Y'|$. □

The previous result implies that:

Theorem 8.94 *The* `Switch_paragraph` *rule is correct.*

Proof: This follows from Definition 8.90 and Lemma 8.93. □

We note that the `Switch_paragraph` rule indeed preserves flow-graph equivalence (see Section 8.1.1) since the target program and its original have the same flowchart.

8.6.4 An automatable method for proving correctness of transformation rules

In the previous section, we have proved by hand correctness of some transformation rules in [96] for goto elimination in the setting of PGA. This shows that PGA provides a mathematical framework for reasoning about correctness and equivalence of transformation rules for restructuring Cobol programs. Although one can learn PGA quickly, it still requires time, effort and theoretical skills to prove correctness of all of them. In this last section, we will suggest an automatable method [105, 68] for formally proving correctness of these transformation rules.

We observe that the correctness of transformation rules in [96] is straightforward if the programs do not contain PERFORM statements. Hence for a given transformation rule `Transf_rule` of [96], if we can prove that for every program X in CoPA the following equation holds

$$\texttt{Perform\_removal}(\texttt{Transf\_rule}(X)) = \texttt{Transf\_rule}(\texttt{Perform\_removal}(X)) \quad (8.1)$$

then the correctness proof of this transformation rule follows. It is because the program after the transformation behaves the same as the original one, i.e.:

$$\begin{aligned} |\texttt{Transf\_rule}(X)| &= |\texttt{Perform\_removal}(\texttt{Transf\_rule}(X))| \quad \text{(by definition)} \\ &= |\texttt{Transf\_rule}(\texttt{Perform\_removal}(X))| \quad \text{(by (8.1))} \\ &= |\texttt{Perform\_removal}(X)| \quad \text{(straightforward)} \\ &= |X| \quad \text{(by definition)} \end{aligned}$$

Intuitively, Equation (8.1) holds for most transformation rules in [96], for instance the `Goto_elimination` rule and the `Loop_reformulation` rule. Its proof is not straightforward and requires induction as seen in the proof of Theorem 8.82. However, it can be attained automatically by using existing theorem provers such as PVS (a Prototype Verification System) [77]. This method guarantees that these transformation rules are correct. Furthermore, it can save us time and effort in providing formal correctness proofs for all transformation rules in [96]. We note that for certain transformation rules of [96] that do not satisfy (8.1) such as the `Switch_paragraph` rule, their correctness can be proved by hand.

8.7 Conclusion

We have studied the correctness and equivalence of various standard algorithms and transformation rules for goto removal in the setting of PGA.

First of all, to eliminate gotos using additional variables, we have shown that the algorithm of Cooper [41] for proving the Folk theorem is correct under behavioral equivalence with respect to additional variables. This equivalence is finer than the input-output equivalence used in the literature [55, 33] when dealing with goto removal using additional boolean variables. We note that one can achieve a similar result for the approach of Böhm and Jacopini [33].

To eliminate gotos without the use of additional variables we have proposed a technique that removes head-to-head crossings in the programs. Subsequently using the results of Peterson et al. and Ramshaw, we have proven that gotos can be eliminated without using additional variables by introducing loops with multi-level exits under behavioral equivalence.

By assuming that Cobol programs do not produce the unexpected behaviors as studied in [98], we have proved correctness and equivalence of some transformation rules for restructuring Cobol programs given in [96]. We also suggested an automatable method for formally proving correctness of these transformation rules.

Our work shows that gotos can be eliminated in the setting of PGA by the use of additional variables under behavioral equivalence with respect to these variables, or by introducing loops with multi-level exits under behavioral equivalence. Furthermore, PGA creates a systematic and mathematical framework for reasoning about and classifying correctness and equivalence of standard algorithms and transformation rules for goto removal and the restructuring of programs. We hereby show that PGA's mechanism can explain goto elimination with mathematical rigor to a larger public.

