Formal Enforcement of Security Policies: An Algebraic Approach

Thèse

GUANGYE SUI

Doctorat en Informatique
Philosophiæ doctor (Ph.D.)

Québec, Canada

Résumé

La sécurité des systèmes d'information est l'une des préoccupations les plus importantes du domaine de la science informatique d'aujourd'hui. Les particuliers et les entreprises sont de plus en plus touchés par des failles de sécurité et des milliards de dollars ont été perdus en raison de cyberattaques.

Cette thèse présente une approche formelle basée sur la réécriture de programmes permettant d'appliquer automatiquement des politiques de sécurité sur des programmes non sécuritaires. Pour un programme P et une politique de sécurité Q, nous générons un autre programme P' qui respecte une politique de sécurité Q et qui se comporte comme P, sauf si la politique est sur le point d'être violée.

L'approche présentée utilise l'algèbre $\mathcal{E}BPA^*_{0,1}$ qui est une variante de $BPA^*_{0,1}$ (*Basic Process Algebra*) étendue avec des variables, des environnements et des conditions pour formaliser et résoudre le problème. Le problème de trouver la version sécuritaire P' à partir de P et de Q se transforme en un problème de résolution d'un système linéaire pour lequel nous savons déjà comment extraire la solution par un algorithme polynomial. Cette thèse présente progressivement notre approche en montrant comment la solution évolue lorsqu'on passe de l'algèbre de $BPA^*_{0,1}$ à $\mathcal{E}BPA^*_{0,1}$.

Abstract

The security of information systems is one of the most important preoccupations of today's computer science field. Individuals and companies are more and more affected by security flaws and billions of dollars have been lost because of cyber-attacks.

This thesis introduces a formal program-rewriting approach that can automatically enforce security policies on non-trusted programs. For a program P and a security policy Q, we generate another program P' that respects the security policy Q and behaves like P except when the enforced security policy is about to be violated.

The presented approach uses the $\mathcal{E}BPA^*_{0,1}$ algebra that is a variant of the BPA (*Basic Process Algebra*) algebra extended with variables, environments and conditions to formalize and resolve the problem. The problem of computing the expected enforced program P' is transformed to a problem of resolving a linear system for which we already know how to extract the solution by a polynomial algorithm. This thesis presents our approach progressively and shows how the solution evolves when we move from the $BPA^*_{0,1}$ algebra to the $\mathcal{E}BPA^*_{0,1}$ algebra.

Contents

List of Tables

List of Figures

Acknowledgements

I want to express my great appreciation to my supervisor Pr Mohamed Mejri. I feel really blessed to have a hard-working and talented professor like him. His advises always are important inspirations. When I feel frustrated, he is always patient and present to encourage me. I believe that the knowledge and the research method he taught me will be a great treasure for me in the future. I also wish thank to Professor Hatem Ben Sta from University of Tunis El Manar, who not only provides advises to my research but also shares wisdom about how to become a good Ph.D student.

Special thanks are due to professors Nadia Tawbi , Béchir Ktari and François Laviolette from Université Laval, for their advises about how to improve my thesis made me benefit a lot. I also want to thank professors Béchir Ktari, Claude-Guy Quimper and Pascal Tesson for their brilliant courses during my Ph.D.

I and thankful all the members in Laboratoire de Sécurité Informatique (LSI) for their warm support and companionship, including Etienne Theodore Sadio, He Lei, Maxime Leblanc, Mina Alishahi, Jaouhar Fattahi, Memel Emmanuel Lathe, Laila Boumlik, Marwa Ziadia, Saeed Abbasi, Ahmed Mohamed El-Marouf, Parvin Ramezani, Khadija Arrachid and Imen Sallem.

I would like to address special thanks to my parents for their unconditional love and support.

Chapter 1

Introduction

1.1 Motivation and Background

Security is one of the most important preoccupations of today's information systems. Billions of dollars have been lost during the last years because of cyber attacks and here are some examples:

- In April 2011, Sony announced in [58] that all PlayStation Network accounts and users' credit card information had been accessed by an "unauthorized person". Some investigations state that this may have affected 70 million Sony customers.
- According to [51], in April 2013, the famous hacktivist group known as *Anonymous* launched a massive cyber-attack against Israel, causing, according to the hackers, multi-billion dollar damage.
- According to [67], beginning from August 31, 2014, a collection of almost 500 private pictures of various celebrities were posted on the Internet. The images were believed to have been obtained via a breach of Apple's cloud services suite iCloud.
- According to [52], health insurer Premera Blue Cross was a victim of a cyber attack in Mars, 2015 that may have exposed medical data and financial information of 11 million customers.

Individuals and companies are more and more affected by security flaws, and financial losses are not only the possible consequences. Human losses are also possible even for non-military staff (for example: vehicle and aviation systems).

Formal methods are mathematically-based languages, techniques and tools that can be used during all the development steps to produce high-quality systems with proved properties. This is essential for critical systems (nuclear power stations, aircraft control systems, etc.), where design or implementation errors may engender disastrous consequences. It is a well-known fact that using formal methods is generally difficult and expensive since it requires highly-qualified persons. For that reason, these techniques can be offered only by rich institutions (government, military, etc.). However, if user-friendly tools that make the enforcement process completely automatic are developed, then we can cut corresponding costs and make formal methods available for small companies or even for individuals.

This thesis aims to automatically and formally enforce security policies in some systems. Fortunately, there are already some fundamental results that allow understanding which security policies can be enforced and by which mechanism.

In [57], Schneider distinguished the security policies and security properties (safety and liveness) and showed that only a part of safety properties can be enforced by EM (Execution Monitoring). If a given security property includes a liveness part, then the execution monitors without the ability to modify the target program, such as an edit automaton [31], is not appropriate. Fortunately, any property can be separated into safety and liveness parts as it has been established in [3, 4], otherwise the task of enforcing security will be harder.

More recent work of Clarkson and Schneider introduces hyperproperties which extend the definition of security properties, and based on the discussion in [15], they believe that they are powerful enough to specify all discovered security policies.

Execution Monitoring belongs to the class of dynamic analysis techniques that includes many other interesting works like [28, 35, 37, 63, 66]. Static analysis [14, 16, 27, 36, 50] approaches, on the other hand, can be used to significantly decrease the overhead involved by dynamic approaches. For instance, statical analysis can prove that a program is safe with respect to some properties and avoid its monitoring.

Recently, program rewriting [18, 29, 43, 59] techniques show their promising future since they gather advantages of previous techniques. The basic idea is to rewrite an untrusted program statically so that the new generated version respects some security requirements.

A formal approach was introduced in [24, 48] to automatically enforce security policy on non-trusted programs. But, it was for a limited language called $BPA^*_{0,1}$ (an extended version of BPA: Basic Process Algebra). In this thesis, we want to extend the expressiveness of the language by handling variables, conditions and environments. This new algebra is called basic process algebra with environment and denoted by $\mathcal{E}BPA^*_{0,1}$.

1.2 Problem

To clarify the idea, let us consider the academic example shown by Figure 1.1.

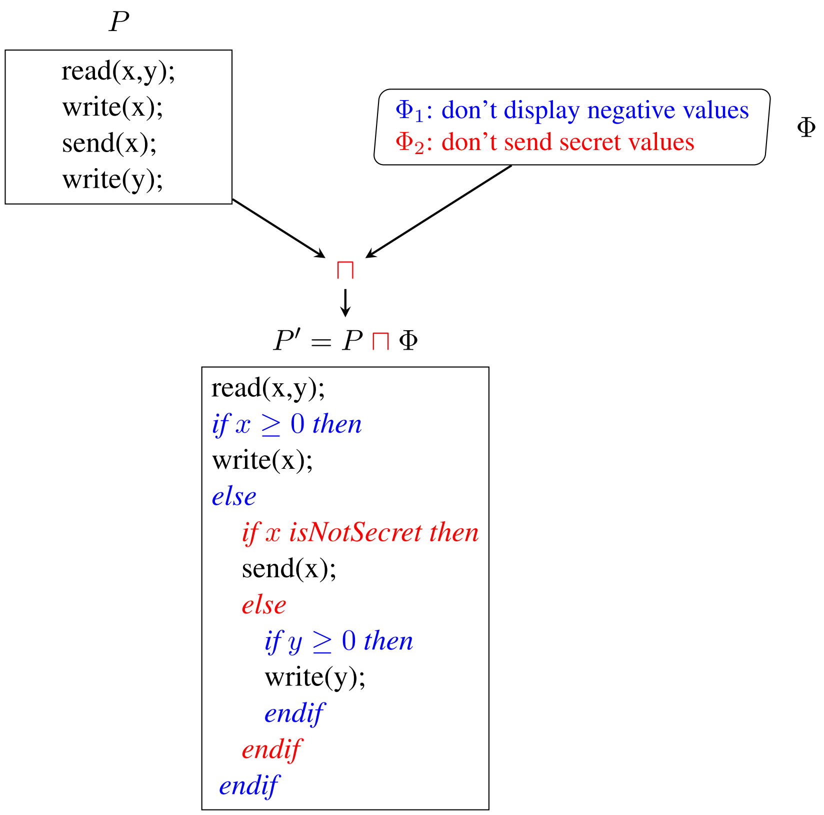


Figure 1.1: Idea.

Given a security policy Φ and a process P, we want to define an operator $\sqcap$ that generates a process P' containing the enforcement of Φ on P. P' respects Φ, preserves all the good behaviors of P and it does not introduce any new ones. Generally P' is computed by inserting some tests in P at some critical points so that the security policy will always be respected. The red text in P' of Figure 1.1 is the test inserted to enforce Φ_2 and the green one is the test added to enforce the policy Φ_1.

A more realistic example related to buffer overflow is given hereafter. Suppose that our program P is the following:

```
void main(int argc, char *argv[]){
buffer[10];
strcpy(tampon, argv[1]);
}
```

Suppose that we have a security policy stating that no buffer overflow should occur. A buffer overflow takes place when we write in a given memory zone more than what it can support. In our example, if *argv[1]* receives more than ten characters, the exceeded characters will be written outside *buffer* and can cause serious damages. Normally, if we use a dynamic approach, we should set up another monitoring program to enforce this security policy. But, thanks to program rewriting, we can add some checks at critical points of the program statically. One possible solution is as follows:

```
void main(int argc, char *argv[]){
buffer[10];
if(sizeof(argv[1])<=10)
strcpy(tampon, argv[1]);
}
```

1.3 Methodology

The methodology that we adopt is the following:

- We introduce a new algebra $\mathcal{E}BPA^*_{0,1}$, which takes variables, conditions and environment into consideration. $\mathcal{E}BPA^*_{0,1}$ is an extension of $BPA^*_{0,1}$ [9, 10, 48]. Authors in [43] had already introduced a dynamic approach based on an ACP (algebra for communicating process in [8]) like algebra.

  We choose BPA algebra because the technique that we propose is very efficient for a sequential language but not for a concurrent one. In fact, our approach needs to explore the execution paths of the program and this may explode for a concurrent program. However, the technique tries to minimize the control actions that are involved during the execution of the program. Indeed, instead of having a monitor that control every action of the program, we use a rewriting approach to statically insert tests in some critical points of the program so that the security policy will be enforced. This approach can be applied to some script languages (PHP, PERL, etc.) and can be combined with other approaches such that [43] to better enforce security for the concurrent programs.

- We suppose that the program P is given as a process or it can be transformed to a process in $\mathcal{E}BPA^*_{0,1}$. This process algebra is expressive enough to handle an interesting simplified version of C-like programs (we will do not take into consideration pointer for example).

- We suppose that the security policy Φ is given as a formula in the LTL (Linear Temporal Logic) [54]. We know how to translate it into a term in $\mathcal{E}BPA^*_{0,1}$.

- We define an operator $\sqcap$ that enforces Φ on P. Basically, the operator "$\sqcap$" transforms the problem of enforcing Φ on P to a problem of generating a set of equations in $\mathcal{E}BPA^*_{0,1}$ and resolving them as shown by Figure 1.2. We already know how to get the solution of the equations generated by $\sqcap$.

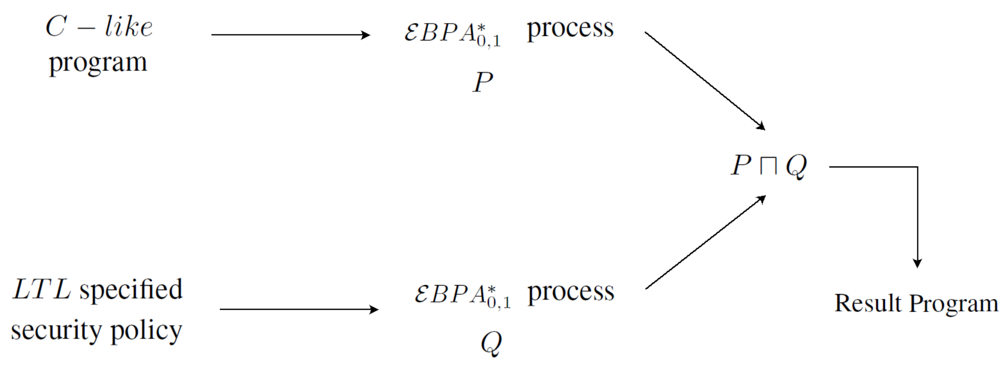

$C - like$ program
$\mathcal{E}BPA^*_{0,1}$ process P
$P \sqcap Q$
LTL specified security policy
$\mathcal{E}BPA^*_{0,1}$ process Q
Result Program

Figure 1.2: Security policy enforcement process with $\mathcal{E}BPA^*_{0,1}$.

- We prove that the secure version of the program generated by "$\sqcap$" is sound and complete with respect to the inputs(the untrusted program and the security policy). The soundness property states that all traces of the newly generated process respect the security policy and belong to the set of traces of the original insecure one. The completeness property, or transparency, on the other hand, states that any trace of the original program that respects the security policy should be kept in the secure version.

1.4 Advantages of our approach

Our approach significantly benefits from the following three major advantages:

1.4.1 Formal Approach

Formal methods are mathematically-based languages, techniques and tools that can be used during all the development steps to produce high-quality systems with proved properties. Unlike test-based approaches, which can only ensure that systems satisfy the requirements for test cases, formal methods use mathematical proofs to ensure properties of a system. This is essential for critical systems (nuclear power stations, aircraft control systems, etc.), where design or implementation errors may engender disastrous consequences.

For our approach, when an end user proposes a program and a security policy, we transfer them into two processes P and Q in $\mathcal{E}BPA^*_{0,1}$, then we generate a greatest common factor of P and Q, denoted by $P \sqcap Q$ that contains only the traces of P that respect the security policy. Process $P \sqcap Q$ behaves like P except that it stops any execution path whenever a security policy is about to be violated. Finally we translate the process $P \sqcap Q$ to the original language (e.g. C-Like language). The presented approach uses the $\mathcal{E}BPA^*_{0,1}$ algebra which is a variant of *BPA* (*Basic Process Algebra*) extended with variables, environments and conditions to formalize and resolve the problem.

1.4.2 Automatic Approach

The cost of formal approaches can be very expensive. However, if user-friendly tools that make the enforcement process completely automatic is developed, then we can cut corresponding costs significantly and remove human interventions that are error prone. For our approach, once an end user has specified his security policies, we will automatically enforce them with a program rewriting technique.

1.4.3 Aspect Oriented Approach

Our approach belongs to aspect-oriented paradigm (AOP) [17, 38, 61, 64] which aims to increase productivity and maintenance by dealing with different aspects separately.

AOP makes systems easier to maintain

Maintenance consumes 80%-90% of the total cost of most systems. With classical development approaches, such as procedural programming, functional programming, logic programming and object-oriented programming, each aspect of system's specification is divided and scattered throughout different slices (module, function, object, etc) that include different aspects in the same time (as shown in Figure 1.3), this makes systems difficult to generate and maintain. For example, if one experienced software engineer leave and a new employee want to change one aspect of the system, it will be very difficult and time-consuming to go through all the modules to find the right codes that deal with a particular aspect, even with well established documents.

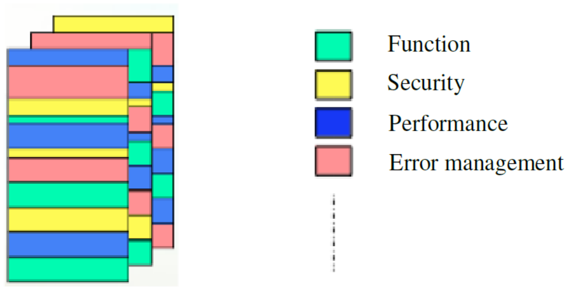


Figure 1.3: Problem of software development approaches.

AOP increase modularity

To increase modularity, software system should imitate other mature industrialize products (houses, cars, airplanes, etc.), which means separating the system into different aspects and allowing different high qualified experts to contribute in their special fields. An example of this idea is shown in Figure 1.4.

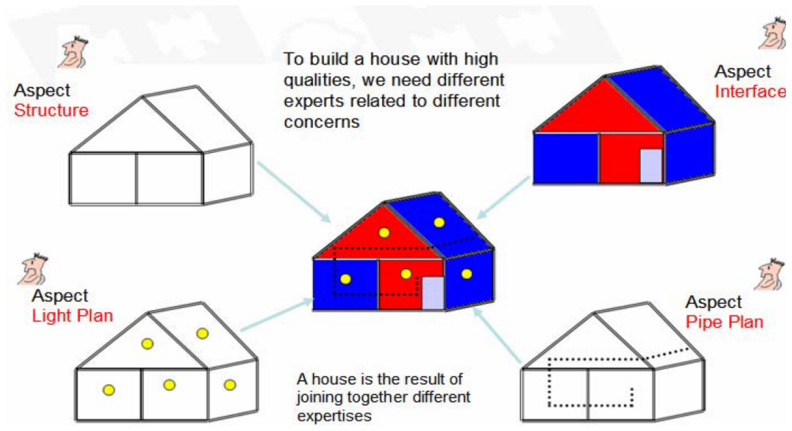

Aspect
Structure
To build a house with high qualities, we need different experts related to different concerns
Aspect
Interface
Aspect
Light Plan
Aspect
Pipe Plan
A house is the result of joining together different expertises

Figure 1.4: Different high qualified experts for different aspects.

More specifically, when an end user specifies requirements for a system, aspect-oriented programming approach will separate them into different aspects (shown in Figure 1.5). Then, each aspect will be developed by appropriate experts. Finally, the approach will integrate different module related to different concerns into the final system automatically. The whole process is presented in Figure 1.6 (page 8).

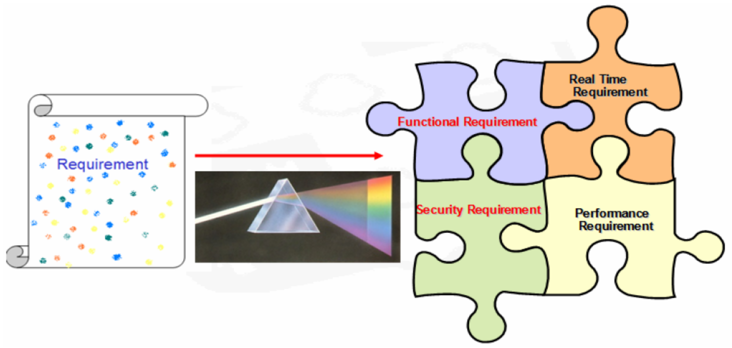


Figure 1.5: Separation of requirements.

AOP languages

Actually, aspect-oriented programming (AOP) is not a new concept, CSS (Cascading Style Sheets) is an aspect that separate web-page layout from content. But today, it is more established as a paradigm and there are programming languages supporting it including $AspectJ$, $AspectC++$, $Aspect\#$ and $HyperJ$.