Chapter 9

Summary

This thesis is about semantics and applications of process and program algebra, which are algebraic frameworks for formalizing and analyzing system behaviors and computer programs. We have given solutions to two open questions raised in [29] on orthogonal bisimulation, a semantic equivalence that deals with abstraction in process algebra. Furthermore, we have studied the semantics of thread algebra, a process algebra for the semantics of recent object-oriented and multi-threaded programming languages such as C# and Java. In the study of program algebra itself, we have explored the expressiveness of extensions of program algebra with conditional statements and while-loops. In the applications of thread and program algebra, we have shown that thread algebra can be applied to define various notions of noninterference in language-based security. Also, program algebra can be used for proving correctness of algorithms and transformation rules for goto removal.

Chapter 2 defines a trace characterization of orthogonal bisimulation called the compression structure of a process. This definition depends on a notion of compression content of the traces of a process. That is, the traces of the process from which all internal actions are removed, and if a trace ends in an internal action then its compression content is extended with the internal action symbol. We have shown that this notion of compression structure characterizes orthogonal bisimilarity in the same way as the branching structure in [51] characterizes branching bisimilarity. In particular, two processes have the same compression structure if and only if they are orthogonally bisimilar.

Chapter 3 studies the complexity of deciding orthogonal bisimulation in a finite state transition system. We have presented an algorithm for deciding orthogonal bisimulation. This algorithm is based on the well-known algorithm for deciding branching bisimulation [52] given by Groote and Vaandrager [54]. We have shown that the complexity of our algorithm remains the same as the one of Groote and Vaandrager's algorithm. Thus if n is the number of states, and m the number of transitions then it takes $O(n(m+n))$ time to decide orthogonal bisimilarity on a finite labeled transition system, using $O(m+n)$ space.

Chapter 4 gives a structural operational semantics (SOS) [81] for thread algebra.

We have presented transition rules for the strategic interleaving operators of [23] and shown that bisimulation equivalence defined by the new SOS characterizes the equality induced by the axioms of thread algebra.

Chapter 5 presents a denotational semantics [11] for thread algebra. We have followed the metric methodology of de Bakker and Zucker [11] to turn the domains of thread algebra into complete metric spaces. We have shown that the complete metric space consisting of projective sequences is an appropriate domain for thread algebra. More precisely, in the setting of single threads, we have proved that this domain represents infinite threads in a unique way. Furthermore, it is compatible with the domain based on complete partial orders (cpo's) of thread algebra [16]. Moreover, it deals naturally with abstraction in comparison with the domain consisting of Cauchy sequences. We also have proved that the specification of a regular thread determines a unique solution in this domain. In the setting of multi-threads, it has been shown that this domain can be extended with the strategic interleaving operators of [23] in a natural way, while the domain based on cpo's cannot. We also have proposed a particular interleaving strategy with respect to abstraction for thread algebra.

Chapter 6 explores the expressiveness of the extensions PGAuc and PGAucw of program algebra (PGA) with units, conditional statements and while-loops with respect to the lazy projection semantics proposed in [22]. First of all, we have presented a projection from PGAuc (PGA extended with units and conditional statements) to PGAu (PGA extended with units) [82]. The projection from PGAuc to PGA is a composition of our projection and the projection from PGAu to PGA defined in [82]. Next, we have shown that PGA with while-loops yields non-regular behaviors in certain cases. Under the restriction that consecutive occurrences of while-loops are forbidden, we have given a projection from PGAucw (PGAuc extended with while-loops) to PGLBu, a variant of PGAu with backward jumps [82]. The projection from PGAucw to PGA, therefore, is a composition of this projection and the projection from PGLBu to PGA described in [82]. The existence of our projections shows that conditional statements and while-loops, while allowing for a flexible style of programming, are not needed as primitive instructions in terms of expressiveness. Finally, these projections can be used to study PGA itself.

Chapter 7 shows that thread algebra, the semantics of program algebra, creates a process-algebraic framework for formalisation and analysis of security properties in language-based security [88]. We have interpreted various standard notions of noninterference, an important security property that characterizes systems whose execution does not reveal secret information. In particular, we have defined termination-insensitive noninterference (TINI), termination-sensitive noninterference (TSNI) and timing-sensitive noninterference (TISNI) in thread algebra. We have proved soundness for these noninterference properties, meaning that if a thread satisfies one of these properties then it satisfies the noninterference property given by Goguen and Meseguer [53]. Furthermore, we have shown that our approach accepts all secure programs that are accepted by the current approaches based on type systems [99, 91] of language-based security. In the setting of multi-threads, it has been proved that TISNI composes with respect to the cyclic interleaving operator, a basic interleaving strategy of [23]. In order to preserve compositionality for TINI and TSNI, we have

proposed a particular interleaving strategy for thread algebra that invokes the rotation of the thread vector only in the case that the current action is not persistent. By assuming that high actions are persistent, the analysis can be made compositional. For checking our noninterference properties, one can use existing tools such as [48, 49, 31] for deciding process-equivalence to develop our security checkers. We hereby have shown that thread algebra it is suitable as a process-algebraic framework for considering security properties in unstructured and multi-threaded programming languages. Hence, previous work on security for sequential and multi-threaded programming languages can be reconsidered in this framework.

Chapter 8 shows that program algebra provides a mathematical and systematic framework for reasoning about the correctness and equivalence of algorithms and transformation rules for goto removal [45]. We have considered three existing classes of goto removal: removing all gotos using additional variables, removing all gotos without additional variables, and removing certain types of gotos for knowledge extraction. In the first class, we have proved correctness for the algorithm given by Cooper [41] under behavioral equivalence with respect to additional boolean variables. This equivalence is finer than the input-output equivalence used in [41] and other techniques [33, 41, 76, 3, 39, 72] dealing with goto removal using additional variables. We note that one can achieve a similar result for the approach of Böhm and Jacopini [33]. In the second class, both well-known approaches of Peterson et al. [80] and Ramshaw [83] require a condition that programs are reducible or free of head-to-head crossings. Hence, to remove all gotos without the use of additional variables, we have proposed a technique to get rid of head-to-head crossings in programs and subsequently used the results of Peterson et al. and Ramshaw. We have shown that our algorithm is correct under behavioral equivalence, an analogous notion of path equivalence used in [80, 83]. Finally, in the class of removing certain types of gotos in order to extract knowledge embedded in legacy software systems, we have provided formal correctness proofs for some transformation rules to restructure Cobol programs in a real-life application [96]. We also have suggested an automatable method to prove correctness for all transformation rules in [96]. Therefore, we have shown that gotos can be eliminated in the setting of program algebra. Furthermore, we have explained goto removal with mathematical rigour.

Samenvatting*

Dit proefschrift gaat over semantiek en toepassingen van proces- en programma-algebra, algebraïsche kaders voor formalisering en analyse van systeemgedrag en computerprogramma's. Het bevat oplossingen voor twee problemen, opgeworpen in [29], met betrekking tot orthogonale bisimulatie, een semantische equivalentie die te maken heeft met abstractie in de procesalgebra. Verder bestuderen we de se man tiek van de draad-algebra, een procesalgebra voor de semantiek van recente objectgeorienteerde en meerdradige programmeertalen zoals C# en Java. Binnen de programma-algebra zelf onderzoeken we de expressiviteit van uitbreidingen met voorwaardelijke keuze en de zolang-lus. Op het stuk van toepassingen laten we zien dat ver scheidene noninterferentiebegrippen op het gebied van de taalgebaseerde beveiliging gedefinieerd kunnen worden met behulp van de draad-algebra; en we gebruiken programma-algebra om de correctheid te bewijzen van algoritmen en transformatieregels voor goto-verwijdering.

Hoofdstuk 2 bevat een karakterisering van orthogonale bisimulatie. Deze karakterisering berust op een notie van compressie-inhoud van de sporen van een proces: de sporen van het proces na verwijdering van alle interne acties, met uitzondering van een eventuele interne actie aan het eind van het spoor. De compressiestructuur staat tot orthogonale bisimilariteit als de vertakkingsstructuur van [51] tot vertakkende bisimilariteit. In het bijzonder hebben twee processen dezelfde compressiestructuur dan en slechts dan als ze orthogonaal bisimilair zijn.

In hoofdstuk 3 onderzoeken we de complexiteit van het beslissingsprobleem voor orthogonale bisimulatie in een eindig toestandsovergangssysteem. We formuleren een beslissingsalgoritme. Dit algoritme is gebaseerd op het bekende algoritme voor vertakkende bisimulatie [52] beschreven door Groote en Vaandrager [54]. Dus als n het aantal toestanden is, en m het aantal overgangen, dan is $O(n(m+n))$ tijd nodig om orthogonale bisimilariteit te beslissen op een eindig gelabeld overgangssysteem, en O(m + n) ruimte.