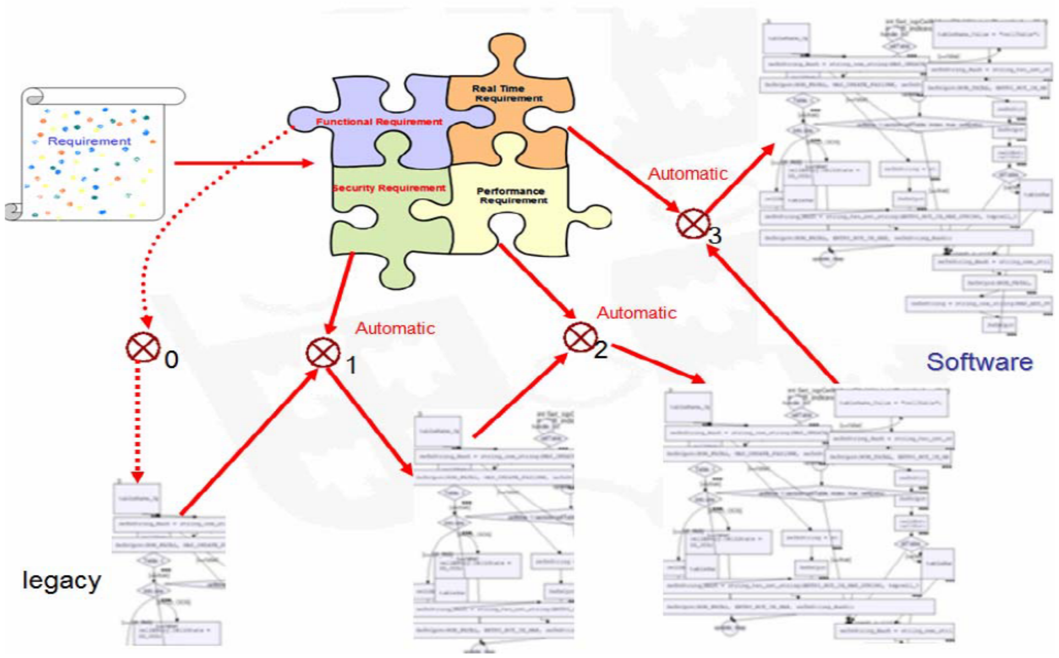


Figure 1.6: Integration of different aspects.

1.5 Organization of the thesis

This thesis is structured as follows:

- In chapter 2, we present three major enforcement approaches and we discuss their enforcement abilities.
- Chapter 3 gives an overview of important related works.
- Chapter 4 introduces $CBPA^*_{0,1}$, a process algebra that can be used to specify system together with their security policies. It also introduces our rewriting technique to enforce security policies on programs.
- Chapter 5 extends the approach to $\mathcal{E}BPA^*_{0,1}$, which is $BPA^*_{0,1}$ enriched with conditions, variables and environments.
- Chapter 6 presents a prototype that we implemented to illustrate our approach.
- Chapter 7 concludes this thesis and discusses some perspectives.

Chapter 2

Enforcement Approaches

In this chapter we introduce enforcement mechanisms. State of the art contains three major enforcement mechanisms: static approach, dynamic approach and program rewriting. Based on the works in [34], we discuss in this chapter different security enforcement approaches (static analysis, dynamic analysis and program rewriting) and we show that program rewriting can enforce more security policies than the other two mechanisms.

This chapter is structured as follows:

- In Section 2.1, we introduce static approach and discuss its enforcement ability.
- In Section 2.2, we present dynamic approach and show some examples of it. We also discuss its enforcement ability and we compare it to the static approach.
- In Section 2.3, we introduce the essence of program rewriting approach and we illustrate it by an example.
- In Section 2.4, we introduce an approach developed by Mejri and Fujita in [48], which can solve the security policy enforcement problem by program rewriting. We will also introduce an interesting extension of it from [30].
- Finally, in Section 2.5, we discuss the enforcement ability of program rewriting approach and compare it with other approaches.

2.1 Static Approaches

Static approaches aim to check codes before their executions. Generally, they require fewer system resources, since they are applied before the execution of the program. However, they have their disadvantages. Since static analysis does not run the program, some security policies that need some dynamic information can not be enforced by this approach. Examples of static analysis method include model-checking [14], abstract interpretations [16], proof-carrying code (PCC) [50], type system [23, 27] and symbolic execution [36, 39]. In the remaining part of this section, we show some examples of these approaches and discusses their enforcement ability.

2.1.1 Model Checking

In computer science, Model Checking generally aims to check whether a given model of a system satisfies its specification. Usually, it involves three major steps:

1. Build a model $\mathcal{M}$ (e.g. Kripke structure [56]) for the system we want to check.
2. Specify a security policy φ using some formal language (e.g. Temporal logic).
3. Check whether the security policy is true in the model, i.e: $\mathcal{M} \models \varphi$.

The process of model checking is resumed by Figure 2.2.

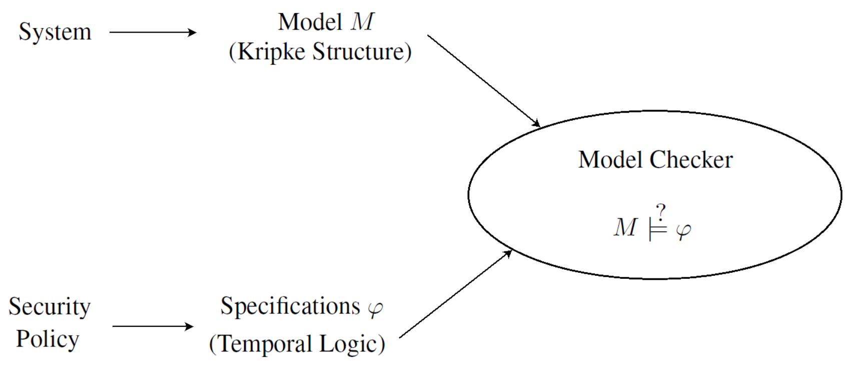


Figure 2.1: Model Checking Verification.

The Kripke structure for specifying the model is a variation of nondeterministic automaton. It has a set of global states, a set of states transition relations and a labeling function to specify which proposition is true in each state. The temporal logic includes Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) and it is used to specify properties with time related operators. For example: "property P will hold eventually" or "property P always hold, until property Q is true". For the systems modeled by a Kripke structure, as the number of states increase, the complexity of the analysis increase exponentially, this causes state space explosion [12, 47] problem and it is one of the major challenges for model checking.

2.1.2 Proof-Carrying Code (PCC)

When people distribute codes remotely, a mechanism is required to build trust between code producers and code consumers. For example, when agent A wants to send a machine language code L to agent B, he wants to convince agent B that L respects type safety property. In [50], the author provides a mechanism (Proof-Carrying Code) to address this problem. Cryptography is an alternative solution, but it highly depends on the credibility of the code producer, but even highly professional code producers may make mistakes.

Unlike Cryptography, Proof-Carrying code (PCC) allows code consumers to announce their security policies to the code producers. Code producers will generate a formal security proof to prove that untrusted code satisfies the security policies and send the code and proof to the code consumer. At last, consumers will use a proof validator to check the untrusted code with the provided proof. More specifically, according to [50], the whole process of PCC contain 3 steps as shown by Figure 2.2.

1. Certification: The code producer compiles the source code and generates a proof ensuring that this code satisfies the code consumer's security policies. In general, the PCC binary consists of a native code component and a proof of successful verification.
2. Validation: The code consumer validates the proof part of the PCC binary and loads the native code component for execution.
3. Execution: Finally, the code consumers can decide to execute the code when it needed, without validating the proof again. This is because the previous stage already makes sure that the code satisfies the security policies.

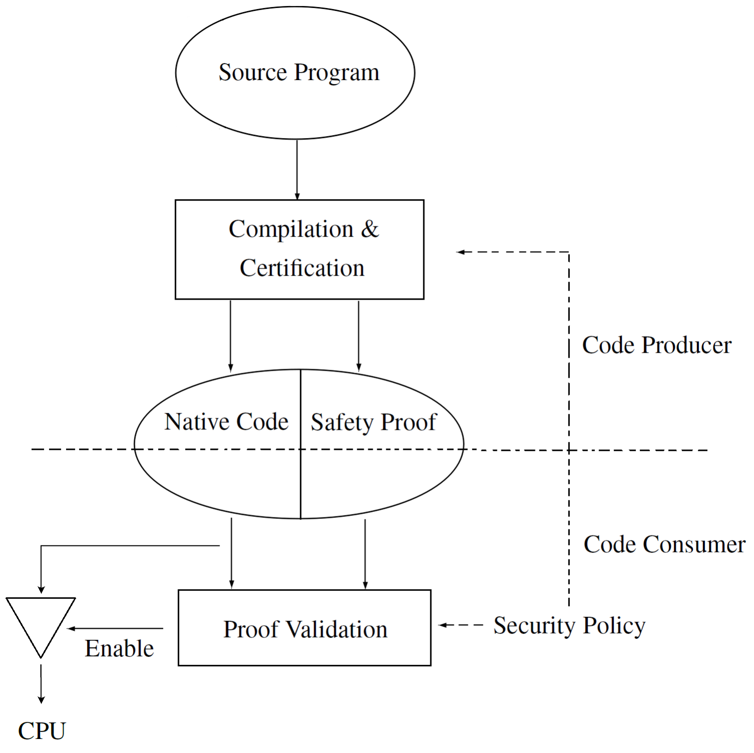


Figure 2.2: Proof-Carrying Code Steps from [50].

2.1.3 Enforcement Abilities of Static Approach

In [34], Hamlen, Morrisett and Schneider provided a formal model to discuss the computability classes of different enforcement mechanisms.

In [34], authors use a Turing Machine (TM) introduced in [42]. TM has finite states and only one tape, it seems to be an obvious candidate for modeling the untrusted program. However, in [34], authors state that there are mainly two reasons for using a more powerful mechanism (Program Machine or PM) instead of TM. Firstly, the PMs' infinite input tapes provide a method to model infinite length input strings, while TM can not. Secondly, a TM encodes all its runtime information into one tape, while a PM separates them into three tapes and distinguishes the information that is available to the enforcement mechanism from the information that is not. As a PM runs, it exhibits the information to enforcement mechanisms by writing an encoding form of this information into the trace tape.

In this model, untrusted programs are modeled by a PM, which is a deterministic Turing machine that controls three infinite tapes:

- An input tape: it contains all the information unavailable to the enforcement mechanism initially. For example: user inputs, non-deterministic choice outcomes, etc.
- A work tape: it is blank initially and can be accessed by the PM without restriction.
- A write-only trace tape: it records security relevant behavior that can be observed by the enforcement mechanism.

Security policies divide the sets of all untrusted programs (modeled by PMs) into those satisfying it and the others. In the remaining part of this section, the ability of different security policies enforcing approaches (static analysis, dynamic analysis and program rewriting) will be discussed respectively using this formal model. We will also introduce enforceable policy classes for different mechanisms.

In [34], it has been stated that a security policy $\mathcal{P}$ can be considered as statically enforceable in the model above, if there exists a Turing machine $M_{\mathcal{P}}$ that takes an encoding of the system (a Turing machine M belonging to PM) as input, and if the system satisfies the security policy $\mathcal{P}$, then $M_{\mathcal{P}}(M)$ accepts it in a finite time; otherwise $M_{\mathcal{P}}(M)$ rejects it in a finite time. In other words, if a security policy P is statically enforceable, then we can find a Turing machine, which can check whether a given system (specified by M) respect P or not within a finite time.

For example, the security policy stating that "The program should terminate within 10 execution steps" is a statically enforceable policy. We can use PM M to model the untrusted program and since M can only read 10 symbols from the input tape within the first 10 steps, then we can find a Turing machine $M_{\mathcal{P}}$ that simulate M with all the possible 10 inputs to see whether it is terminated within 10 steps. On the other hand, "The program terminates eventually" and "The program will write α eventually" are not statically enforceable.

2.2 Dynamic approaches

The basic idea for dynamic approaches is to write a monitor which runs in parallel with the untrusted program to enforce a given security property. It can enforce more security properties than static approaches according to [34], but in most cases, it requires more system resources. Dynamic and static approaches are often used together. SASI [22] is an example of the dynamic approach which benefits from static analysis to get balanced performance. To enforce security with limited resources, the author in [63] discussed how to improve the security of Java ME-CLDC related web applications and characterize dynamic enforcement with constrained memory. In [35], the author worked on developing a more powerful monitor and stated that a monitor that is allowed to modify its input is more powerful than one lacking this ability. Other examples of dynamic approaches could be found in [27, 43, 66]. In the remaining art of this section, we will present some examples for this approach and discuss their enforcement abilities.

2.2.1 SASI

In [22], authors provide a mechanism to transfer security policies into automata and enforce them on untrusted programs. An automaton is defined by a set of states, an input alphabet and a transition relation. A transition corresponds to a legal move of the program and when no transition is possible, then the program sequence should be rejected. According to [22], these automata are expressive enough to specify all the security policies that are enforceable by execution monitoring. Figure 2.3 indicates a security policy specified by an automaton.

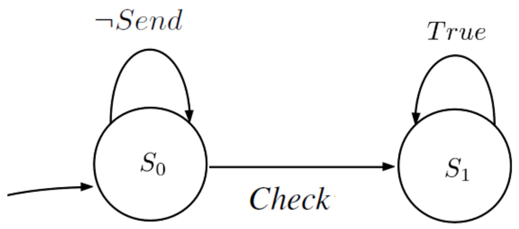

$$\begin{tikzcd} \arrow[r] & S_0 \arrow[loop above, "\neg Send"] \arrow[r, "Check"'] & S_1 \arrow[loop above, "True"] \end{tikzcd}$$

Figure 2.3: No Send Action before Check.

Merging Security Automaton

SASI has a rewriting system that inserts a new code before every action that accesses memory. This new code is generated according to the security automaton and ensure that each action satisfies the security policy. More precisely, the whole process contains four stages:

1. **Insert security automata:** put a copy of the security automaton before each instruction of the untrusted program.

2. **Evaluate transitions:** evaluate all the possible transitions in the automata.

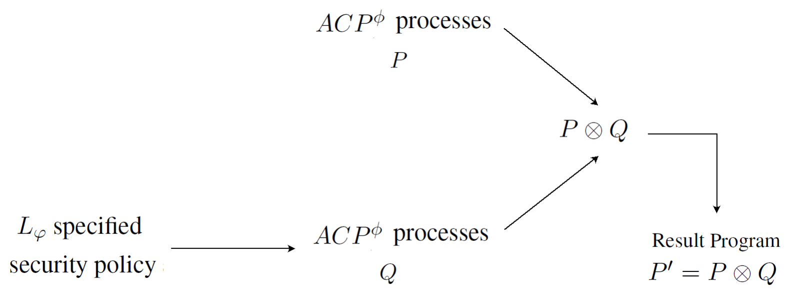
3. **Simplify automata:** remove all the transitions labeled as false.

4. **Compile automata:** transfer all the remaining security automata into codes that enforce the security policy. If an automaton has to reject the input, we transfer it into fail to stop the program sequence.

Example 2.2.1 *Suppose we have a program sequence: "$Print; Send; Check;$" and a security policy to ensure that there is no send action before check action (described in Figure 2.3). we use four steps to enforce our security policy as shown by Figure 2.4 (page 15).*

2.2.2 Security Enforcement for Concurrent Systems

In [43] authors give an approach that enforces security policies on concurrent systems dynamically. This approach introduces a modified version of ACP (algebra for communicating process in [8]) to specify untrusted programs and a modified version of the LTL logic denoted by L_φ to specify security policies. Then, this approach uses a function to translate the specified security policies into an ACP^ϕ process. The methodology is as following:



ACP^ϕ processes P
$P \otimes Q$
L_φ specified security policy
ACP^ϕ processes Q
Result Program $P' = P \otimes Q$

Figure 2.5: Methodology for the Approach in [43].

Program specification

The syntax of ACP^ϕ is presented in Table 2.1.

Table 2.1: Synatax of ACP^ϕ Formula from [43].

P	$::=$	$1 \mid \delta \mid a \mid P_1.P_2 \mid P_1 + P_2 \mid P_1 \|\|_\gamma P_2 \mid P_1 \mathbin{\|\!\lfloor}_\gamma P_2 \mid P_1 \|_\gamma P_2 \mid$
		$P_1^* P_2 \mid \partial_H(P) \mid \tau_I(P) \mid \partial_\phi^\xi(P)$

Here the merge operator $||_\gamma$ and the communication operator $|_\gamma$ are indexed by a communication function γ, which is defined as following according to [43].

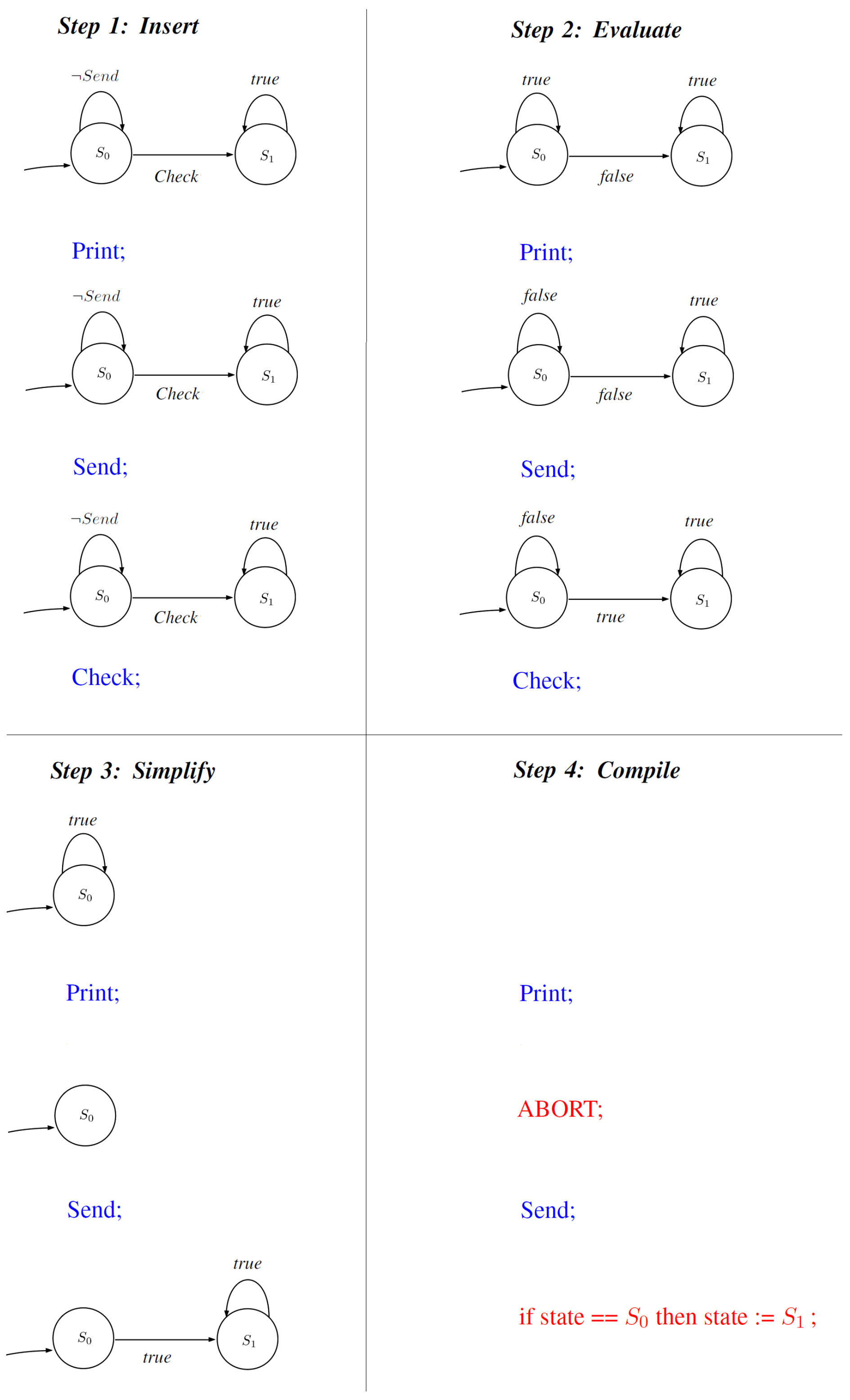


Figure 2.4: Four Steps of SASI.

Definition 2.2.2 (Communication function) *A communication function is any commutative and associative function form $\mathcal{A} \times \mathcal{A}$ to $\mathcal{A}$, i,e. $\gamma : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ is a communication function if:*

1. $\forall a, b \in \mathcal{A} : \gamma(a, b) = \gamma(b, a)$*, and*
2. $\forall a, b, c \in \mathcal{A} : \gamma(\gamma(a, b), c) = \gamma(a, \gamma(b, c))$

According to [43], the meanings of operators in Table 2.1 are as following:

- 1 means that the process has finished normally its executions.
- δ means that the process is in a deadlock state.
- Constants $a, b, c, ...$ are called atomic actions.
- $P_1 + P_2$ is a choice between two processes P_1 and P_2.
- $P_1.P_2$ is a sequential composition between P_1 and P_2.
- $P_1^* P_2$ is the process that behaves like $P_1.(P_1^* P_2) + P_2$. It is a binary version of the Kleene star operator [40].
- $P_1 ||_\gamma P_2$ is the process that executes P_1 and P_2 in parallel with the possibility of synchronization according to the function γ.
- $P_1 \|\!\lfloor_\gamma P_2$ is the process that first executes an action in P_1 and then run the remaining part of P_1 in parallel with P_2.
- $P_1 |_\gamma P_2$ is the merge of two processes P_1 and P_2 with the restriction that the first step is a communication between P_1 and P_2.
- ∂_H is a restriction operator, where H is a set of actions. The process $\partial_H(P)$ can evolve only by executing actions that are not in H.
- τ_I is an abstraction operator, where I is any set of atomic actions called internal actions. $\tau_I(P)$ will abstract all the output actions from P by the silent action τ, if these actions belongs to I.
- ∂_ϕ^ξ is an enforcement operator, where ϕ indicate a security policy and ξ is a trace of actions. $\partial_\phi^\xi(P)$ is the processes that can evolve only if P can evolve by actions that do not lead to the violation of the security policy ϕ.

According to [43], the semantics of ACP^ϕ is as given by Table 2.3 (page 17).

Table 2.3: Operational Semantics of ACP^{ϕ} According to [43].

$$(R_{\equiv})\ \frac{P{\equiv}P_1 \quad P_1\xrightarrow{a}P_2 \quad P_2{\equiv}Q}{P\xrightarrow{a}Q} \qquad\qquad (R^{a})\ \frac{\square}{a\xrightarrow{a}1}$$

$$(R_{.})\frac{P\xrightarrow{a}P'}{P.Q\xrightarrow{a}P'.Q} \qquad\qquad (R_{+})\frac{P\xrightarrow{a}P'}{P{+}Q\xrightarrow{a}P'}$$

$$(R_{*})\frac{P\xrightarrow{a}P'}{P^{*}Q\xrightarrow{a}P'.(P^{*}Q)} \qquad\qquad (R^{d}_{*})\frac{Q\xrightarrow{a}Q'}{P^{*}Q\xrightarrow{a}Q'}$$

$$(R_{\lfloor\!\lfloor_\gamma})\frac{P\xrightarrow{a}P'}{P\lfloor\!\lfloor_\gamma Q\xrightarrow{a}P'\lfloor\!\lfloor_\gamma Q} \qquad\qquad (R_{||_\gamma})\frac{P\xrightarrow{a}P'}{P||_\gamma Q\xrightarrow{a}P'||_\gamma Q}$$

$$(R^{\mathcal{C}}_{||_\gamma})\frac{P\xrightarrow{a}P' \quad Q\xrightarrow{b}Q'}{P||_\gamma Q\xrightarrow{\gamma(a,b)}P'||_\gamma Q'}\gamma(a,b)\neq\delta \qquad\qquad (R_{|_\gamma})\frac{P\xrightarrow{a}P' \quad Q\xrightarrow{b}Q'}{P|_\gamma Q\xrightarrow{\gamma(a,b)}P'||_\gamma Q'}\gamma(a,b)\neq\delta$$

$$(R^{\emptyset}_{\tau})\frac{P\xrightarrow{a}P'}{\tau_I(P)\xrightarrow{\tau}\tau_I(P')}a\in I \qquad\qquad (R_{\tau})\frac{P\xrightarrow{a}P'}{\tau_I(P)\xrightarrow{a}\tau_I(P)}a\notin I$$

$$(R_{\partial_H})\frac{P\xrightarrow{a}P'}{\partial_H(P)\xrightarrow{a}\partial_H(P')}a\notin H \qquad\qquad (R_{\partial^{\xi}_{\emptyset}})\frac{P\xrightarrow{a}P'}{\partial^{\xi}_{\emptyset}(P)\xrightarrow{a}\partial^{\xi.a}_{\emptyset}(P')}\quad \xi.a|\!\sim\emptyset$$

Security policy specification

In this section we introduce the specification language L_{φ} used in [43] to specify linear and temporal security properties. The syntax of L_{φ} is presented by the following BNF grammar:

$$\varphi_1,\varphi_2 ::= tt \mid f\!\!f \mid 1 \mid a \mid \varphi_1.\varphi_2 \mid \varphi_1\vee\varphi_2 \mid \varphi_1\wedge\varphi_2 \mid \neg\varphi_1 \mid \varphi_1^{*}\varphi_2$$

According to [43], the semantics of L_{φ} is as shown by Table 2.5.

Table 2.5: Semantics of L_{φ} Formula.

$$\begin{array}{lcl}
[\![tt]\!] & = & \mathcal{T} \\
[\![f\!\!f]\!] & = & \emptyset \\
[\![1]\!] & = & \{\epsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![\varphi_1.\varphi_2]\!] & = & \{\xi_1.\xi_2|\xi_1\in[\![\varphi_1]\!] \text{ and } \xi_2\in[\![\varphi_2]\!]\} \\
[\![\varphi_1\vee\varphi_2]\!] & = & [\![\varphi_1]\!]\cup[\![\varphi_2]\!] \\
[\![\varphi_1\wedge\varphi_2]\!] & = & [\![\varphi_1]\!]\cap[\![\varphi_2]\!] \\
[\![\varphi_1^{*}\varphi_2]\!] & = & \begin{cases} [\![\varphi_1]\!]^{*}\cup\{\xi_1.\xi_2|\xi_1\in[\![\varphi_1]\!]^{*} \text{ and } \xi_2\in[\![\varphi_2]\!]\}, & if[\![\varphi_2]\!]\neq\emptyset \\ [\![\varphi_1]\!]^{\omega}, & \text{elsewhere} \end{cases} \\
[\![\neg\varphi_1]\!] & = & \mathcal{T}\backslash[\![\varphi_1]\!]
\end{array}$$

Problem Formalization

In [43], authors introduce a new operator $\otimes$, and state that for a program P and a security policy ϕ, $P \otimes \phi = \partial_{\phi}^{\xi}(P)$. Here $P \otimes \phi$ is the enforcement result, i.e: only traces that respect the security policy ϕ in the program P are kept in $P \otimes \phi$, and $P \otimes \phi$ will not generate traces that are not originally in the program P. After that, they tried to find an equivalent version of $\partial_{\phi}^{\xi}(P)$ that uses classical operators of the ACP algebras. For that reason, they try to use the merge operator $||_{\gamma}$ to setup a monitor which enforce a security policies on concurrent systems. Their technique is based on the introduction of synchronization actions (commonly used in synchronization logic).

To better explain their technique, we need the following notations:

- Given a set of actions A, its corresponding synchronization set, denoted by $A_{\mathcal{C}}$ is: $A_{\mathcal{C}} = \bigcup\limits_{a \in A} \{a_d, a_f, \overline{a_d}, \overline{a_f}\}$.
- $\mathcal{A}_{\mathcal{C}}(P)$ returns the set of synchronization actions in the process P.
- For $a \in \mathcal{A}_{\mathcal{C}}$, $a^c = \sum\limits_{\alpha \in \mathcal{A}_{\mathcal{C}} \setminus \{a\}} \alpha$.
- $\mathcal{A}_{\mathcal{C}}^i$ is the indexed form of $\mathcal{A}_{\mathcal{C}}$, i.e: $\mathcal{A}_{\mathcal{C}}^i = \bigcup\limits_{a \in \mathcal{A}_{\mathcal{C}}} \{a^i\}$
- The set H_i is used to denote the set $\mathcal{A}_{\mathcal{C}}^i$.
- The set I_i is used to denote the set $\bigcup\limits_{\alpha \in \mathcal{A}_{\mathcal{C}}^i} \{\alpha | \overline{\alpha}\}$.
- The function γ_0 is defined as follows:
  $$\gamma_0(a, \overline{a}) = \begin{cases} a|\overline{a}, & if\, a \in \mathcal{A} \cup \mathcal{A}_{\mathcal{C}} \\ \delta, & \text{else} \end{cases}$$

For a security policy ϕ, each action a will be replaced by $\overline{a_d}.\overline{a_f}$ to capture the start and the end of the action a. The authors provide a function $\llbracket - \rrbracket$ to transfer a security policy specified in L_{φ} into a synchronization ACP^{ϕ} process as shown by Table 2.7 (page 19).

On the other hand, for a process, each action a will be replaced by $\overline{a_d}.a.\overline{a_f}$. A function $\lceil - \rceil$ that modify ACP^{ϕ} processes in this way is given by Table 2.9 (page 19).

Main result

In [43], authors introduce $\tau-bissimulation$ and use it to present an equivalence version of $\partial_{\varphi}^{\xi}(P)$.

Definition 2.2.3 *(τ–$bissimulation$) A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ over processes is a τ–$bissimulation$, if for all $(P, Q) \in S$ we have:*

Table 2.7: L_φ Translation Function According to [43].

$\lceil\!\lceil - \rceil\!\rceil : L^a_{N(\phi)} \times \mathbb{N} \to ACP$		
$\lceil\!\lceil tt \rceil\!\rceil_i$	$=$	$(\sum_{\alpha\in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha\in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1$
$\lceil\!\lceil ff \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil 1 \rceil\!\rceil_i$	$=$	1
$\lceil\!\lceil \delta \rceil\!\rceil_i$	$=$	δ
$\lceil\!\lceil a \rceil\!\rceil_i$	$=$	$\overline{a}^i_d.\overline{a}^i_f$
$\lceil\!\lceil \phi_1.\phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i.\lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1 \vee \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i + \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \phi_1^* \phi_2 \rceil\!\rceil_i$	$=$	$\lceil\!\lceil \phi_1 \rceil\!\rceil_i^* \lceil\!\lceil \phi_2 \rceil\!\rceil_i$
$\lceil\!\lceil \neg a \rceil\!\rceil_i$	$=$	$\overline{a}_d^{i\,c}.\overline{a}_f^{i\,c}\big((\sum_{\alpha\in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f)^* \sum_{\alpha\in A} \overline{\alpha}^i_d.\overline{\alpha}^i_f + 1\big)$

Table 2.9: ACP^ϕ Processes Translation Function According to [43].

$\lceil - \rceil : ACP^\phi \times \mathbb{N} \times 2^{\mathcal{A}} \to ACP^\phi$		
$\lceil 1 \rceil^H_i$	$=$	1
$\lceil \delta \rceil^H_i$	$=$	δ
$\lceil a \rceil^H_i$	$=$	$\begin{cases} a, & if\ \ a \in H \cup \{\tau\} \\ \overline{a_d}.a.\overline{a_f}, & \text{Else} \end{cases}$
$\lceil P_1.P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i.\lceil P_2 \rceil^H_i$
$\lceil P_1 + P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i + \lceil P_2 \rceil^H_i$
$\lceil P_1^* P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^{H*}_i \lceil P_2 \rceil^H_i$
$\lceil P_1 \parallel_{\gamma_0} P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \parallel_{\gamma_0} \lceil P_2 \rceil^H_i$
$\lceil P_1 \rVert\!\lfloor P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \rVert\!\lfloor \lceil P_2 \rceil^H_i$
$\lceil P_1 \mid P_2 \rceil^H_i$	$=$	$\lceil P_1 \rceil^H_i \mid \lceil P_2 \rceil^H_i$
$\lceil \partial_{H'}(P) \rceil^H_i$	$=$	$\partial_{H'}\big(\lceil P \rceil^{H\cup H'}_i\big)$
$\lceil \tau_I(P) \rceil^H_i$	$=$	$\tau_I\big(\lceil P \rceil^{H\cup I}_i\big)$
$\lceil \partial^{\xi}_{\bigwedge_{j\in 1..n}\varphi_j}(P) \rceil^H_i$	$=$	$\lceil \partial^{\xi}_{\bigwedge_{j\in 2..n}\varphi_j} \lceil \partial^{\xi}_{\varphi_1}(P) \rceil^H_i \rceil^{H\cup H_1}_{i+1}$
$\lceil \partial^{\xi}_{\varphi}(P) \rceil^H_i$	$=$	$\partial_{H_i}\Big(\tau_{I_i}\big(\lceil P \rceil^H_i \parallel_{\gamma_0} \lceil\!\lceil [\varphi]_\xi \rceil\!\rceil_i\big)\Big)$
where $H_1 = \mathcal{A}_{\mathcal{C}}(\lceil \partial^{\xi}_{\varphi_1}(P) \rceil^H_i)$		

1. *If $P \overset{a}{\twoheadrightarrow} P'$ then $Q \overset{a}{\twoheadrightarrow} Q'$ and $(P', Q') \in S$ and*

2. *If $Q \overset{a}{\twoheadrightarrow} Q'$ then $P \overset{a}{\twoheadrightarrow} P'$ and $(Q', P') \in S$ where $\overset{a}{\twoheadrightarrow} = (\overset{\tau}{\to})^* \overset{a}{\to} (\overset{\tau}{\to})^*$.*

Definition 2.2.4 *$\underline{\leftrightarrow}_\tau$ is defined as the biggest $\tau-bissimulation$:*

$$\underline{\leftrightarrow}_\tau = \cup\{S : S \textit{ is a } \tau-bissimulation\}$$

Finally, an equivalence version of $\partial^{\xi}_{\varphi}(P)$ is presented as following:

Theorem 2.2.5 ***(Main Theorem)*** *$\forall P \in ACP^{\phi}$, $\forall \varphi \in L^d_{N(\varphi)}$, and $\forall \xi \in \mathcal{T}$, we have:*

$$\partial^{\xi}_{\varphi}(P) \underline{\leftrightarrow}_\tau \partial_{H_i}\Big(\tau_{I_i}\big(\lceil P \rceil^H_i ||_{\gamma_0} \lceil\!\rceil [\varphi]_\xi \lceil\!\rceil_i\big)\Big) \textit{ for any } i \in \mathbb{N}$$

Since $P \otimes \phi = \partial^{\xi}_{\phi}(P)$, the problem of enforcing a security policy ϕ on a program P turn to run the ACP process $\partial_{H_i}\Big(\tau_{I_i}\big(\lceil P \rceil^H_i ||_{\gamma_0} \lceil\!\rceil [\varphi]_\xi \lceil\!\rceil_i\big)\Big)$. So when an end user provides a program P in ACP^{ϕ} and a security policy ϕ in L_φ, we can use functions $\lceil - \rceil$ and $\lceil\!\rceil - \lceil\!\rceil$ to transfer P and ϕ into ACP process P' and ϕ'. Then the communication operator $||_{\gamma_0}$ will make sure that the process $\partial_{H_i}\Big(\tau_{I_i}\big(P' ||_{\gamma_0} \phi'\big)\Big)$ generates only traces of P that satisfies the security policy ϕ.

2.2.3 Enforcement Ability of Dynamic Approach

EM Class, defined by [57], specifies a set of security policies that can be enforced by an execution monitor. In [34], authors reduce the size of EM by giving out more restrictions and introduce the co-recursively enumerable (coRE) properties.

coRE: A security policy $\mathcal{P}$ is coRE when there exists a Turing machine $M_{\mathcal{P}}$ that takes an encoding of the system (a Turing machine M belonging to PM) as input and rejects it in a finite time if the system does not comply with the security policy; otherwise $M_{\mathcal{P}}(M)$ loops forever. In other words, if a given program does not satisfy the security policy, we can find a Turing machine that rejects this program within a finite time, otherwise, the Turing machine runs forever.

In [34], Hamlen, Morrisett and Schneider proved that statically enforceable policies are a subset of coRE and give the example of $\mathcal{P}_{boot}$: if a system memory has a boot area, then the program should never write in it.

As shown by Figure 2.6, static analysis can not ensure that a program "never" write in the boot area, while execution monitoring can enforce $\mathcal{P}_{boot}$.

The relationship between statically enforceable and coRE policies is shown by Figure 2.6.

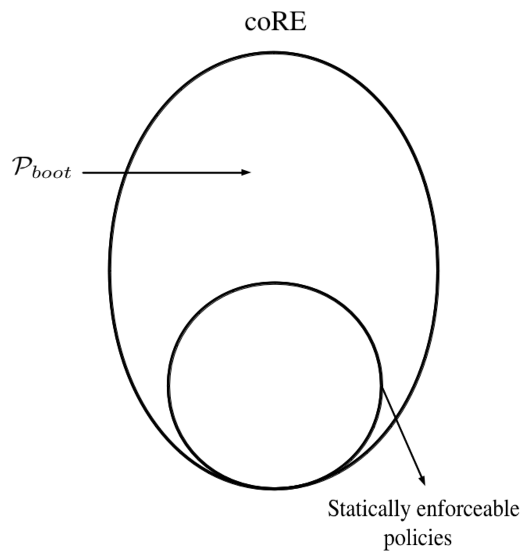


Figure 2.6: Statically Enforceable and coRE Policies According to [34].

2.3 Program Rewriting

An alternative approach to the two previous ones is program rewriting. The basic idea is to modify, during a period of time (finite), the untrusted program before its execution so that the new version respects a given security policy. Generally, it requires fewer system resources than dynamic analysis, since instead of monitoring the execution of a target program, it changes the code at some "critical points" to enforce the security policy. Actually in [34], authors showed that rewriting techniques can enforce some security policies, which static and dynamic approaches can not. Other examples of program rewriting include [18, 21, 59]. In the remaining part of this chapter, we will show some examples of this approach and discusses their enforcement abilities.

2.3.1 IRM Enforcement

In [21], authors provide a program rewriting mechanism with inlined reference monitors (IRM). For an application, a trusted IRM rewriter will merge checking codes into critical points of the program to enforce security policies. After the rewriting process, the untrusted program will become trusted and ready to be executed many times without rewriting again. The whole process of IRM enforcement in [21] is shown in Figure 2.7.

According to [21], the specification of IRM includes the following three parts:

- Security events: Program operations that engender the intervention of the security monitor.
- Security state: Information about previous security events that are used to determine which security events are allowed to be executed.

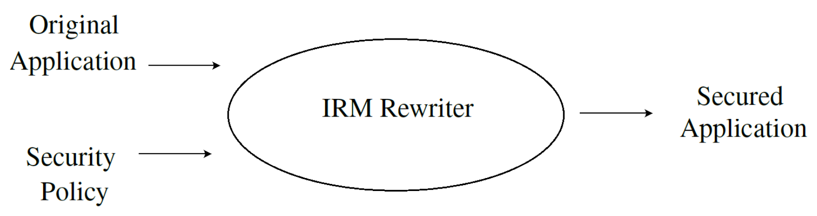


Figure 2.7: IRM Enforcement Process According to [21].