Hoofdstuk 4 geeft een structurele operationele semantiek (SOS) [81] voor de draad-algebra. Er worden transitieregels opgesteld voor de strategische afwisselingsoperatoren van [23]. We bewijzen dat de bisimulatie-equivalentie gedefinieerd door de nieuwe SOS de gelijkheid karakteriseert die wordt geïnduceerd door de axioma's van de draad-algebra.

Hoofdstuk 5 beschrijft een denotationele semantiek [11] voor de draad-algebra. Bij

*Summary in Dutch

de constructie van volledige metrische ruimten uit de domeinen van de draad-algebra is de metrische methodologie van De Bakker en Zucker [11] gevolgd. We bewijzen dat de volledige metrische ruimte gevormd door de projectieve rijen een geschikt domein is voor de draad-algebra. Preciezer gezegd, op het niveau van individuele draden bewijzen we dat dit domein oneindige draden uniek representeert. Het is bovendien compatibel met het draadalgebra-domein gebaseerd op volledige ordes (cpo's) [16]. Daarnaast geeft het, vergeleken bij het Cauchyrijendomein, een natuurlijke behandeling van abstractie. We bewijzen dat een specificatie van een reguliere draad een unieke oplossing heeft in dit domein. Op het niveau van multidraden is bewezen dat dit domein op een natuurlijke manier kan worden uitgebreid met de strategische afwisselingsoperatoren van [23], zulks in tegenstelling tot het domein ge baseerd op cpo's. Met betrekking tot abstractie beschrijven we een speciale strategie voor de draad-algebra.

Hoofdstuk 6 verkent de expressiviteit van de uitbreidingen PGAuc en PGAucw van de programma-algebra (PGA) met de eenheidsoperator, voorwaardelijke keuze, en zolang-lussen, onder de luie projectiesemantiek voorgesteld in [22]. Allereerst de finiëren we een projectie van PGAuc (PGA uitgebreid met de eenheidsoperator en voorwaardelijke keuze) naar PGAu (PGA met alleen de eenheidsoperator) [82]. De projectie van PGAuc naar PGA is de samenstelling van deze projectie met de projectie van PGAu naar PGA gedefinieerd in [82]. Vervolgens laten we zien dat PGA met zolang-lussen in bepaalde gevallen niet-regulier gedrag beschrijft. Onder de restrictie dat opeenvolgende voorkomens van zolang-lussen verboden zijn, geven we een projectie van PGAucw (PGAuc uitgebreid met de zolang-constructie) naar PGLBu, een variant van PGAu met achterwaartse sprongen [82]. De projectie van PGAucw naar PGA is bijgevolg de compositie van deze projectie en de projectie van PGLBu naar PGA die beschreven staat in [82]. Het bestaan van onze projecties laat zien in welke mate voorwaardelijke keuzen en zolang-lussen, afgezien van de flexibele programmeerstijl die ze faciliteren, in termen van expressiviteit niet nodig zijn als primitieve instructies. Tot besluit merken we op dat deze projecties gebruikt kunnen worden bij het onderzoek van PGA zelf.

Hoofdstuk 7 laat zien dat de draad-algebra, de semantiek van de programma-algebra, een procesalgebraïsch kader schept voor formalisering en analyse van taalgebaseerde veiligheidskenmerken [88]. We geven interpretaties van verscheidene standaard-noninterferentiebegrippen; noninterferentie is een belangrijke eigenschap die systemen karakteriseert waarvan de executie geen geheime informatie onthult. In het bijzonder hebben we terminatie-ongevoelige noninterferentie (TINI, voor Termination-Insensitive NonInterference) gedefinieerd in de draad-algebra, terminatiegevoelige noninterferentie (TSNI, voor Termination-Sensitive NonInterference), en tijdgevoelige noninterferentie (TISNI, voor TIming-Sensitive NonInterference). We bewijzen correctheid voor deze noninterferentie-eigenschappen, dat wil zeggen dat als een draad n van deze eigenschappen heeft, hij de noninterferentie-eigenschap heeft die Goguen en Meseguer formuleren in [53]. Verder bewijzen we dat on ze benadering alle veilige programma's accepteert die geaccepteerd worden door de gangbare benaderingen van taalgebonden veiligheid die gebaseerd zijn op typensystemen [99, 91]. Met betrekking tot multidraden is bewezen dat TISNI behouden blijft on der de cyclische