- Security updates: Programs that enforce security policies according to the information from security events and security state (e.g block executions, alert for security violations).

2.4 Enforcing Security Policies Using An Algebraic Approach

In this section, we introduce an approach developed by Mejri and Fujita in [48] to enforce security policies by program rewriting. Our research contributions introduced in the next chapters are based on this approach and the one work of Ould-Slimane in [30] is also presented.

This section is organized as follows. First, we introduce $BPA^*_{0,1}$. According to [48], this is used to specify untrusted programs as well as security policies. Second, we introduce the notion of greatest common factor (gcf) of two processes and we show how it is used to formalize the problem. Third, we introduce the algorithm presented in [48] to compute the gcf of two processes P and Q. Finally, we present some extensions of this approach done by Hakima Ould-Slimane in [30].

2.4.1 $BPA^*_{0,1}$: A Process Specification Language

In this section, we give the syntax and the semantics of $BPA^*_{0,1}$ [9, 10, 48], the language that will be used to specify both the program and the security policy. Secondly, we introduce the notion of trace equivalence between processes and we give some algebraic properties related to this relation.

Syntax of $BPA^*_{0,1}$

Suppose that a is an action that ranges over a finite set of actions Σ. the syntax of $BPA^*_{0,1}$ is defined by the following BNF-grammar.

$$x, y ::= 0 \mid 1 \mid a \mid x + y \mid x.y \mid x^*y$$

Semantics of $BPA^*_{0,1}$

The semantics is defined by the transition relation $\rightarrow \in \mathcal{P} \times \sum \times \mathcal{P}$ given by Table 2.11.

- $\downarrow$ means that a process can end normally: it can be reduced to 1.
- 0 means that the process is in a deadlock state.
- 1 means that the process has finished normally its executions.

Table 2.11: Operational Semantics of $BPA^{*}_{0,1}$.

$(R^{a})\dfrac{\square}{a\xrightarrow{a}1}$	$(R^{1})\dfrac{\square}{1{\downarrow}}$
$(R^{*}_{r\downarrow})\dfrac{y{\downarrow}}{(x^{*}y){\downarrow}}$	$(R^{\cdot}_{\downarrow})\dfrac{x{\downarrow}\quad y{\downarrow}}{(x.y){\downarrow}}$
$(R^{\cdot}_{l})\dfrac{x{\downarrow}\quad y\xrightarrow{a}y'}{x.y\xrightarrow{a}y'}$	$(R^{\cdot}_{r})\dfrac{x\xrightarrow{a}x'}{x.y\xrightarrow{a}x'.y}$
$(R^{+}_{l\downarrow})\dfrac{x{\downarrow}}{(x+y){\downarrow}}$	$(R^{+}_{r\downarrow})\dfrac{y{\downarrow}}{(x+y){\downarrow}}$
$(R^{+}_{l})\dfrac{x\xrightarrow{a}x'}{x+y\xrightarrow{a}x'}$	$(R^{+}_{r})\dfrac{y\xrightarrow{a}y'}{x+y\xrightarrow{a}y'}$
$(R^{*}_{l})\dfrac{x\xrightarrow{a}x'}{x^{*}y\xrightarrow{a}x'.(x^{*}y)}$	$(R^{*}_{r})\dfrac{y\xrightarrow{a}y'}{x^{*}y\xrightarrow{a}y'}$

- $x+y$ is a choice between two processes x and y.
- $x.y$ is a sequential composition between x and y.
- $x^{*}y$ is the process that behaves like $x.(x^{*}y)+y$. It is a binary version of the Kleene star operator [40].

Trace Based Equivalence

According to the works of [48], we introduce the trace based equivalence in this section. The equivalence relation between different program processes is an interesting topic for our further discussion, and it is based on the relation $\twoheadrightarrow$, defined in [48] as following:

$$\frac{\square}{x\overset{\epsilon}{\twoheadrightarrow}x}$$

$$\frac{x\overset{\tau}{\twoheadrightarrow}x'\qquad x'\overset{a}{\rightarrow}x''}{x\overset{\tau.a}{\twoheadrightarrow}x''}$$

Based on $\twoheadrightarrow$, the authors introduced, in [48], the ordering relation $\sqsubseteq_T$ as following:

Definition 2.4.1 *($\sqsubseteq_T$) Let x, x', y be processes and $\tau \in \sum^{*}$. We say that $x \sqsubseteq_T y$, if $x \overset{\tau}{\twoheadrightarrow} x'$ then there exists y' such that $y \overset{\tau}{\twoheadrightarrow} y'$*

Now we can define trace based equivalence $\sim$ form $\sqsubseteq_T$.

Definition 2.4.2 *($\sim$) We say that two processes x and y are trace equivalent and we write $x \sim y$, if $x \sqsubseteq_T y$ and $y \sqsubseteq_T x$.*

Hereafter, we give some other useful properties of $\sim$:

Proposition 2.4.3 *Given three processes x, y and z in $BPA^*_{0,1}$, then the following properties hold.*

(B_1)	$x+(y+z)$	$\sim$	$(x+y)+z$	(B_6)	$0.x$	$\sim$	0		
(B_2)	$x.(y.z)$	$\sim$	$(x.y).z$	(B_7)	$x.1$	$\sim$	$1.x$	$\sim$	x
(B_3)	$x+y$	$\sim$	$y+x$	(B_8)	$x+0$	$\sim$	x		
(B_4)	$(x+y).z$	$\sim$	$x.z+y.z$	(B_9)	x^*y	$\sim$	$y+xx^*y$		
(B_5)	$x.(y+z)$	$\sim$	$x.y+x.z$	(B_{10})	$x+x$	$\sim$	x		

Proof: Directly from the definition of $\sim$ and $\sqsubseteq_T$

2.4.2 Formalization of the Problem

Given a program P and a security policy Φ, the goal of this work is to generate another program P' that respects the security policy Φ and generates no more traces than the original program P. More precisely, we want that the result P' respects the two following properties, stated in [48]:

1. **Correctness:**
   - $P' \sqsubseteq P$: all the traces of P' are traces in P
   - $P' \sqsubseteq \Phi$: all the traces of P' respect the security policy.
2. **Completeness:** If there exists P" such that P" $\sqsubseteq P$ and P" $\sqsubseteq \Phi$, then P" $\sqsubseteq P'$. This properties involves that all the traces in P that respect the security policy are also in P'.

The definition of the greatest common factor or gcf is given in [48] as following:

Definition 2.4.4 (Greatest Common Factor (gcf)) *Let P and Q be two processes. The gcf of P and Q, denote by $P \sqcap Q$, is a process R such that the following conditions hold:*

- $R \sqsubseteq P$
- $R \sqsubseteq Q$
- For all R' such that $R' \sqsubseteq P$ and $R' \sqsubseteq Q$, we have $R' \sqsubseteq R$.

The problem of enforcing a security property Q on a program P turns to find $P \sqcap Q$, the gcf of P and Q. In the next section, we will introduce an algorithm allowing to compute gcf, but before that, we recall some useful properties of $\sqcap$ given in [48].

Proposition 2.4.5 *Let P, Q and R be processes, a and b be two different actions. The following properties hold:*

$$\begin{aligned}
1 \sqcap a &\sim 0 \\
P \sqcap P &\sim P \\
P \sqcap Q &\sim Q \sqcap P \\
a.P \sqcap a.Q &\sim a.(Q \sqcap P) \\
a.P \sqcap b.Q &\sim 0.(Q \sqcap P) \sim 0 \\
P \sqcap (Q + R) &\sim P \sqcap Q + P \sqcap R \\
if\ P \sim P' then\ P \sqcap Q &\sim P' \sqcap Q
\end{aligned}$$

2.4.3 Resolution of the Problem

Given two processes P and Q, according [48], we introduce a way, based on the notion of derivatives introduced by Brzozowski in [11], to compute $P \sqcap Q$. The idea is to generate, using the notion of derivatives, a linear system where its resolution gives $P \sqcap Q$.

Derivatives

In [48], the authors adapt the definition of derivatives introduced by Brzozowski in [11] to fit it with the notion of trace equivalence.

Definition 2.4.6 *The derivative of a process x with respect to an action a, denoted by $\partial_a(x)$, is defined as following:*

$$\begin{aligned}
\partial_a(0) &= 0 \\
\partial_a(1) &= 0 \\
\partial_a(a) &= 1 \\
\partial_a(b) &= 0 \\
\partial_a(x^*y) &= \partial_a(x).x^*y + \partial_a(y) \\
\partial_a(x + y) &= \partial_a(x) + \partial_a(y) \\
\partial_a(x.y) &= \partial_a(x).y + o(x).\partial_a(y)
\end{aligned}$$

Intutively, $\partial_a(x)$ = { $x' |\ x \overset{a}{\rightarrow} x'$ }.

Definition 2.4.7 *The function $o(x) : process \rightarrow \{0,1\}$, allows to know whether $x\downarrow$ holds or not and it is defined as follows:*

$$\begin{aligned} o(0) &= 0 \\ o(1) &= 1 \\ o(a) &= 0 \\ o(x^*y) &= o(y) \\ o(x+y) &= o(x)+o(y) \\ o(x.y) &= o(x).o(y) \end{aligned}$$

$x\downarrow$ shows weather the process x can immediately finish as shown by Definition 2.4.1.

Definition 2.4.8 *We define the function δ from $\mathcal{P}$ to 2^{Σ} as follows:*

$$\begin{aligned} \delta(0) &= \emptyset \\ \delta(1) &= \emptyset \\ \delta(a) &= \{a\} \\ \delta(x^*y) &= \delta(x)\cup\delta(y) \\ \delta(x+y) &= \delta(x)\cup\delta(y) \\ \delta(x.y) &= \delta(x)\cup o(x)\otimes\delta(y) \end{aligned}$$

Intuitively, $\delta(x) = \{a|\ \ x \overset{a}{\rightarrow} x'\}$ where $\otimes$ is defined as following:

$$\begin{aligned} \otimes : \{0,1\}\times 2^{\Sigma} &\longrightarrow 2^{\Sigma} \\ (0,\mathcal{S}) &\mapsto \emptyset \\ (1,\mathcal{S}) &\mapsto \mathcal{S} \end{aligned}$$

Two interesting propositions were introduced in [48], which are important for writing the algorithm allowing to compute the gcf.

Proposition 2.4.9 $x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$

Proposition 2.4.10 $x \sqcap y \sim o(x)\times o(y)$ + $\sum_{a\in\delta(x)\cap\delta(y)} a.(\partial_a(x)\sqcap\partial_a(y))$

Algorithm

From proposition of 2.4.10 and the properties of $\sim$, an algorithm allowing to compute $P \sqcap Q$, by generating a linear system and resolving it, is introduced in [48] as following:

Algorithm 1 calculate $P \sqcap Q$ in $BPA^*_{0,1}$ where $E = \emptyset$

```
1: $E \longleftarrow E \cup \{P \sqcap Q \sim o(P) \times o(Q) + \sum_{a \in \delta(P) \cap \delta(Q)} a.(\partial_a(P) \sqcap \partial_a(Q))\}$
2: while there exists $P_i \sqcap Q_i$ in the right side of any equation in $E$ that does
   not appear (modulo commutativity of $\sqcap$) in the left side on any equation do
     $E \longleftarrow E \cup \{P_i \sqcap Q_i \sim o(P_i) \times o(Q_i) + \sum_{a \in \delta(P_i) \cap \delta(Q_i)} a.(\partial_a(P_i) \sqcap \partial_a(Q_i))\}$
   end while
3: Return the solution of the linear system $E$.
```

The linear system that we obtain at the end has the following form $AX + B = X$ where A is a constant matrix of size $n \times n$, B is a constant vector of size n and X is a vector of variables of size n. The solution of this system is $X = A^*B$, where A^* is computed as following:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a + bd^*c)^* & (a + bd^*c)^*bd^* \\ (d + ca^*b)^*ca^* & (d + ca^*b)^* \end{bmatrix}$$

and the result can be inductively generalized for matrices n by n as following, where A is an $n-1$ by $n-1$ matrix:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^* = \left[\begin{array}{c|c} (A + BD^*C)^* & (A + BD^*C)^*BD^* \\ \hline (D + CA^*B)^*CA^* & (D + CA^*B)^* \end{array}\right]$$

These result hold because, under the trace equivalence, $BPA^*_{0,1}$ is a kind of monodic tree Kleene algebra [62] with a binary star Kleene operator.

Example

Let $P = c.(a.b + c)^*0 = c.P_1$ and $Q = (c.(a + (a.b)^*))^*0$

$$\begin{aligned}
P \sqcap Q = X_1 &= c.(P_1 \sqcap (a + (a.b)^*).Q) = c.X_2 \\
X_2 &= (a.b + c).P_1 \sqcap (a + (a.b)^*).Q) \\
&= a.(b.P_1 \sqcap (1 + b.(a.b)^*1).Q) + c.(P_1 \sqcap (a + (a.b)^*1).Q) \\
&= a.X_3 + c.X_2 \\
X_3 &= b.(P_1 \sqcap (a.b)^*1.Q) = b.X_4 \\
X_4 &= a.(b.P_1 \sqcap (b.(a.b)^*1).Q) + c.(P_1 \sqcap (a + (a.b)^*1).Q) \\
&= a.X_5 + c.X_2 \\
X_5 &= b.(P_1 \sqcap (a.b)^*1.Q) = b.X_4
\end{aligned}$$

$X_5 = X_3$
$X_4 = a.X_3 + c.X_2 = X_2$
$X_3 = b.X_4 = b.X_2$
$X_2 = (a.b + c).X_2$
$X_2 = (a.b + c)^*0$
$X_1 = c.X_2 = c.(a.b + c)^*0$

So the greatest common factor (gcf) of P and Q is $c.(a.b + c)^\omega$, where P^ω means P^*0.

2.4.4 An Extension for $BPA^*_{0,1}$

$EBPA^*_{0,1}$ introduced in [30] by Ould-Slimane is an extension based on $BPA^*_{0,1}$. It provides a way to specify the suspension or the delay of the execution of actions. And based on this new algebra, a corresponding new algorithm is developed to enforce some interesting security properties. The syntax and the semantics of $EBPA^*_{0,1}$ are respectively introduced in the following sections.

Syntax of $EBPA^*_{0,1}$

Suppose that a is an action that ranges over a finite set of actions σ and two processes x and y. The syntax of $EBPA^*_{0,1}$ is defined by the following BNF-grammar.

$$x, y ::= 0 \mid 1 \mid a \mid a^+ \mid a^- \mid x + y \mid x.y \mid x^*y$$

This syntax can be used to specify both programs and properties where:

- 0 means that the process is in a deadlock state.
- 1 means that the process has finished normally its executions.

- a means that the execution of an action a.
- a^- means that the suspension of the execution of an action a.
- a^+ means that the execution of previously suspended actions followed by the execution of the action a.
- $x + y$ is a choice between two processes x and y.
- $x.y$ is a sequential composition between x and y.
- x^*y is the process that behaves like $x.(x^*y)+y$. It is a binary version of the Kleene star operator [40].

Semantics of $EBPA^*_{0,1}$

The semantics is defined by the transition relation $\rightarrow \in (\mathcal{P} \times \Sigma) \times \Sigma \times (\mathcal{P} \times \Sigma)$ given by Table 2.14.

- $\downarrow$ means that a process can end normally: it can be reduced to 1.
- τ is a sequence of actions, it denotes a history maintained by each process. Rules (R^{a^-}), and (R^{a^+}) show how actions can be suppressed and executed later.

Table 2.14: Operational Semantics of $EBPA^*_{0,1}$.

$(R^a)\frac{\square}{a,\tau\xrightarrow{a}1,\tau}$	$(R^1)\frac{\square}{1\downarrow}$
$(R^{a^+})\frac{\square}{a^+,\tau\xrightarrow{\tau.a}1,\epsilon}$	$(R^{a^-})\frac{\square}{a^-,\tau\xrightarrow{\epsilon}1,\tau.a}$
$(R^*_{r\downarrow})\frac{Q\downarrow}{(P^*Q)\downarrow}$	$(R^{\cdot}_{\downarrow})\frac{P\downarrow \quad Q\downarrow}{(P.Q)\downarrow}$
$(R^{\cdot}_l)\frac{P,\tau\xrightarrow{\sigma}1,\tau' \quad Q,\tau'\xrightarrow{\sigma'}Q',\tau''}{P.Q,\tau\xrightarrow{\sigma.\sigma'}Q',\tau''}$	$(R^{\cdot}_r)\frac{P,\tau\xrightarrow{\sigma}P',\tau'}{P.Q,\tau\xrightarrow{\sigma}P'.Q,\tau'}$
$(R^+_{l\downarrow})\frac{P\downarrow}{(P+Q)\downarrow}$	$(R^+_{r\downarrow})\frac{Q\downarrow}{(P+Q)\downarrow}$
$(R^+_l)\frac{P,\tau\xrightarrow{\sigma}P',\tau'}{P+Q,\tau\xrightarrow{\sigma}P',\tau'}$	$(R^+_r)\frac{Q,\tau\xrightarrow{\sigma}Q',\tau'}{P+Q,\tau\xrightarrow{\sigma}Q',\tau'}$
$(R^*_l)\frac{P,\tau\xrightarrow{\sigma}P',\tau'}{P^*Q,\tau\xrightarrow{\sigma}P'.(P^*Q),\tau'}$	$(R^*_r)\frac{Q,\tau\xrightarrow{\sigma}Q',\tau'}{P^*Q,\tau\xrightarrow{\sigma}Q',\tau'}$

2.4.5 Enforcement Ability

Intuitively, $EBPA^*_{0,1}$ is more expressive than $BPA^*_{0,1}$, therefore the new approach developed by Hakima Ould-Slimane should be able to enforce more security properties than the original one.

2.4.6 Conclusion

In this section, we introduced the algebraic language $BPA^*_{0,1}$ used in [48] to specify programs and security properties. Then, we show how it has been used to automatically enforce a security policy on a program. The enforced program obtained by this approach is correct and complete with respect to the original one and the security policy. Finally, we give an extension of this approach which was introduced in [30].

2.5 Enforcement Ability of Program Rewriting

According to [34], a security policy $\mathcal{P}$ is **RW-enforceable** if there exists a computable rewriter function $R : PM \to PM$ such that for all PM's M:

- $\mathcal{P}(R(M))$; the transformation of the original program still satisfy $\mathcal{P}$.
- And if the original PM already satisfy $\mathcal{P}$, then the transformed PM is equivalent to the original one.

2.5.1 Program Rewriting and Static Analysis

In [34], it has been shown that $\mathcal{P}_{boot}$ is an example of properties that are RW-enforceable, but not statically enforceable. A rewriting function can enforce $\mathcal{P}_{boot}$ by taking PM M as input and returning a new PM M'. M' works exactly like M, except that for every step of M, M' check one step ahead for every possible input. If an invalid action is detected, M' terminate immediately. Intuitively, program rewriting is more powerful than static analysis and statically enforceable policies should be therefore a subset of RW-enforceable. But there is a special case, unsatisfied policy (*false* which reject all the possible executions), for which the rewriter function R cannot return a system that satisfy it in this model. We call this policy $\mathcal{P}_{unsatisfy}$. The relationship between RW-enforceable and statically enforceable policies is shown by Figure 2.8.

2.5.2 Program Rewriting and Execution Monitoring

In [34], Hamlen, Morrisett and Schneider state that there are some policies is RW-enforceable which are not in coRE. Secret File policy is one of them:

Suppose that inside a file system, there is a file that should be kept secret from untrusted programs. And the untrusted programs can access to the list of directories that contains the secret file. The security policy stipulates that the untrusted programs should not know the existence of the secret file.

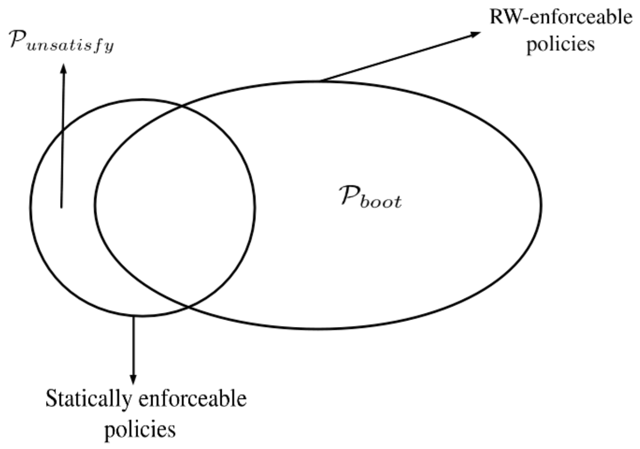


Figure 2.8: Statically Enforceable and RW-enforceable According to [34].

An untrusted program satisfying this policy should behave identically as if the secret file does not exist in the system.

In [34], authors state that deciding whether a given untrusted program behave equivalently on two arbitrary inputs is as hard as deciding whether two untrusted programs are equivalent to each other on all inputs. In [34], untrusted programs are modeled by PM and authors prove that the equivalent of PMs is not coRE. Thus, secret file policy can not be enforced by execution monitors. Authors also state another reason for this, an EM can not enforce this policy by parallel simulation of the untrusted PM on two different inputs, one that includes the secret file and the one that does not. This is because the EM must detect policy violations within finite time on each computational step of the untrusted program, but the process of deciding equivalence may take an infinite of time. This is because executions can be equivalent even if they are not equivalent in each step.

On the other hand, according to [34], program rewriting can enforce this policy, since the program rewriting function never needs to decide whether the secret policy has been violated or not by the untrusted program, it only needs to make sure that the changed program satisfy the security policy. For example, when an untrusted program trying to get a directory list which include the secret file, the rewriting function can change this list and remove the secret file's name from it.

2.5.3 More About Execution Monitor Enforceable Policies

Intuitively, RW-enforceable policies should build an upper-bound for execution monitor enforceable (EM enforceable) policies. But in [34], authors show that there are some policies in coRE (EM_{orig}) but not in the set of RW-enforceable policies and state that these policies can be enforced by execution monitors neither. This interesting result indicates that the four requirements defining the set of EM_{orig} are limited to specify EM enforceable policies. So in the following, we firstly introduce these policies in coRE (EM_{orig}) but can not be enforced by program rewriting. Secondly we provide a new formal definition of EM enforceable policies.

We cannot enforce some security policies when we don't have the ability to alter the target program. For example, let I be the set of all the possible interventions of the monitor and let $\mathcal{P}_I$ be the property stating that all the actions in I are not allowed. This will bring contradiction when a monitor tries to use actions in I to enforce the security policies. Schneider mention that the monitor can only terminate the program once it violates the security policy in the definition of class EM. However, if the security policy states that the program should not terminate, this policy is in EM_{orig}, since it not against the four requirements defining EM_{orig}, but it can not be enforced by monitors defined by Schneider in class EM. In [34], authors showed that execution monitors' abilities are also limited if they can not modify the program in time to enforce the security policy.

In [34], the set of monitoring enforceable policies is defined as the intersection of coRE and RW-enforceable. In [34], authors introduce a new term "benevolent" to describe the EM enforceable policies as following.

$\hat{\mathcal{P}}$ is a predicate over executions. The security policy $\mathcal{P}$ induced by $\hat{\mathcal{P}}$ is defined by:

$$\mathcal{P}(M) = (\forall \sigma : \sigma \in X_M : \hat{\mathcal{P}}(\sigma))$$

where X_M denotes the set of all possible executions exhibited by the PM M. A PM M satisfies a security policy $\mathcal{P}$ if and only if all the possible executions satisfy the predicate $\hat{\mathcal{P}}$.

Finally, the relationship between the different security policies and their enforcement mechanism is resumed by Figure 2.9.

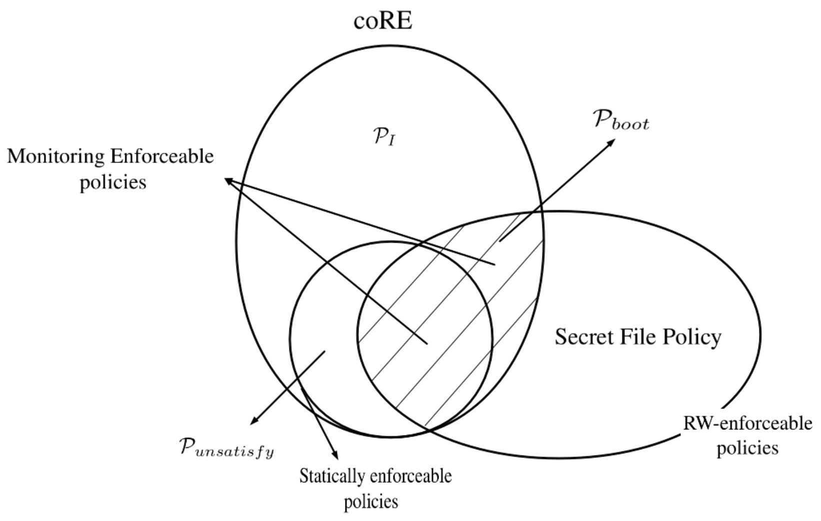

coRE
$\mathcal{P}_I$
$\mathcal{P}_{boot}$
Monitoring Enforceable policies
Secret File Policy
RW-enforceable policies
$\mathcal{P}_{unsatisfy}$
Statically enforceable policies

Figure 2.9: Classes of Security Policies According to [34].

2.6 Conclusion

This chapter discussed different security enforcement approaches (Statics, Dynamic approach and Program Rewriting) as well as their advantages and disadvantages. Static approaches aim to check codes before their executions. Generally, they require fewer system resources, since they are applied offline. However, they can not enforce some a security policy when a run-time information is needed. Execution monitoring is an example of the dynamic approach, it is usually used together with static approaches to get balanced performance. Program rewriting technique adopts the advantage of the two previous approaches to change the program before its execution to enforce the security policy. We also introduce the security policies classes that can be enforced by these approaches as explained in [34].

Furthermore, we highlighted that, in [34], Hamlen, Morrisett and Schneider provide an interesting way to model different security approaches and use it to discuss the relationship between security policy classes for different enforcement approaches.

Based on the knowledge of different enforcement approaches, we will discuss the classes of enforceable security policies in the next chapter.

Chapter 3

Classes of Enforceable Security Policies

3.1 Introduction

In this chapter, based on the works [41, 57], we give, in Section 3.2, the definitions of a security policy and a security property. In Section 3.3, we discuss a method introduced in [4] about how to separate a given security property into a liveness and safety parts. We also introduce the safety and the liveness property in a topology view. In Section 3.4, we talk about the Execution Monitoring (EM) introduced by [57] and its power. In Section 3.5 and Section 3.6, we discuss how we can enlarge the class of enforced properties by extending the power of EM. In Section 3.7, we introduce the hyperproperties which are security policies that are not properties. Finally, we introduce a Temporal Logic for the hyperproperties in Section 3.8.

3.2 Security Policy and Property

Understanding and formalizing the classes of security policies that we can enforce using a given mechanism is a fundamental result for any research related to the enforcement problem. In the following, we formalize the security policies and the security properties based on the works [41, 57] and we discuss the differences between them.

3.2.1 Security Policy

In [41], authors formalize a security policy as a predicate P on sets of executions Σ. A set of executions Σ satisfies a policy P if and only if $P(\Sigma)$. In another word, security policies stipulate whether a given set of executions can be accepted or not. According to [57], the most common program policies can be classified into the following categories:

- Access Control policies: Define what kinds of actions can access to some particular resources (memory area, file, etc.) of the system.

- Availability policies: They state intuitively that whenever a resource is used by a process, it should be released later.

- Information flow policies, which have been addressed by different works including [13] and [15], specify requirements related to information that can be learned by the users of a given system.

3.2.2 Security Property

In [41], authors give a formal definition as following:

A security policy $P(\Sigma)$ is a property if it can be specified by a predicate of the form:

$$P(\Sigma) = \forall\sigma \in \Sigma.\hat{P}(\sigma)$$

where Σ is a set of executions and $\hat{P}$ is a predicate computable on individual execution.

Hereafter we adopt the following notions introduced in [4].

A history h for a program should be viewed as an infinite sequence $\sigma = s_0 s_1...$, where s_0 is the initial state of the program and each state after that is obtained by the execution of an atomic action on the previous one. We use $\sigma[..i]$ to denote the first i states of σ and $\alpha\beta$ to denote the concatenation of two program sequences α and β. All histories could be seen as infinite sequences (for a finite execution, we simply repeat the final state infinitely).

A property is a set of infinite sequences of program states. We say that a program satisfies a property, if all program's histories are in this property. We use $P(\sigma)$ to denote that the infinite sequence σ is in the property P. A program satisfies a property P if for each of its history h, we have $P(h)$.

As we can see a security property is a prediction on each execution in the set of histories Σ, while security policy is a prediction on a set of executions. So based on this definition, security properties is a subset of security policies.

We deduce also that information flow policies are not always security properties, because they can link different executions together. *Noninterference*, as defined by Goguen and Meseguer in [26], is a special case of information flow policies. It states that actions caused by users at hight security level should have no effect on the observations of users at low security levels. To enforce this security policy, we need to compare different users' execution traces, so it is not a security property.

Service Level Agreement (SLA) discussed in [15] specifies the acceptable performance of a system. It can be a security policy or a security property. For example, if a policy states: "the average response time of all executions should be less than 1 second.", it is not a security property, since it needs to record every execution's response time to get the average result. But, if the policy requires that the maxim response time is less than 1 second, it becomes a security property.

3.3 Recognizing Safety and Liveness

In [4], Alpern and Schneider provide an interesting way to separate a given property into a safety and a liveness ones thanks to Buchi automaton. This important work can be used to know whether a property is enforceable, not enforceable or just partly enforceable by a given enforcement mechanism. For example, EM can only enforce safety properties and edit automata can enforce some liveness properties besides safety ones as it will be discussed in following sections.

In the remaining part of this section, we firstly introduce the Buchi automaton. Secondly, we show how it can be used to specify both safety and liveness properties. Finally, we point out how we can extract liveness and safety part from a given property.

3.3.1 Buchi Automata

One important method for studying security properties is to represent them as security automata. A Buchi automaton, which introduced first in [20], is a widely used example to specify these kind of properties.

A Buchi automaton m, which has finite sates and accepts infinite sequences can be formally defined as a five-tuple (S, Q, Q_0, Q_F, δ), where:

- S is the alphabet (a set of labels) of m.
- Q is the set of the states of m.
- $Q_0 \subseteq Q$ is the set of the start states of m.
- $Q_F \subseteq Q$ is the set of the accepting states of m.
- $\delta \in (Q \times S) \rightarrow Q$ is the transition function of m.

An infinite sequence of states is accepted if it contains accepting state infinitely often. The set of sequences accepted by the Buchi automaton m are denoted by $L(m)$. And we say that m is reduced if from every state there is a path to an accepting one. For a given reduced automaton m, if we convert every state of it into accepting one, then we obtain a closure $cl(m)$ of it.

An example of Buchi automaton is shown by Figure 3.1.

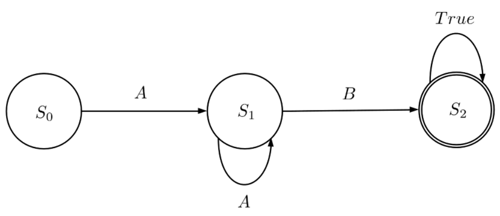


Figure 3.1: Example of Buchi Automaton (B_{action}).

3.3.2 Safety and Liveness Properties

Safety and liveness properties are two important subclasses of security properties. Here, we show that they can be specified by different forms of Buchi automatons. Also, we introduce the separation of a given security property into its safety and liveness parts.

Safety

A safety property states that "bad thing" never happens during the entire program execution (it is an invariant property). Once a finite sequence violates a safety property, we cannot add something to this sequence to amend this violation. Formal definition according to [3] is as follows:

P is a safety property if and only if:

$$(\forall \sigma : \sigma \in S^\omega : \neg P(\sigma) \Rightarrow (\exists i : i \geq 0 : (\forall \beta : \beta \in S^\omega : \neg P(\sigma[..i]\beta))))$$

. For example, if a security property states that: "For every ATM machine, a correct PIN number is required before money withdraw", then once a "bad thing" (using incorrect PIN number to withdraw money) take place, there is nothing we can add to the program trace to satisfy the security property again.

Other examples include the deadlock free property which states that a deadlock never happens, the first-come-first-serve property which states that "incorrect sequences of serving" never happens and the mutual exclusion property which state that no critical section can be used by more than one process.

The closure of a Buchi automaton can be used to specify safety properties. If m is a Buchi automaton and its closure $cl(m)$ accepts the same language ($L(m) = L(cl(m))$), then m recognizes a safety property. This is because $cl(m)$ rejects a sequence only when there is an undefined action ("bad thing"). It never reject because of failing to enter an accepting state (lack of "good things").

Liveness

A liveness property states that a "good thing" happens during the execution of the program execution (it captures well-foundedness). From any program state, we can always "do something", so that the property will be respected. A formal definition of liveness, according to [3], is as follows:

P is a liveness property if and only if:

$$(\forall \alpha : \alpha \in S^* : (\exists \beta : \beta \in S^\omega : P(\alpha\beta)))$$

. For example, if we have the following security property: "In a program C, if some part of a memory is reserved, it should be released sometime later". If for a finite sequence, a segment of memory is taken and it is not released, we can always add a "release action" to it and correct the situation.

Other examples include termination property and starvation freedom property.

For a liveness property m, since the $cl(m)$ can reject an input action only when it represents "bad thing", then $cl(m)$ must accept all the possible input program sequences, i.e: $L(cl(m)) = S^\omega$.

3.3.3 Partitioning Safety and Liveness Properties

Some properties are neither safety nor liveness properties, but it can be separated, using Buchi automaton, into a safety and a liveness parts, as it has been proven in [4].

For a given Buchi automaton m, which specifies a property, we can build a $Safe(m)$ and a $Live(m)$ automata that represent the safety and liveness parts of m. The closure of m, $cl(m)$ gives $Safe(m)$. For $Live(m)$, we build the automaton such that $L(Live(m)) = L(m) \cup (S^\omega - L(cl(m)))$, i.e:

$$L(Safe(m)) = L(cl(m))$$

$$L(Live(m)) = L(m) \cup (S^\omega - L(cl(m)))$$

$$L(Safe(m)) \cap L(Live(m)) = L(m)$$

The safety part of the Buchi automaton B_{action} given by Figure 3.1 is shown by Figure 3.2 and it is obtained by transforming every state into a final one.

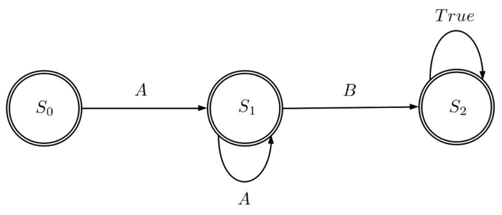


Figure 3.2: $Safe(B_{action})$.

To build $Live(B_{action})$, we add a new accepting trap state S_T, so that for every state in B_{action} is connected to it by the complementary set of action as shown by Figure 3.3.

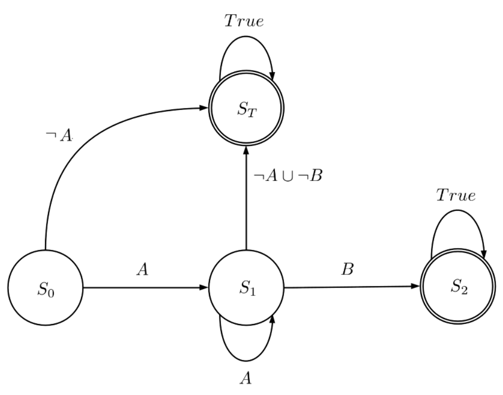


Figure 3.3: $Live(B_{action})$.

3.3.4 Liveness and Safety: Topological View

In [3], Alpern and Schneider introduce another interesting way to specify liveness and safety by the help of topology. In [7], a topological space is defined as following:

For a set X and element τ (a collection of subsets of X), we say that τ is a topology on X if and only if τ is closed under arbitrary union and finite intersection.

According to the definition in [7], the sets in τ are called the open sets and their complements in X are called the closed sets. A dense set in τ intersects every non-empty open set in τ.

In [15], Alpern and Schneider states that the set of finitely observable properties is a topology on S^{ω}. Finitely observable properties are like the "bad things" in safety properties and we should be able to decide whether they hold or not within a finite time. So, the sets of finitely observable properties become open sets. Intuitively, safety properties state "bad things" not happen, which are not finitely observable and it is the complements of the open sets. In another word, safety properties correspond to the closed sets. And any finite observation can be extended to be in a dense set and become a "good thing". So the dense sets correspond to liveness.

Every property P is the intersection of a safety and a liveness property. In [3], Alpern and Schneider give a proof as following:

Let $\overline{P}$ be the smallest safety property include P and L be $\neg(\overline{P} - P)$, then:

$$\begin{aligned} L \cap \overline{P} &= \neg(\overline{P} - P) \cap \overline{P} \\ &= (\neg\overline{P} \cup P) \cap \overline{P} \\ &= (\neg\overline{P} \cap \overline{P}) \cup (P \cap \overline{P}) \\ &= (P \cap \overline{P}) = P \end{aligned}$$

We can prove that L is dense (liveness) by contradiction. If L is not dense, there must be a non-empty open set O in $\neg L$. So O is in $\overline{P} - P$, then we can deduce that P is in $\overline{P} - O$, since the intersection of two closed sets is closed. So $\overline{P} - O$ is a safety property, which contradicts the fact that $\overline{P}$ is the smallest safety property include P.

For example, the security property stating that some actions of type B will eventually happen, but all actions in front of them should be of type A has a safety part: all actions in front of a type B action should have the type A. It also has a liveness part: actions of type B will eventually happen.

3.4 Execution Monitor

In [57], Schneider defined the properties class EM which can be enforced by an execution monitor that run along with a target program. When the target program wants to execute an action, the monitor checks it against its own security policy. If there is a violation, the target program will be terminated. Otherwise, the program continues.

According to [57], there are three major aspects that we should consider when formalizing the definition of the class EM. First, it should accept or reject an execution by analyzing it separately.

$$P(\Sigma) = \forall \sigma \in \Sigma . \hat{P}(\sigma) \qquad (1)$$

Second, the EM cannot make decisions based on the future actions, since this information is unavailable for it. So, once the monitor detects a violation of the security policy, it will not allow the program to continue, even thought there is a possibility that this violation can be remedied in the future. This is formalized by:

$$(\forall \tau \in \Sigma : \neg \hat{P}(\tau) \Rightarrow (\forall \sigma \in \Sigma : \neg \hat{P}(\tau\sigma))) \qquad (2)$$

where Σ is the set of possible executions and $\tau\sigma$ denotes the execution τ followed by the execution σ.

Third, any rejection of action by EM must be done after a finite period of time, which means:

$$(\forall \sigma \in \Sigma : \neg \hat{P}(\sigma) \Rightarrow (\exists i \geqslant 0 : \neg \hat{P}(\sigma[..i]))) \qquad (3)$$

where σ is a finite or infinite execution and $\sigma[..i]$ denote the prefix of σ including its first i steps.