afwisselingsoperator, een basale afwisselingsstrategie uit [23]. Met het oog op het behoud van TINI en TSNI introduceren we een speciale afwisselingsstrategie voor de draad-algebra die de rotatie van de draadvector alleen inroept in het geval dat de lopende actie niet persistent is. Behoud kan worden afgedwongen door aan te nemen dat hoge acties persistent zijn. Om te controleren op noninterferentie-eigenschappen kan men bestaande beslissings-tools voor procesalgebraïsche equivalentie gebruiken, zoals [48, 49, 31], om veiligheidscontroleprogramma's te ontwikkelen. Hiermee laten we zien dat de draad-algebra een geschikte procesalgebraïsche theorie is voor de studie van veiligheidseigenschappen in ongestructureer de en meerdradige programmeertalen. Eerder werk aan veiligheid voor sequentiële en meerdradige programmeertalen kan in dit kader dus opnieuw worden bezien.

Hoofdstuk 8 laat zien dat de programma-algebra een wiskundig en systematisch kader verschaft voor redeneringen over correctheid en equivalentie van algoritmen en trans formatieregels voor goto-verwijdering [45]. Wij beschouwen drie bestaande klassen van goto-verwijdering: verwijdering van alle goto's met gebruik van extra variabelen, verwijdering van alle goto's zonder extra variabelen, en verwijdering van bepaalde typen goto's voor kennisextractie. In de eerste klasse bewijzen we de correctheid van het algoritme van Cooper [41] onder gedragsequivalentie met betrekking tot extra boolese variabelen. Deze equivalentie is fijner dan de invoer-uitvoer-equivalentie die gebruikt wordt in [41] en andere technieken [33, 41, 76, 3, 39, 72] voor goto-verwijdering met gebruik van extra variabelen. We merken op dat een soortgelijk resultaat kan worden bereikt voor de benadering van Böhm en Jacopini [33]. In de tweede klasse moet voor de welbekende benaderingen van Peterson et al. [80] en Ramshaw [83] worden voorondersteld dat de programma's reducibel zijn of vrij van kop-kop-kruisingen. Voor de verwijdering van alle gotos zonder invoering van extra variabelen stellen we daarom een techniek voor om kop-kop-kruisingen in programma's kwijt te raken, en vervolgens gebruiken we de resultaten van Peterson et al. en Ramshaw. We laten zien dat ons algoritme correct is onder gedragsequivalentie, een relatie analoog aan de padequivalentie die gebruikt wordt in [80, 83]. En tenslotte, in de derde klasse, het verwijderen van bepaalde typen goto's met het doel kennis te extraheren die ligt ingebed in geërfde programmatuur, geven we correctheidsbewijzen voor enkele transformatieregels voor het herstructureren van Cobol-programma's in een realistische toepassing [96]. We suggereren ook een automatiseerbare methode om correctheid te bewijzen voor alle transformatieregels in [96]. Om kort te gaan, we hebben laten zien dat goto's geëlimineerd kunnen worden met de aanpak van de programma-algebra; en we hebben de goto-verwijdering met wiskundige strengheid uitgelegd.

Bibliography

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 197–222. Elsevier, 2001.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[3] E. Ashcroft and Z. Manna. The translation of "goto" programs to "while" programs. *Inform. Proc.*, 71:147–152, 1972.

[4] J. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings 4th Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 477–492, 1993.

[5] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78(3):205–245, 1988.

[6] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fund. Inform.*, 9:127–168, 1986.

[7] J.C.M. Baeten and R.J. van Glabbeek. Another look at abstraction in process algebra. In Th. Ottmann, editor, *ICALP 87*, volume 267 of *Lecture Notes in Computer Science*, pages 84–94. Springer-Verlag, 1987.

[8] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[9] C. Baier and M. Majster-Cederbaum. The connection between initial and unique solutions of domain equations in the partial order and metric approach. *Formal Aspects of Computing*, 9:425–445, 1997.