According to the definition of safety property introduced above, security properties that satisfy (1) (2) and (3) are safety properties. In [57], Schneider notes that safety property can be used as an upper bound for the class of EM. We should notice that there are some safety properties that can not be enforced by execution monitors, this is due to the limited power of the EM monitors. For example, if the passage of time is responsible of the violating a security property, the monitor can not do anything about it, since it cannot control time. The power of execution monitoring can be extended so that it will be able to deal with a larger class of security property, as it will be discussed in the next section.

3.4.1 Security Automaton for EM

A property in the class EM can be specified by a security automaton. In [57], Schneider defined it as a deterministic automaton $I =< A, Q, Q_0, \delta >$ where:

- A is the set of alphabet of I.
- Q is a finite or countably infinite set of states.
- Q_0 is the set of initial states and it is the subset of Q.
- $\delta : Q \times A \rightarrow Q$ is a transition function.

We can see that a security automaton don't have accepting states, it can only reject the input when an undefined attempt (bad things) is detected. So, similar to the closure of Buchi automaton, it can only specify safety properties.

For the sake of further discussion, we specify the execution of a security automaton I on a sequence of program actions σ by labeled operational rules. Let σ, σ' and σ" be three sequences, q, q' and q" be three states, τ be a sequence of actions and ϵ be an empty sequence of action. As defined in [57], formal definition of single step semantics is as following:

$(\sigma, q) \xrightarrow{\tau} (\sigma', q')$:

if $\sigma = a\sigma'$ and $\delta(a, q) = q'$ then $\qquad (\sigma, q) \xrightarrow{a} (\sigma', q')$

otherwise $\qquad (\sigma, q) \xrightarrow{\epsilon} (\epsilon, q)$

3.5 More Powerful Monitors

In [41], authors discuss the power of different automata within uniform and non-uniform systems. For a software system S, let $\mathcal{A}$ be its set of all actions and Σ be its set of all possible traces. S is an uniform system if $\Sigma = \mathcal{A}^*$, while, S is an non-uniform system if $\Sigma \subset \mathcal{A}^*$.

Although, the security automata discussed above can only enforce safety properties, this is not the limit of all the execution monitors. In [41], Bauer, Ligatti and Walker point out that once the monitor can modify the program actions, it can enforces security properties beyond safety. The modification of actions includes suppress, insert and edit. Respectively, we introduce hereafter these three different types of automata.

3.5.1 Insertion Automaton

As defined in [41], insertion automata is a deterministic automaton $I =< A, Q, q_0, \delta, \gamma >$ where:

- A is the set of alphabets.
- Q is a finite or countably infinite set of states.
- q_0 is the initial state.
- $\delta : Q \times A \rightarrow Q$ is a transition function.
- γ is the function that can insert several actions into the program sequences.

In [41], it has been stated that insertion automata still can enforce only safety properties in uniform systems, while in non-uniform systems, their power can be extended to some liveness properties. This is due to the fact that we can access to the assistance of static analysis in non-uniform systems, which is forbidden in uniform one. For example, lets our liveness property states that: "action α happens eventually". We can mark the end of all given execution traces firstly. Then, during the execution of the program, if the monitor encounter an end mark and the action α is still not happened, it simply inserts the action α.

3.5.2 Suppression Automaton

As defined in [41], suppression automaton is a deterministic automaton $I =< A, Q, q_0, \delta, \omega >$ where:

- A is the set of alphabets.
- Q is a finite or countably infinite set of states.
- q_0 is the initial state.
- $\delta : Q \times A \rightarrow Q$ is a transition function.
- $\omega : A \times Q \rightarrow \{-, +\}$ is the partial function that indicate whether or not the action should be suppressed (-) or emitted (+).

In the uniform systems, the suppression automata still can enforce only safety properties. While in the non-uniform systems, their power are less than insertion automata. In [41], it has been stated that suppression automata cannot enforce bounded-availability policies and for every suppression automaton, we can build an insertion automaton that enforces the same property. But, there are some cases when suppression automata are more suitable to use. For example, if we want to change the original security policies, a suppression automaton can be more efficient.

3.5.3 Edit Automaton

Edit automata is simply the combination of the previous two automatons. Intuitively, edit automaton is the most powerful enforcement mechanism we discussed so far. But, when considering uniform system, it can enforce only safety properties.

For non-uniform systems, since an insertion automaton is more powerful than a suppression automaton, the edit automata enforces exactly the same set of properties as the insertion automata. Also, it builds an upper-bound for any other methods mentioned before. A taxonomy about the relationship between different security policies discussed in this section is shown in Figure 3.4 and Figure 3.5.

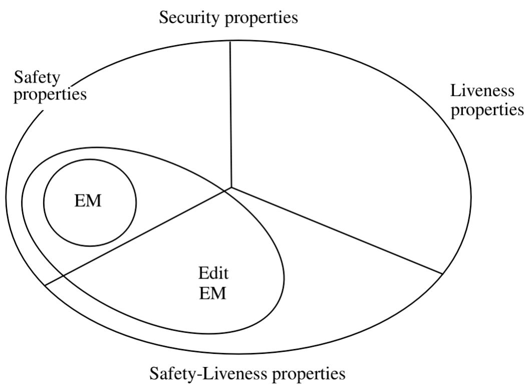


Figure 3.4: Enforcing Power of Edit Automata According to [41].

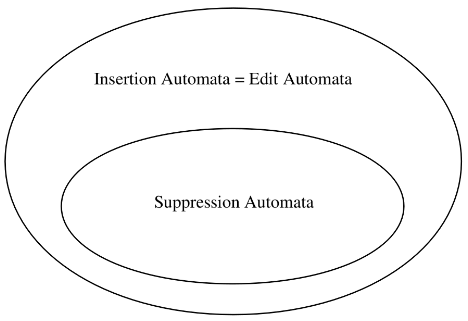


Figure 3.5: Enforcing Power of the three Automata for Non-Uniform Systems According to [41].

3.6 Further Discussion about Enforcing Abilities

In [32, 33], authors introduce a more elaborate framework to study the enforcement power of different monitors. To better understand this framework, it is important to clarify the meaning enforcement. In the remaining part of this section, we first give the formal definitions of three types of enforcement given in [33]. Then, we use these definitions to discuss the enforcement power of different automata.

3.6.1 Three Types of Enforcement

In [33] authors believe that enforcement mechanisms can accomplish their task effectively only when they respect the following two abstract principles.

- (Soundness) An enforcement mechanism must ensure that all observable outputs obey the property in question.
- (Transparency) An enforcement mechanism must preserve the semantics of executions that already respect the property in question.

The first criterion stipulates that all the bad executions of the program (which do not respect the security properties) should not be seen in the output. The second one requires that all the good executions (which already satisfy the security properties) should remain the same. If an enforcement mechanism can satisfy the first requirement (Soundness), but not necessarily the second, then this mechanism can conservatively enforce the property. The formal definition of Conservative Enforcement given in [33] is as following:

Conservative Enforcement: An automaton A with a starting state q_0 conservatively enforces a property $\hat{P}$ on a system with an action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^* \; \exists q', \sigma' \in \mathcal{A}^*$, such that:

1. $(\sigma, q_0) \overset{\sigma'}{\Longrightarrow}_A (., q')$ and
2. $\hat{P}(\sigma')$

Conservative Enforcement provides a great freedom to automata, for example, one automaton could just always output an empty stream of actions to enforce any property. This definition of enforcement can make an automaton seems powerful, but actually is not useful for end users. To solve this problem, we need to take the second criterion (Transparency) into consideration. In [33], authors give the formal definition of precise enforcement as following:

Precise Enforcement: An automaton A with a starting state q_0 precisely enforces a property $\hat{P}$ on a system with an action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^* \; \exists q', \sigma' \in \mathcal{A}^*$, such that:

1. $(\sigma, q_0) \overset{\sigma'}{\Longrightarrow}_A (., q')$,
2. $\hat{P}(\sigma')$ and
3. $\hat{P}(\sigma) \Rightarrow \forall i \; \exists q''. \; (\sigma, q_0) \overset{\sigma[..i]}{\Longrightarrow}_A (\sigma[i+1..], q'')$

In the definition of precise enforcement, the first two requirement stipulate that Precise Enforcement is included in Conservative Enforcement, the third requirement stipulates that if a stream of actions of the target program already satisfies the security property, the automaton should output it without any interruption or change.

The definition of Precise Enforcement does not consider the fact that two action streams can be semantically equivalent, but not necessarily syntactically the same. For example, if we want to write something on two files, which one we write first should not affect the final result. In [33], authors use an equivalence relation ($\cong$) to define semantically equivalent and require that $\cong$ is reflexive, symmetric and transitive. The formal definition of this equivalence is in [33] as following:

$$\sigma \cong \sigma' \Rightarrow \hat{P}(\sigma) \Leftrightarrow \hat{P}(\sigma')$$

The above definition stipulates that if one trace satisfies one property, all the equivalent ($\cong$) traces also satisfy it. Based on the above definition, authors introduce, in [33], the formal definition of Effective Enforcement as following:

Effective Enforcement: An automaton A with a starting state q_0 effectively enforces a property $\hat{P}$ on a system with an action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^* \ \exists q', \sigma' \in \mathcal{A}^*$.

1. $(\sigma, q_0) \overset{\sigma'}{\Longrightarrow}_A (., q')$,
2. $\hat{P}(\sigma')$ and
3. $\hat{P}(\sigma) \Rightarrow \sigma \cong \sigma'$

3.6.2 Enforcement Abilities of Different Automaton

In [32, 33], authors show four types of automata(Truncation Automata, Suppression Automata, Insertion Automata, Edit Automata) and discuss their enforcement abilities based on various levels of enforcement. Here, Truncation Automata are similar to the security automata introduced in Section 3.4.1. For the sake of simplicity, we use the name of automata to identify the set of properties that they can enforce, For example, editing properties means the set of properties that can be enforced by Edit Automata. Based on the result of [32, 33], Figure 3.5 is true only at the level of Precise Enforcement. A taxonomy about the updated result is shown in Figure 3.6 (page 47).

At the level of effective enforcement, a taxonomy about the relationship between different automata is shown in Figure 3.7 (page 47).

3.6.3 Enforcement Power of Edit Automata

In the definition of Effective Enforcement, if we change the "semantically equivalence" ($\cong$) by "syntactically equivalence" ($=$), we obtain a conservative version of Effective Enforcement. The formal definition of this level of enforcement is introduced in [32] as following:

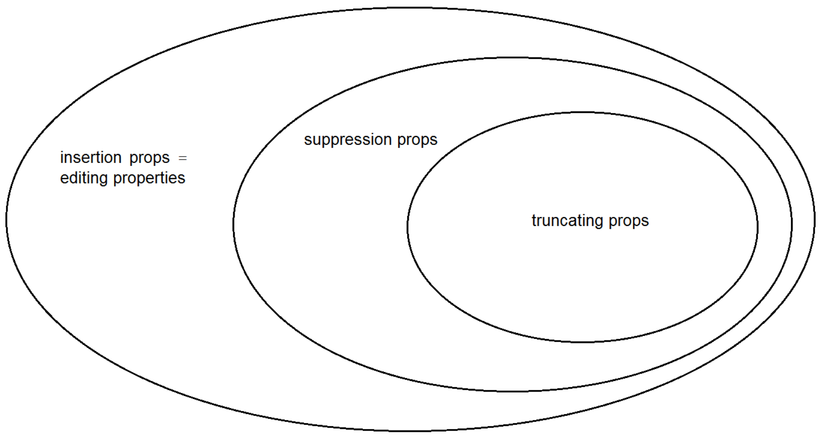


Figure 3.6: Enforcing Power of Different Automata for Precise Enforcement According to[32, 33].

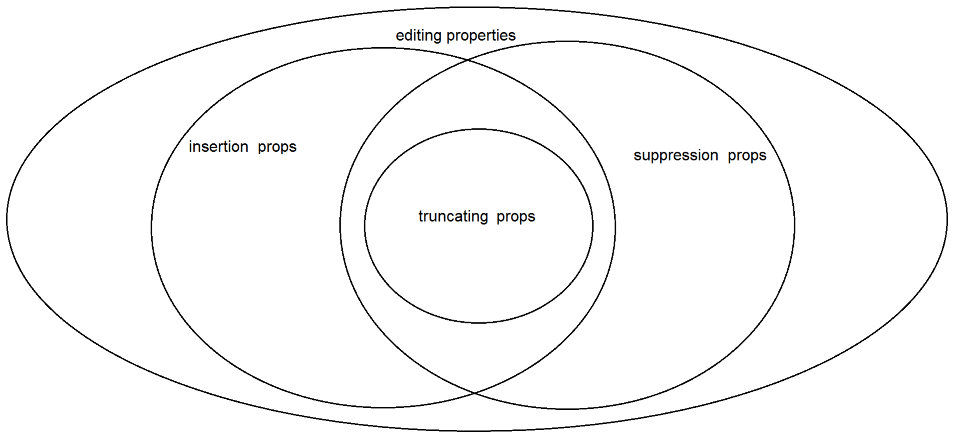


Figure 3.7: Enforcing Power of Different Automata for Effective Enforcement According to [32, 33].

Effective$_=$ Enforcement: An automaton A with a starting state q_0 effectively$_=$ enforces a property $\hat{P}$ on a system with an action set $\mathcal{A}$ if and only if $\forall \sigma \in \mathcal{A}^* \ \exists q', \sigma' \in \mathcal{A}^*$.

1. $(\sigma, q_0) \overset{\sigma'}{\Longrightarrow}_A (., q')$,
2. $\hat{P}(\sigma')$ and
3. $\hat{P}(\sigma) \Rightarrow \sigma = \sigma'$

In [32], the authors discuss the enforcing power of edit automata and concluded that an edit automaton can effectively$_=$ enforce any reasonable infinite renewal property. The formal definition of renewal property is given in [32] as following:

A property $\hat{P}$ is an infinite renewal property on a system with action set $\mathcal{A}$ if and only if:

$\forall \sigma \in \mathcal{A}^\omega : \hat{P}(\sigma) \Longleftrightarrow \{\sigma' \preceq \sigma | \hat{P}(\sigma')\} \ is \ an \ infinite \ set \quad (RENEWAL_1)$

In [32], the authors also give the condition $(RENEWAL_2)$ and use it to replace $(RENEWAL_1)$ to get an equivalent definition of renewal property.

$$\forall \sigma \in \mathcal{A}^{\omega} : \hat{P}(\sigma) \iff (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{P}(\tau)) \quad (RENEWAL_2)$$

The relationship between renewal properties, safety properties and liveness properties is shown by Figure 3.8.

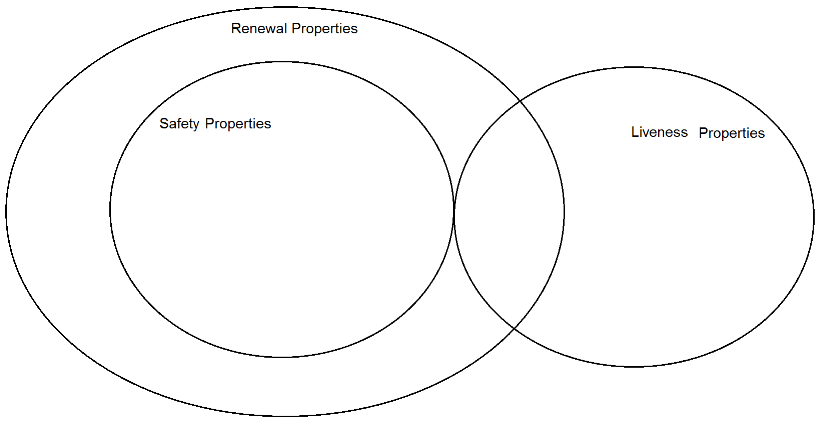


Figure 3.8: Relationship Between Renewal Properties, Safety Properties and Liveness Properties According to [32].

3.7 Hyperproperties

We have already stated that some security policies are not security properties, since they need to consider the relationship between different traces. Also, some interesting policies (non-interference, SLA) cannot be specified as properties. In [15], Clarkson and Schneider introduce Hyperproperties to deal with this problem.

3.7.1 Property and Hyperproperty

A property can be seen as a set of possibly infinite traces. A set T of traces satisfies a property P, if and only if all the traces of T are in P, i.e: ($T \subseteq P$).

Suppose for example that we have a boot area in the memory and the security property states that a writing action should not happen in this area. This property can be formalized by the following set of traces.

$$\text{NWB} = \{t \in S^{\omega} | \forall i \in \mathbb{N},\ \ noBootAreaWriting(t(i))\}$$

where S^{ω} denotes the set of all infinite traces, $\mathbb{N}$ denotes the set of natural numbers and $noBootAreaWriting(t(i))$ is a predicate on the ith action of the trace t.

A hyperproperty is defined in [15] as a set of sets of infinite traces, or equivalently a set of properties. Thus, a set T of traces satisfies a hyperproperty H, if and only if T is in H ($T \in P$). In mathematical logic, a security property corresponds to the first order logic and hyperproperty corresponds to the second order logic.

Noninterference, as defined by Goguen and Meseguer in [26], is an information flow policy and it states that commands caused by users at hight security levels should not have effects on the observation of users at low security levels. In [15], Clarkson and Schneider state that it is a hyperproperty and formalize it as following:

$$\begin{aligned}\text{GMNI} = \{T \in Prop \mid T \in GMSys \implies \\ (\forall t \in T : (\exists t' \in T : \\ ev_{Hin}(t') = \epsilon \wedge ev_L(t) = ev_L(t')))\}\end{aligned}$$

where $Prop$ is the set of all properties and $GMSys$ is the special requirement made in [26]. We use $ev(t)$ to denote the input (commands) and output (observations) events of an end user that take place during the trace t of the system, $ev_L(t)$ to denote the low level events within $ev(t)$ and $ev_{Hin}(t)$ to denote the high level command events within $ev(t)$.

Service Level Agreement (SLA), discussed in [15], specifies the acceptable performance of a system. For example: "the average response time of all executions should be less than 1." Again this is a hyperproperty and in [15], it is specified as following:

$$\text{RT} = \{T \in Prop \ | mean(\textstyle\bigcup_{t \in T} respTime(t)) < 1\}$$

3.7.2 Hypersafety

Just like safety and liveness for properties, there are hypersafety and hyperlivness for hyperproperties. Hypersafety in [15] is defined as following:

A hyperproperties S is a hypersafety if and only if:

$$\forall T \in Prop : T \notin S \implies (\exists M \in Obs : M \leq T$$

$$\wedge (\forall T' \in Prop : M \leq T' \implies T' \notin S))$$

where Obs is the set of all finite subsets of the sets of finite traces. If we have $T, T' \in Prop$, then we say that T is a prefix of T', denoted by $T \leq T'$, if and only if:

$$\forall t \in T : (\exists t' \in T' : t \leq t')$$

Clarkson and Schneider state that the GMNI is an example of hypersafety. The "bad thing" is a pair of traces (t, t'), where t' has no high input events and t and t' have the same low input events but different low output (observation) events.

k-safety hyperproperty

GMNI is an example where we need to consider two traces for "bad things". In [15], k-safety hyperproperty is a generalized definition in which the "bad thing" never involves more than k traces.

$$\forall T \in Prop : T \notin S \Longrightarrow (\exists M \in Obs : M \leq T$$
$$\wedge |M| \leq k \wedge (\forall T' \in Prop : M \leq T' \Longrightarrow T' \notin S))$$

KSHP (k) denotes the set of all k-safety hyperproperties. Intuitively, when $k = 1$, it contains all safety properties.

SS_k is an example of KSHP (k) stated in [15] as follows: Suppose that we have a secret file and we separate it into k shares, where each one can be displayed by an action. Then, a hyperproperty may stipulate that for all the executions of the system, all the k shares cannot all be revealed.

3.7.3 Hyperliveness

Hyperliveness is defined in [15] as following:

A hyperproperties L is a hyperliveness if and only if:

$$\forall T \in Obs : (\exists T' \in Prop : T \leq T' \wedge T' \in L)$$

RT is an example of hyperliveness. The "good thing" is the average response time should be lower than a finite threshold and we can always add a finite set of quick response traces to meet this requirement.

In [15], authors show that Generalized Noninterference (GNI) introduced by McLean in [46] is another example of hyperliveness properties. GNI requires that for any traces t_1 and t_2, a system must contain an interleaved trace t_3 whose high inputs are the same as t_1 and his low events are the same as t_2. It is modeled in [15] as following:

$$\text{GNI} = \{T \in Prop \ \ |\Big(\forall t_1, t_2 \in T : (\exists t_3 \in T : ev_{Hin}(t_3) = ev_{Hin}(t_1)$$
$$\wedge ev_L(t_3) = ev_L(t_2))\Big)\}$$

In [15], it states that if we define true = $Prop$, then it is both hypersafety and hyperliveness. On the other hand the minimal hyperproperty false = $\{\emptyset\}$ is a hypersafety, but not a hyperliveness.

3.7.4 Partition of Hyperproperties

Like properties, every hyperproperty is the intersection of a hypersafety and a hyperliveness. For example, considering a system containing secret files and providing service for users holding different security levels. Suppose that our policy specifies both confidential requirements specified by GMNI and response time requirement specified by RT, then this hyperproperty has a hypersafety and a hyperliveness parts.

The classification of hyperproperties is shown in Figure 3.9, where HP is the set of all hyperproperties, SHP is the set of all hypersafety and LHP is the set of all hyperliveness. NWB, GMNI, RT, $ture$ and $false$ are some examples that we have already introduced before.

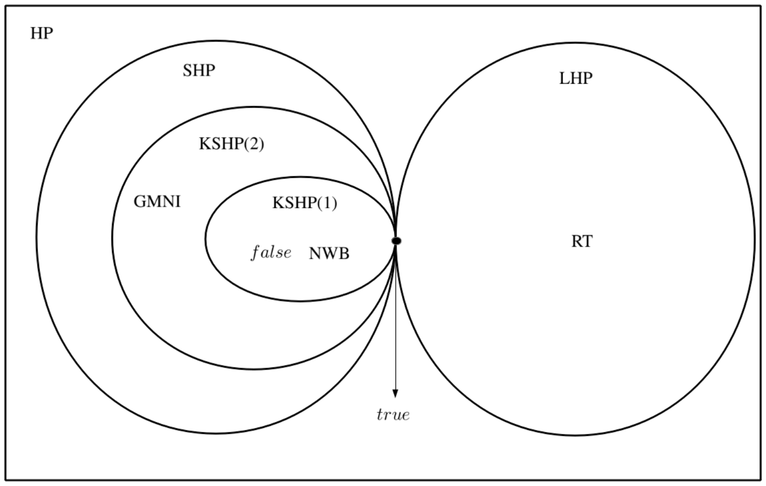


Figure 3.9: Classification of Hyperproperties.

3.8 Temporal Logics for Hyperproperties

Since a Hyperpropertie is a set of security properties, standard temporal logics (LTL, CTL and CTL^*) that can refer only to a single path each time are not suitable for specifying Hyperproperties and in [44], authors introduce two new logics ($HyperLTL$ and $HyperCTL^*$) to address this problem.

In [44], authors give a HyperLTL$_2$ prototype model checker based on algorithms for LTL model checking [25, 55, 65]. HyperLTL$_2$ is a simple version of HyperLTL. Authors also state that the worst-case running time is exponential but their purpose in building this prototype was a proof-of-concept for model checking of hyperproperties.

3.9 Conclusion

In this chapter, we have introduced the most relevant works related to the classification of security policies. We highlighted that security properties is a subset of security policies, since security policy need to consider different program traces, while security property is a predicate on one trace. Safety and liveness properties have also been introduced and we can extract them from any given property.

Also, we discussed the execution monitors, for the security property class EM, which are monitors that can only run along with the analysed program and enforce security properties by terminating it whenever it tries to violate the specified policy. EM is a subset of safety properties, but once it get the power to change the target programs, then some liveness properties can also be enforced.

Hyperporperties extend the definition of security properties, according to [15], they are powerful enough to specify all discovered security policies.

To sum up, all the results of this chapter are important for our work. Once a security policy was given for enforcement, we should know to which class it belongs and choose the appropriate enforcement method for it. For example, if a part of the security property is liveness, then the execution monitors without the ability to modify the target program should not be chosen. But what if we can choose different enforcement approaches for one given security property? In this case, we should understand well the advantages and disadvantages of different approaches.

Chapter 4

FASER (Formal and Automatic Security Enforcement by Rewriting) on BPA with Test

4.1 Introduction

Based on the work introduced in [48], we want to make many extensions so we can address a real programming language together with a rich variety of security policies.

To that end, we started by fixing the main foundation of the approach. Amongst authors, we changed the definition of the trace equivalence so that we obtain a congruence relation which is a helpful property. We have also proved all the result related to the correctness and the completeness of the approach, which was not available within the original work and we implemented a prototype showing the efficiency of the approach.

This chapter extends the expressiveness of the algebra $BPA^*_{0,1}$ by adding tests to allow the approach to enforce security policies on more systems. More specifically, first, we introduced a new algebra $CBPA^*_{0,1}$ which extend $BPA^*_{0,1}$. The untrusted code (program in C-like language for example) together with the security policy should be first specified as processes in $CBPA^*_{0,1}$. Secondly, we use our rewriting algorithm to get a kind of intersection between them as the result of the enforcement operation. Finally, we translate our result back to the original language. This whole enforcement process is as shown in Figure 4.1.

4.1.1 Ingredients and Steps to Solve the Problem

To solve the above problem, we need to provide the following ingredients:

- A formal language to specify systems.
- A formal language to specify security policies.

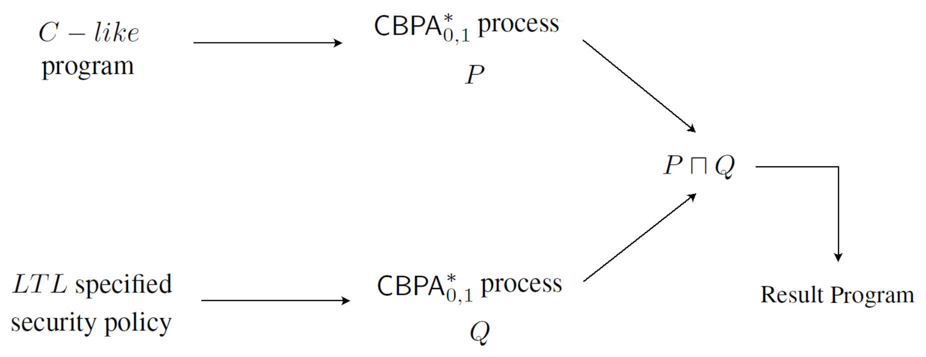

$C-like$ program
CBPA$^*_{0,1}$ process P
$P \sqcap Q$
LTL specified security policy
CBPA$^*_{0,1}$ process Q
Result Program

Figure 4.1: Security policy enforcement process with $CBPA^*_{0,1}$.

- Formalization of the problem: link between inputs and output of $\sqcap$.
- Resolution of the problem, i.e: find $P \sqcap Q$ shown in Figure 4.1.

In the remaining part of this chapter, we will address the above problems respectively.

4.2 Formal Language to Specify Systems: $CBPA^*_{0,1}$

To be able to specify more interesting programs and security policies, it is important to endow the algebra with conditional actions. In this section, we give the syntax and the semantics of $CBPA^*_{0,1}$, which is an extension of $BPA^*_{0,1}$ [9, 10, 48]. We also provide a more user-friendly interface to allow end-users to write their program in a $C-$ like language that can be translated into $CBPA^*_{0,1}$ by a given function.

4.2.1 Syntax of $CBPA^*_{0,1}$

Let $\mathcal{P}$ be the set of processes in $BPA^*_{0,1}$. Intuitively, $CBPA^*_{0,1}$ is $BPA^*_{0,1}$ endowed with an embedded boolean algebra $(\mathcal{B}, +, ., ^-, 0, 1)$, where $\mathcal{B} \subseteq \mathcal{P}$. More precisely, let a be an atomic action in $\mathcal{A}$ and c be a condition. The syntax of $CBPA^*_{0,1}$ is defined by the following BNF-grammar.

$$x, y ::= 0 \mid 1 \mid a \mid c \mid x + y \mid x.y \mid x^*y$$

Informally, the semantics of $CBPA^*_{0,1}$ is as following:

- 0 means that the process is in a deadlock state.
- 1 means that the process has finished normally its executions.
- c is either 0 or 1.
- $x + y$ is a choice between two processes x and y.

- $x.y$ is a sequential composition between x and y.
- x^*y is the process that behaves like $x.(x^*y)+y$. It is a binary version of the Kleene star operator [40].

To reduce the number of parenthesis in $CBPA^*_{0,1}$ terms, we use the following priority between operators (from high to low): "$*$", ".", "$+$". Notice also that "." is omitted when there is no ambiguity.

4.2.2 Semantics of $CBPA^*_{0,1}$

Suppose $[\![-]\!]_{\mathcal{B}}$ is an evaluation function from $\mathcal{B}$ to {0,1}, we introduce the notation $x\downarrow$ to know whether the process x can immediately terminate with success or not, by the inference rules given in Table 4.1.

Table 4.1: Definition of the Operator $\downarrow$.

$$(R^1)\frac{\square}{1\downarrow} \qquad (R^c)\frac{\square}{c\downarrow}[\![c]\!]_{\mathcal{B}}=1$$

$$(R^*_{r\downarrow})\frac{y\downarrow}{(x^*y)\downarrow} \qquad (R^{.}_{\downarrow})\frac{x\downarrow \quad y\downarrow}{(x.y)\downarrow}$$

$$(R^+_{l\downarrow})\frac{x\downarrow}{(x+y)\downarrow} \qquad (R^+_{r\downarrow})\frac{y\downarrow}{(x+y)\downarrow}$$

Now, the semantics of $CBPA^*_{0,1}$ can be defined by the transition relation $\rightarrow \in \mathcal{P}\times\Sigma\times\mathcal{P}$ given by Table 4.2.

Table 4.2: Operational Semantics of $BPA^*_{0,1}$.

$$(R^a)\frac{\square}{a\xrightarrow{a}1}$$

$$(R^{.}_l)\frac{x\downarrow \quad y\xrightarrow{a}y'}{x.y\xrightarrow{a}y'} \qquad (R^{.}_r)\frac{x\xrightarrow{a}x'}{x.y\xrightarrow{a}x'y}$$

$$(R^*_l)\frac{x\xrightarrow{a}x'}{x^*y\xrightarrow{a}x'.x^*y} \qquad (R^*_r)\frac{y\xrightarrow{a}y'}{x^*y\xrightarrow{a}y'}$$

4.2.3 Handling a $C-$Like Programming Language

$CBPA^*_{0,1}$ can be used to specify programs, but formal languages are difficult for many end-users. In order to provide a more user-friendly interface, we consider the following C-like programming language:

- `P::= ;| exit() | a| P;P′ | if (c) {P} else {P′} | while (c) do {P} | do {P} while (c)`

Any program written on this language can be translated to $CBPA^*_{0,1}$ by the following function:

$\lceil - \rceil$: C-Like $\longrightarrow \mathcal{E}BPA^*_{0,1}$		
$\lceil ; \rceil$	$=$	$1 \quad \lceil exit() \rceil = 0$
$\lceil P; P' \rceil$	$=$	$\lceil P \rceil . \lceil P' \rceil \quad \lceil a \rceil = a$
$\lceil if(c)\ \{P\}\ else\ \{P'\} \rceil$	$=$	$c.\lceil P \rceil + \overline{c}.\lceil P' \rceil$
$\lceil while(c)\ do\ \{P\} \rceil$	$=$	$(c.\lceil P \rceil)^* \overline{c}$
$do\ \{P\} \lceil while(c)\ \rceil$	$=$	$(\lceil P \rceil .c)^* \overline{c}$

4.3 A Formal Language to Specify Security Policies (LTL-like logic)

Suppose that a ranges over a finite set of actions $\mathcal{A}$, then the syntax of the logic is as following:

$$\Phi, \Phi_1, \Phi_2 \quad ::= \quad \top \mid \bot \mid 1 \mid a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid X\Phi \mid \Phi_1 U \Phi_2 \mid \Phi_1.\Phi_2$$

Let τ be a trace (a sequence of actions) and let τ^i be its suffix starting from the action at the i^{th} position. The semantics of a formula is given by $\models$ as follows:

- $\tau \models \top$, $\tau \not\models \bot$, $a \models a$, $\epsilon \models 1$,
- $\tau \models \neg\Phi$ if $\tau \not\models \Phi$,
- $\tau \models \Phi_1 \vee \Phi_2$ if $\tau \models \Phi_1$ or $\tau \models \wedge\Phi_2$
- $\tau \models \Phi_1 \wedge \Phi_2$ if $\tau \models \Phi_1$ and $\tau \models \wedge\Phi_2$
- $\tau \models X\Phi$ if $\tau^1 \models X\Phi$
- $\tau \models \Phi_1 U \Phi_2$ if there exists k such that $\tau^k \models \Phi_2$ and for all $0 \leq i < k$ $\tau^i \models \Phi_1$.
- $\tau \models \Phi_1.\Phi_2$ if there exists τ_1 and τ_2 such that $\tau = \tau_1.\tau_2$, $\tau_1 \models \Phi_1$ and $\tau_2 \models \Phi_2$.

Any formula Φ in the LTL-like logic can be translated to $BPA^*_{0,1}$ using following four steps:

1. We keep applying the following rules on Φ, until all "$\neg$" operators are in front of atomic actions.

$$\begin{aligned}
\neg\bot &\longrightarrow \top \\
\neg\top &\longrightarrow \bot \\
\neg\neg\Phi &\longrightarrow \Phi \\
\neg(X\Phi) &\longrightarrow X(\neg\Phi) \\
\neg(\Phi_1 \wedge \Phi_2) &\longrightarrow \neg\Phi_1 \vee \neg\Phi_2 \\
\neg(\Phi_1 \vee \Phi_2) &\longrightarrow \neg\Phi_1 \wedge \neg\Phi_2 \\
\neg(\Phi_1 U \Phi_2) &\longrightarrow (\neg\Phi_1) \vee (\Phi_1 U \neg\Phi_2) \\
\neg(\Phi_1 . \Phi_2) &\longrightarrow (\neg\Phi_1) \vee (\Phi_1 . \neg\Phi_2)
\end{aligned}$$

2. We keep applying the following rules until Φ becomes in a conjunctive normal form (CNF).

$$\begin{aligned}
X(\wedge_i \Phi_i) &\longrightarrow \wedge_i (X\Phi_i) \\
((\wedge_i \Phi_i) U (\wedge_j \Phi_j)) &\longrightarrow \wedge_i \wedge_j (\Phi_i U \Phi_j) \\
(\Phi_1 \wedge \Phi_2) \vee \Phi_3 &\longrightarrow (\Phi_1 \vee \Phi_3) \wedge (\Phi_2 \vee \Phi_3) \\
\Phi_1 \vee (\Phi_2 \wedge \Phi_3) &\longrightarrow (\Phi_1 \vee \Phi_1) \wedge (\Phi_1 \vee \Phi_3)
\end{aligned}$$

3. The previous steps transform any Φ to a form like $\wedge_{i=1}^{n} \Phi_i$, where each Φ_i does not contain the "$\wedge$" operator and all the "$\neg$" operators are in front of atomic actions. Now, we translate each Φ_i as a process in $BPA^*_{0,1}$ using the following function:

$\lceil - \rceil : LTL \longrightarrow BPA^*_{0,1}$		
$\lceil \bot \rceil$	$=$	0
$\lceil \top \rceil$	$=$	$(\sum_{a \in \mathcal{A}} a)^* 1$
$\lceil 1 \rceil$	$=$	1
$\lceil \neg 1 \rceil$	$=$	$(\sum_{a \in \mathcal{A}} a)^* (\sum_{a \in \mathcal{A}} a)$
$\lceil a \rceil$	$=$	a
$\lceil \neg a \rceil$	$=$	$\sum_{a_i \in \mathcal{A} - \{a\}} a_i$
$\lceil X\Phi \rceil$	$=$	$(\sum_{a \in \mathcal{A}} a) . \lceil \Phi \rceil$
$\lceil \Phi_1 \vee \Phi_2 \rceil$	$=$	$\lceil \Phi_1 \rceil + \lceil \Phi_2 \rceil$
$\lceil \Phi_1 U \Phi_2 \rceil$	$=$	$\lceil \Phi_1 \rceil^* \lceil \Phi_2 \rceil$

4. Finally, the enforcement of Φ on a process P becomes as following:

$$P \sqcap \Phi = P \sqcap \wedge_{i=1}^{n} \Phi_i = (P \sqcap \lceil \Phi_1 \rceil) ... \sqcap \lceil \Phi_n \rceil$$

Example 4.3.1 (Transfer LTL-like Language into $CBPA^*_{0,1}$) *Let r, w and s denote the actions $read$, $write$ and $send$ respectively. The security policy $\Phi = (\neg(r \vee w)) U ((r \vee w).(\neg s) U 1)$ in LTL-like language states that once the $read$ or the $write$ happens, then the $send$ action will be forbidden.*

First, we translate Φ into its conjunctive normal form:

$$\begin{aligned}\Phi &= (\neg r \wedge \neg w))U((r \vee w).(\neg s)U1)\\ &= (\neg r)U((r \vee w).(\neg s)U1) \wedge (\neg w)U((r \vee w).(\neg s)U1)\\ &= \Phi_1 \wedge \Phi_2\end{aligned}$$

Secondly, we translate Φ_1 and Φ_2 into $BPA^*_{0,1}$:

$$\begin{aligned}Q_1 = \lceil \Phi_1 \rceil &= \lceil (\neg r)U((r \vee w).(\neg s)U1) \rceil\\ &= (w+s)^*((r+w).(w+r)^*1)\\ Q_2 = \lceil \Phi_2 \rceil &= \lceil (\neg w)U((r \vee w).(\neg s)U1) \rceil\\ &= (r+s)^*((r+w).(w+r)^*1)\end{aligned}$$

Finally, the enforcement of Φ on a given program P will be obtained by resolving the following problem:

$$((P \sqcap Q_1) \sqcap Q_2)$$

4.4 Formalization of the Problem: Link Between Inputs and Output of $\sqcap$

In the sequel, we formalize the properties of an enforced process. To this end, we need the following definitions.

4.4.1 Trace Based Equivalence

To compare the behaviors of processes, we use a congruent relation based on traces as introduced in [24]. A sequence of actions τ is a trace of a process x if there exists another process x' such that $x \overset{\tau}{\twoheadrightarrow} x'$, where the relation $\twoheadrightarrow$ is defined as following:

Definition 4.4.1 (Definition of $\twoheadrightarrow$)

$$\frac{\square}{x \overset{\epsilon}{\twoheadrightarrow} x} \qquad \frac{x \overset{\tau}{\twoheadrightarrow} x' \quad x' \overset{a}{\rightarrow} x''}{x \overset{\tau.a}{\twoheadrightarrow} x''}$$

Also, we say that $x{\downarrow}\tau$, if there exists x', such that $x \overset{\tau}{\twoheadrightarrow} x'$.

Now we can use traces to compare two processes using the following ordering relation.

Definition 4.4.2 (Definition of $\sqsubseteq_T$) *Let x and y be two processes. We say that $x \sqsubseteq_T y$, if for $\forall \tau \in \Sigma^*$, we have:*

1. *if* $x{\downarrow}\tau$ *then* $y{\downarrow}\tau$
2. *if* $x \overset{\tau}{\twoheadrightarrow} 1$ *then* $y \overset{\tau}{\twoheadrightarrow} 1$.