[10] B.S. Baker and S.R. Kosaraju. A comparison of multilevel break and next statements. *Journal of the ACM*, 26:555–566, 1979.

[11] J.W. Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[12] T. Basten. Branching bisimulation is an equivalence indeed. *Information Processing Letters*, 58(3):333–337, 1996.

[13] D.E. Bell and L.J. La Padula. Secure computer systems: mathematical foundations and model. Tech. Rep. M74-244, MITRE Corporation, Bedford, Massachussets, 1973.

[14] J.A. Bergstra. `http://www.science.uva.nl/~janb/pga/whypga.html`.

[15] J.A. Bergstra and I. Bethke. Molecular dynamics. *J. Logic Algebr. Programming*, 51:125–156, 2002.

[16] J.A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2003.

[17] J.A. Bergstra and I. Bethke. Polarized process algebra with reactive composition. *Theoretical Computer Science*, 343(3):285–304, 2005.

[18] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, The ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[19] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Center, Amsterdam, 1982.

[20] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Inform. and Control*, 60 (1-3):109–137, 1984.

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

[22] J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *J. of Logic and Algebraic Programming*, 51:125–156, 2002.

[23] J.A. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. Computing Science Report PRG0404, Programming Research Group, University of Amsterdam, 2004.

[24] J.A. Bergstra and C.A. Middelburg. Thread algebra with multi-level strategic interleaving. Computing Science Report 200441, Department of Mathematics and Computing Science, Eindhoven University of Technology, 2005.

[25] J.A. Bergstra and C.A. Middelburg. Thread algebra with multi-level strategic interleaving. In S.B. Cooper, B.Loewe, and L. Torenvliet, editors, *CiE*, volume 3526 of *LNCS*, pages 35–48, 2005.

[26] J.A. Bergstra and C.A. Middelburg. Maurer computers with single-thread control. *Fundamenta Informaticae*, 2007. To appear.

[27] J.A. Bergstra and A. Ponse. Combining programs and state machines. *J. of Logic and Algebraic Programming*, 51:175–192, 2002.

[28] J.A. Bergstra and A. Ponse. A bypass of Cohen's impossibility result. In P.M.A. Sloot, A.G. Hoekstra, T.Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing-EGC 2005*, volume 3407 of *Lecture Notes in Computer Science*, pages 1097–1106. Springer-Verlag, 2005.

[29] J.A. Bergstra, A. Ponse, and M.B. van der Zwaag. Branching time and orthogonal bisimulation equivalence. *Theoretical Computer Science*, 309:313–355, 2003.

[30] I. Bethke and A. Ponse. *Programma-algebra, een inleiding tot de programmatuur (in Dutch).* Amsterdam University Press, Vossiuspers, 2003.

[31] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Common, and A. Finkel, editors, *Proc. 13th Conference on Computer Aided Verification-CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[32] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall: Englewood Cliffs NJ, 1981.

[33] C. Böhm and G.Jacopini. Flow diagram, Turing machines and languages with only two formation rules. *Comm. ACM*, 9:366–371, 1966.

[34] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *Journal of the ACM*, 43(5):863–914, 1996.

[35] A. Bossi, C. Piazza, and S. Rossi. Compositional information flow security for concurrent programs. Technical Report CS-2005-4, Dipartimento di Informatica, Università Cá Foscari di Venezia, Italy, 2005.

[36] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281:109–130, 2002.

[37] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Oliver, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[38] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structure in propositional temporal logic. *Theoretical Computer Science*, 59(1,2):115–131, 1988.

[39] J. Bruno and K. Steiglitz. The expression of algorithms by charts. *J. ACM*, 19:517–525, 1972.

[40] F.W. Calliss. Problems with automatic restructures. *ACM SIGPLAN Notices*, 23(3):13–21, 1988.

[41] D.C. Cooper. Böhm and Jacopini's reduction of flow charts. *Comm. ACM*, 10:463–473, 1967.

[42] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for re-engineering. *Software Maintenance and Evolution*, 14(4):257–292, 2002.

[43] M. Dam. Decidability and proof systems for language-based noninterference relations. In *POPL'06*, pages 67–78, 2006.

[44] D.E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[45] E.W. Dijkstra. Goto statement considered harmful. *Comm. ACM*, 11:147–148, 1968.

[46] R. Engelking. *General Topology*. Polish Scientific Publishers, 1977.

[47] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *J. of Computer Security*, 3(1):5–33, 1995.

[48] R. Focardi and R. Gorrieri. Automatic compositional verification of some security properties for process algebras. In T. Margaria and B. Steffen, editors, *Proc. of TACA'96*, volume 1055 of *LNCS*, pages 111–130, 1996.

[49] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.

[50] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *FOSSACS'05*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[51] R.J. van Glabbeek. What is branching time semantics and why to use it? *Bulletin of the EATCS*, 53:190–198, 1994.

[52] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[53] J. Goguen and J. Meseguer. Secure policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[54] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, editor, *ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer-Verlag, 1990.

[55] D. Harel. On Folk theorem. *Comm. ACM*, 23:379–389, 1980.

[56] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF-Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[57] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[58] C. Jones. *Estimating Software Costs*. McGraw-Hill: New York, 1998.

[59] W.A. Kirk and B. Sims. *Handbook of Metric Fixed Point Theory*. Kluwer Academic, London, 2001.

[60] P. Klint. Meta-environment for generating program environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[61] D.E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974.

[62] D.E. Knuth and R.W. Floyd. Notes on avoiding goto statements. *Inform. Processing Letters*, 1:23–31, 1971.

[63] P.C. Kocher. Timing attacks on implementations on Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[64] S.R. Kosaraju. Analysis of structured programs. *J. of Computer and Systems Sciences*, 9:232–255, 1974.

[65] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001.

[66] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software: Practice and Experience*, pages 1395–1438, 2001.

[67] H.F. Ledgard and M. Marcotty. A genealogy of control structures. *Comm. ACM*, 18:629–639, 1975.

[68] S. Lerner. *Automatically Proving the Correctness of Program Analyses and Transformations*. PhD thesis, Univ. of Washington, 2006.

[69] G. Lowe. Semantic models for information flow. *Theoretical Computer Science*, 315:209–256, 2004.

[70] J. McKee. Maintenance as a function of design. In *Proceedings of the 1984 National Computer Conference (AFIPS Conference Proceedings)*, volume 53, pages 187–193. AFIPS Press: Reston VA, 1984.

[71] A.C. Meyers. Jflow: Practical mostly-static information flow control. In *ACM Symp. on Principles of Programming Languages*, pages 228–241, 1999.

[72] H.D. Mills. The new math of computer programming. *Comm. ACM*, 18:43–48, 1975.

[73] R. Milner. *A calculus of communicating system*. LNCS 92. Springer Verlag, Berlin, 1980.

[74] R. Milner. A modal characterisation of observable machine behaviour. In G. Astesiano and C. Böhm, editors, *CAAP 81*, volume 112 of *Lecture Notes in Computer Science*, pages 25–34. Springer-Verlag, 1981.

[75] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[76] G. Mirkowska. *Algorithmic logic and its applications*. PhD thesis, Univ. of Warsaw, 1972.

[77] S. Owre, J.M. Rushby, and N. Shankar. Pvs: a prototype verification system. In Deepak Kapur, editor, *CADE'92*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

[78] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16 (6):973–989, 1987.

[79] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, volume 104 of *LNCS*, pages 167–183, 1982.

[80] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Comm. ACM*, 16:503–512, 1973.

[81] G. Plotkin. A structural approach to operational semantics. Aarhus DAIMI FN-19, Computing Science Department, 1981.

[82] A. Ponse. Program algebra with unit instruction operators. *J. Logic Algebr. Programming*, 51:157–174, 2002.

[83] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35:893–920, 1988.

[84] P. Ryan and S. Schneider. Process algebra and non-interference. *J. of Computer Security*, 9, 2001.

[85] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. A. Ershov 5th International Conference on Perspectives of System Informatic*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–274, 2003.

[86] A. Sabelfeld and D.Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. Computer Security Foundations Workshop*, Lecture Notes in Computer Science, pages 200–214, 2000.

[87] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 376–394. Springer-Verlag, 2002.

[88] A. Sabelfeld and A. Myers. Language-based information flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, 2003.

[89] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[90] A. Sellink, H. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming*, 45:193–243, 2002.

[91] G. Smith and D. Volpano. Secure information flow in multi-threaded imperative languages. In *POPL'98*, volume 29, pages 355–364, 1998.

[92] H.M. Sneed. Extracting business logic from existing COBOL programs as a basis for redevelopment. In *Ninth International Workshop on Program Comprehension (IWPC'01)*, pages 167–176, 2001.

[93] H.M. Sneed and S.H. Sneed. Creating web services from legacy host programs. In *5th International Workshop on Web Site Evolution*, pages 59–65, 2003.

[94] V. Stoltenberg-Hansen, I. Lindstrom, and E.R. Griffor. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science 22. Cambridge University Press, 1994.

[95] F.W. Vaandrager. Two simple protocols. In J.C.M. Baeten, editor, *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*, pages 23–45. Cambridge University Press, Cambridge, 1990.

[96] N. Veerman. Revitalizing modifiability of legacy assets. *Journal of software maintenance and evolution*, 16:219–254, 2004.

[97] N. Veerman. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*, 57:129–163, 2005.

[98] N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice and Experience*. To appear.

[99] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. of Computer Security*, 4(3):167–187, 1996.

[100] T.D. Vu. Deciding orthogonal bisimulation. *Formal Aspects of Computing*. To appear. Also as Chapter 3 of this thesis.

[101] T.D. Vu. Noninterference in thread algebra. Submitted for publication. Available from `http://www.science.uva.nl/~tdvu`. Also as Chapter 7 of this thesis.

[102] T.D. Vu. Structural operational semantics for thread algebra. Available from `http://www.science.uva.nl/~tdvu`. Also as Chapter 4 of this thesis.

[103] T.D. Vu. The compression structure of a process. *Information Processing Letters*, 96:225–229, 2005. Also as Chapter 2 of this thesis.

[104] T.D. Vu. Metric denotational semantics for BPPA. Report PRG0503, Programming Research Group, University of Amsterdam, 2005. Also as Chapter 5 of this thesis.

[105] V.L. Winter. *Proving the Correctness of Program Transformations*. PhD thesis, Univ. of New Mexico, 1994.

Acknowledgements

This thesis cannot be completed without the support of many people. First of all, I would like to thank my promoter Jan Bergstra who has given me the opportunity to work in his group, in a wonderful research environment. He has laid the foundation for the work presented in this thesis.

I thank Alban Ponse for his guidance. He is always patient to listen to my question in research as well as in life. He suggested most topics investigated in the thesis, giving help on many technical details, and checking the many revisions of the chapters. I also thank Inge Bethke for the same reason. In fact, Alban and Inge are the greatest supervisors I could ever have.

I thank members of the doctoral committee for reviewing this thesis.

I would like to thank Wan Fokkink for his guidance during my master study. In addition, he gave me many comments to improve an earlier version of Chapter 3.

I would like to thank Piet Rodenburg for translating the summary into Dutch. He also gave me nice suggestions to improve Chapter 6.

I thank Mark van der Zwaag for his comments on Chapter 5.

I thank Niels Veerman for the suggestions on Cobol that has improved Chapter 8 considerably.

Bob Diertens is the finest office mate I can imagine. He has always answered passionately any questions I asked. I thank him for that.

I thank PAM members for the many talks and discussions on Wednesdays.

I thank all my friends in the Netherlands for the parties, barbecues and many other interesting events. Anh Hieu-chi Hoa and Chip were like family to me. Thieu-Nga and Hung-Giang were always very kind and enthusiastic, especially in helping us move to the new place. I also thank all my friends overseas. Their email and chat via Yahoo messenger have been a great support for me during my stay in Europe.

I would like to thank my sisters-in-law Hoa and Chi for their support. Chi was there when our son was born and Hoa traveled from Delft to Amsterdam every week-end to take care of the baby so that I could focus on the thesis.

I thank my parents-in-law for making me feel like a real child in the family.

I would like to thank my parents, my sister and brother-in-law for their love and support through the years.

Last but not least, I thank my husband Thang for his love, encouragement and for always being there when I needed. This thesis cannot be completed without the

agreement of my dearest one, our son Khoi. I thank them for bringing the happiest moments into my life.