$\sqsubseteq_T$ has some important properties.

Proposition 4.4.3 *$\sqsubseteq_T$ is a congruence relation.*

Proof: The proof is in Section 4.6, on page 68.

The transitivity of $\sqsubseteq_T$ is also an interesting property and it will be useful for the proof later.

Proposition 4.4.4 *$\sqsubseteq_T$ is a transitive relation.*

Proof: Directly from the definition of $\sqsubseteq_T$.

Now we can define the trace based equivalence $\sim$ form $\sqsubseteq_T$ above.

Definition 4.4.5 *($\sim$) We say that two processes x and y are trace equivalent and we write $x \sim y$, if $x \sqsubseteq_T y$ and $y \sqsubseteq_T x$.*

$\sim$ has the following important property.

Proposition 4.4.6 *Trace equivalent ($\sim$) is a congruence relation.*

Proof: Directly from the congruence of $\sqsubseteq_T$.

Hereafter, we give some other useful properties of $\sim$:

Proposition 4.4.7 *Given three processes x, y and z in $CBPA^*_{0,1}$, then the following properties hold.*

(B_1)	$x + (y + z)$	$\sim$	$(x + y) + z$	(B_7)	$0.x$	$\sim$	0
(B_2)	$x.(y.z)$	$\sim$	$(x.y).z$	(B_8)	$x.1$	$\sim$	$x \sim 1.x$
(B_3)	$x + y$	$\sim$	$y + x$	(B_9)	$x + 0$	$\sim$	x
(B_4)	$(x + y).z$	$\sim$	$x.z + y.z$	(B_{10})	x^*y	$\sim$	$y + xx^*y$
(B_5)	$x.(y + z)$	$\sim$	$x.y + x.z$	(B_{11})	$x + y \sqsubseteq_T z$	$\Rightarrow$	$x \sqsubseteq_T z, y \sqsubseteq_T z$
(B_6)	$x + x$	$\sim$	x	(B_{12})	$x.z + y \sqsubseteq_T z$	$\Rightarrow$	$x^*y \sqsubseteq_T z$

Proof:
B_1 to B_{11} are directly from the definition of $\sim$ and $\sqsubseteq_T$,
The proof of B_{12} is in Section 4.6.2, on page 74.

Now based on the definition of trace equivalence, we can discuss the relationship between inputs and output of our enforcement operator. Given a program P and a security policy Φ, the goal of this work is to generate another program P' that respects the security policy Φ and behaves like P except when the security policy is going to be violated. More precisely, we want that the result P' respects the two following properties:

1. **Correctness:**
   - $P' \sqsubseteq P$: all the traces of P' are traces in P.
   - $P' \sqsubseteq \Phi$: all the traces of P' respect the security policy.
2. **Completeness:** If there exists P" such that P" $\sqsubseteq P$ and P" $\sqsubseteq \Phi$, then P" $\sqsubseteq P'$. This property involves that all the traces in P that respect the security policy are also in P'.

The two previous properties can be stated using the notion of a greatest common factor or gcf defined as following:

Definition 4.4.8 (Greatest Common Factor (gcf)) *Let P and Q be two processes. The gcf of P and Q, denote by $P \sqcap Q$, is a process R that respects the following three conditions:*

- $R \sqsubseteq P$.
- $R \sqsubseteq Q$.
- For all R' such that $R' \sqsubseteq P$ and $R' \sqsubseteq Q$, we have $R' \sqsubseteq R$.

Now, the problem of enforcing a security property Φ on a program P turns to find $P \sqcap Q$. In the next section, we introduce an algorithm allowing to compute gcf, but before that, we give some useful properties of $\sqcap$.

Proposition 4.4.9 *let P, Q and R be three processes, the following properties hold:*

$$\begin{gathered} 1 \sqcap a \sim 0 \\ P \sqcap P \sim P \\ P \sqcap Q \sim Q \sqcap P \\ a.P \sqcap a.Q \sim a.(P \sqcap Q) \\ P \sqcap (Q + R) \sim P \sqcap Q + P \sqcap R \end{gathered}$$

Proof: directly from the definition of $\sqcap$.

4.5 Resolution of the Problem, Find $P \sqcap Q$ of Figure 4.1.

Hereafter, we give an algorithm, based on the notion of derivatives, allowing to compute the gcf of two processes P and Q.

4.5.1 Derivatives in $CBPA^*_{0,1}$

We adapt the definition of derivatives introduced by Brzozowski in [11] as following:

Definition 4.5.1 (Derivative of a process) *The derivative of a process x with respect to an action a, denote by $\partial_a(x)$, is defined as following:*

$$\partial : \Sigma \times \mathcal{P} \longrightarrow \mathcal{P}$$

$$\begin{array}{rcl}
\partial_a(0) & = & 0 \\
\partial_a(1) & = & 0 \\
\partial_a(c) & = & 0 \\
\partial_a(a) & = & 1 \\
\partial_a(b) & = & 0 \\
\partial_a(x^*y) & = & \partial_a(x).x^*y + \partial_a(y) \\
\partial_a(x+y) & = & \partial_a(x) + \partial_a(y) \\
\partial_a(x.y) & = & \partial_a(x).y + o(x).\partial_a(y)
\end{array}$$

Informally, the derivative of a process with respect to a given action is the remaining part of the process after the execution of this action, i.e:

$$\partial_a(x) = \sum_{\{x' \in \mathcal{P} \mid x \xrightarrow{a} x'\}} x'$$

Let ϵ denotes the empty trace, τ, τ_1 and τ_2 be trace in Σ^*, $\mathcal{T}$ be a set of traces in Σ^* and $\mathcal{P}$ be a set of processes. The notion of derivatives can be extended to a trace and a set of traces as following:

$$\begin{array}{rcl}
\partial_\epsilon(P) & = & P \\
\partial_{\tau_1.\tau_2}(P) & = & \partial_{\tau_1}(\partial_{\tau_2}(P)) \\
\partial_\tau(\mathcal{P}) & = & \bigcup_{P \in \mathcal{P}} \{\partial_\tau(P)\} \\
\partial_{\mathcal{T}}(P) & = & \bigcup_{\tau \in \mathcal{T}} \{\partial_\tau(P)\}
\end{array}$$

Some other definitions are also important for our algorithm.

Definition 4.5.2 (Immediate successful termination of a process) *The function $o(x)$ allows to know whether $x \downarrow$ holds or not and it is defined as follows:*

$o : \mathcal{P} \longrightarrow \{0, c, 1\}$		
$o(0)$	$=$	0
$o(1)$	$=$	1
$o(a)$	$=$	0
$o(c)$	$=$	c
$o(x^*y)$	$=$	$o(y)$
$o(x+y)$	$=$	$o(x) + o(y)$
$o(x.y)$	$=$	$o(x).o(y)$

Definition 4.5.3 (Immediate possible actions of a process) *The following function δ gives the immediate possible actions of a process:*

$\delta : \mathcal{P} \longrightarrow 2^{\Sigma}$		
$\delta(0)$	$=$	$\emptyset$
$\delta(1)$	$=$	$\emptyset$
$\delta(a)$	$=$	$\{a\}$
$\delta(c)$	$=$	$\emptyset$
$\delta(x^*y)$	$=$	$\delta(x) \cup \delta(y)$
$\delta(x+y)$	$=$	$\delta(x) \cup \delta(y)$
$\delta(x.y)$	$=$	$\delta(x) \cup o(x) \otimes \delta(y)$

Intuitively, $\delta(x) = \{a | x \overset{a}{\rightarrow} x'\}$ where $\otimes$ is defined as following:

$$o(x) \otimes \{\} = \{\}$$
$$o(x) \otimes (\{a\} \cup S) = \{o(x).a\} \cup o(x) \otimes S$$

To ensure that δ returns elements in 2^{Σ}, we need to consider that $\{1.a\} = \{a\}$ and $\{0.a\} = \{\}$

Definition 4.5.4 (Immediate possible conditions of a process) *The following function $\mathcal{C}$ gives the immediate possible conditions of a process:*

$$\mathcal{C}(P) := \{c | c.a \in \delta(P)\}$$

Intuitively, $\mathcal{C}(P)$ returns the condition part of each element in $\delta(P)$.

Definition 4.5.5 (Conditional process) *The conditional process P given c, denoted by P/c, is defined as following:*

$$
\begin{array}{rcl}
\hline\hline
\multicolumn{3}{c}{/ : \Sigma \times \mathcal{P} \longrightarrow \mathcal{P}} \\
\hline\hline
0/c & = & 0 \\
1/c & = & 0 \\
a/c & = & 0 \\
c_1/c_2 & = & 0, \text{ when } c_1 \neq c_2 \\
c/c & = & 1 \\
(x^*y)/c & = & (x/c).x^*y + (y/c) \\
(x+y)/c & = & (x/c) + (y/c) \\
(x.y)/c & = & (x/c).y + \kappa(x).(y/c) \\
\hline\hline
\end{array}
$$

where κ is defined as following:

$$
\begin{array}{rcl}
\hline\hline
\multicolumn{3}{c}{\kappa : \mathcal{P} \longrightarrow \{0,1\}} \\
\hline\hline
\kappa(0) & = & 0 \\
\kappa(1) & = & 1 \\
\kappa(a) & = & 0 \\
\kappa(c) & = & 0 \\
\kappa(x^*y) & = & \kappa(y) \\
\kappa(x+y) & = & \kappa(x) + \kappa(y) \\
\kappa(x.y) & = & \kappa(x).\kappa(y) \\
\hline\hline
\end{array}
$$

Notice that if the condition is $c = c_1c_2 \ldots c_n$, then $P/c = (P/c_1)/c_2 \ldots c_n$.

The definition of derivative allows us to present the relationship between the intersection of two processes and the intersection of their derivatives.

Proposition 4.5.6 *Let P be a process in $BPA^*_{0,1}$, then*

$$P \sim o(P) + \sum_{a \in \delta(P)} a.\partial_a(P)$$

Proof: The proof is in Section 4.6, on page 81. □

Proposition 4.5.7 *Let P and Q be two processes of $BPA^*_{0,1}$, then*

$$P \sqcap Q \sim o(P) \times o(Q) + \sum_{a \in \delta(P) \cap \delta(Q)} a.(\partial_a(P) \sqcap \partial_a(Q))$$

Proof: The proof is in Section 4.6, on page 86. □

For $CBPA^*_{0,1}$, we update above propositions as following:

Proposition 4.5.8 *Let P be a process in $CBPA^*_{0,1}$, then*

$$P \sim o(P) + \sum_{c \in \mathcal{C}(P)} c \sum_{a \in \delta(P/c)} a\, \partial_a(P/c)$$

Proof: The proof is similar to the proposition 4.5.6. □

Proposition 4.5.9 *Let P and Q be two processes of $BPA^*_{0,1}$, then*

$$P \sqcap Q = o(P).o(Q) + \sum_{\substack{(c_p, c_q) \in \\ \mathcal{C}(P) \times \mathcal{C}(Q)}} c_p\, c_q \sum_{\substack{a \in \delta(P/c_p) \cap \\ \delta(Q/c_q)}} a(\partial_a(P/c_p) \sqcap \partial_a(Q/c_q))$$

Proof: The proof is similar to the proposition 4.5.7. □

4.5.2 Algorithm

Based on Proposition 4.5.9, we write an algorithm allowing to generate a linear system where $P \sqcap Q$ could be extracted form its solution. This algorithm is as following:

Algorithm 1 calculate $P \sqcap Q$ in $BPA^*_{0,1}$

```
1: $E \longleftarrow \{P \sqcap Q = o(P).o(Q)+$
   $\sum_{\substack{(c_p, c_q) \in \\ \mathcal{C}(P) \times \mathcal{C}(Q)}} c_p\, c_q \sum_{\substack{a \in \delta(P/c_p) \\ \cap \delta(Q/c_q)}} a(\partial_a(P/c_p) \sqcap \partial_a(Q/c_q))\}$
2: while there exists $P_i \sqcap Q_i$ in the right side of any equation in $E$ that does
   not appear (modulo commutativity of $\sqcap$ and ACIT of $+$) in the left side
   on any equation do
   $E \longleftarrow E \cup \{P \sqcap Q = o(P).o(Q)+$
$\sum_{\substack{(c_p, c_q) \in \\ \mathcal{C}(P) \times \mathcal{C}(Q)}} c_p\, c_q \sum_{\substack{a \in \delta(P/c_p) \\ \cap \delta(Q/c_q)}} a(\partial_a(P/c_p) \sqcap \partial_a(Q/c_q))\}$
    end while
3: Return the solution of the linear system $E$.
```

- ACIT of $+$ is an abbreviation of Associativity ($(x+y)+z \sim x+(y+z)$), Commutativity ($x+y \sim y+x$), Identity ($x+x \sim x$) and Triviality ($x+0 \sim 0+x \sim x$, $0.x \sim 0$ and ($1.x \sim x.1 \sim x$).

- The algorithm terminates since the number of partial derivatives of regular terms x, denoted by $\mathcal{PD}(x)$ and defined as ∂_{Σ^*}(x), is finite as shown by Brzozowski [11].

- The complexity of the algorithm for two inputs P and Q is $O(||P|| \times ||Q||)$, where $||P||$ is the size of P, i.e. the number of elements in $\Sigma \cup \{0, 1\}$ in P. In fact, it was proved in [5] that the number of partial derivatives of P is smaller than $||P|| + 1$. Also, since any equation is the intersection of an element from $\mathcal{PD}(P)$ and another from $\mathcal{PD}(Q)$, then the number of equations N is no more than $(||P|| + 1) \times (||Q|| + 1)$ and their resolutions can be done by elimination using the Arden's Lemma [6] less than N times.

- The system generated by the algorithm is linear and has the following form $AX + B = X$ where A is a constant matrix of size $n \times n$, B is a constant vector of size n and X is a vector of variables of size n.

- The system can be solved iteratively by using Arden's Lemma [6]. Also, a generalized version of Arden Lemma shows that the solution of this system is $X = A^*B$, where A^* is computed as following:
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$
and the result can be inductively generalized for matrices n by n as following, where A is an $n-1$ by $n-1$ matrix:
$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^* = \left[\begin{array}{c|c} (A+BD^*C)^* & (A+BD^*C)^*BD^* \\ \hline (D+CA^*B)^*CA^* & (D+CA^*B)^* \end{array}\right]$$
These result holds because, under the trace equivalence, $CBPA^*_{0,1}$ is a kind of monodic tree Kleene algebra [62] with a binary version of the Kleene star.

4.5.3 Simple Example

Computing $P \sqcap Q$

- Inputs: $P = (c_1.a.b^*d)^*1$ and $Q = c_2.a.d^*1 + c_3.(a.d)^*1$

- Linear system generation:

$$P \sqcap Q = X_1$$

$=$ $\{\!|$Proposition 4.5.9 $|\!\}$

$$o(P).o(Q)+$$

$$\sum_{\substack{(c_p, c_q) \in \\ \mathcal{C}(P) \times \mathcal{C}(Q)}} c_p\ c_q \sum_{\substack{a \in \delta(P/c_p) \cap \\ \delta(Q/c_q)}} a(\partial_a(P/c_p) \sqcap \partial_a(Q/c_q))$$

$=$ $\{\!|o(P) = 1$ and $o(Q) = c_3$ and $\mathcal{C}(P) = \{c_1\}$ and $\mathcal{C}(Q) = \{c_2, c_3\}$ $|\!\}$

$$1.c_3 + c_1\ c_2 \sum_{a \in \delta(P/c_1) \cap \delta(Q/c_2)} a(\partial_a(P/c_1) \sqcap \partial_a(Q/c_2))$$
$$+c_1\ c_3 \sum_{a \in \delta(P/c_1) \cap \delta(Q/c_3)} a(\partial_a(P/c_1) \sqcap \partial_a(Q/c_3))$$

$=$ $\{\!|$Definition 5.5.1, Definition 4.5.5 $|\!\}$

$$c_3 + c_1\ c_2.(a.b^*d.P \sqcap a.d^*1) + c_1\ c_3.(a.b^*d.P \sqcap (a.d)^*1)$$

$=$

$$c_3 + c_1\ c_2.X_2 + c_1\ c_3.X_3$$

The rest of linear system can be generated by repeating the above processes.

$$\begin{aligned}
X_2 &= a.(b^*d.P \sqcap d^*1) = a.X_4 \\
X_4 &= d.(P \sqcap d^*1) = d.X_5 \\
X_5 &= c_1.0 = 0 \\
\\
X_3 &= a.(b^*d.P \sqcap d.(a.d)^*1) = a.X_6 \\
X_6 &= d.(P \sqcap (a.d)^*1) = d.X_7 \\
X_7 &= c_1.(a.b^*d.P \sqcap (a.d)^*1) = c_1.X_3
\end{aligned}$$

- Resolution of the system by using Arden's Lemma and variable elimination:

$$\begin{aligned}
X_2 &= a.d.0 \\
X_3 &= a.d.c_1.X_3 \\
X_3 &= (a.d.c_1)^*0 \\
X_1 &= c_3 + c_1.c_2.X_2 + c_1.c_3.X_3 \\
&= c_3 + c_1.c_2.a.d.0 + c_1.c_3.(a.d.c_1)^*0
\end{aligned}$$

- Conclusion:
  $P \sqcap Q = X_1 = c_3 + c_1.c_2.a.d.0 + c_1.c_3.(a.d.c_1)^*0.$

4.5.4 Example with Program

Let P be the following program:

```
while(true) {
   if(c) {
      Link(); Send(); Print();
   }
   else {
      Print(); Send();  Link();
   }
}
```

Let l, s and p denote the actions $Link$, $Send$ and $Print$ respectively. The program P can be specified in $BPA^*_{0,1}$ by the following process:

$$P = (c.l.s.p + \bar{c}.p.s.l)^*0$$

Suppose that we have a security policy stating that any $Send$ action needs to be preceded by a $Link$ one. An LTL property specifying this fact is $(\neg s)U(l.true)$ and its equivalent $CBPA^*_{0,1}$ process is $Q = (l+p)^*l.(l+p+s)^*1$. More details about transforming LTL formula to $CBPA^*_{0,1}$ are given in Section 4.3.

Now, we can compute $P \sqcap Q$ using Algorithm 1 as following:

$$\begin{array}{rcl}
P \sqcap Q = X_1 & = & c.l.(s.p.P \sqcap (Q + (l+s+p)^*1)) \\
 & & +\bar{c}.p.(s.l.P \sqcap Q) \\
 & = & c.l.X_2 + \bar{c}.p.X_3 \\
X_2 & = & s.(p.P \sqcap (l+s+p)^*1) \\
 & = & s.X_4 \\
X_3 & = & 0 \\
X_4 & = & p.(P \sqcap (l+s+p)^*1) \\
 & = & p.X_5 \\
X_5 & = & P
\end{array}$$

After substitution and simplification, we obtain:

$P \sqcap Q = c.l.s.p.(nc.p.s.l + c.l.s.p)^*0 + \overline{c}.p.0$

If we traduce the result to the C-like language, we obtain:

```
if(c) {
   Link(); Send(); Print();
   while(true) {
      if(c) {
         Link(); Send(); Print();
      }
      else {
         Print(); Send(); Link();
      }
   }
}
else {
   Print();
}
```

4.6 Proof of Main Result

4.6.1 Proof of Proposition 4.4.3

Proposition 4.4.3: $\sqsubseteq_T$ is a congruence relation.

Proof:

If $x \sqsubseteq_T x'$, then by Definition 4.4.2 ($\sqsubseteq_T$):

1. if $x \downarrow \tau$ then $x' \downarrow \tau$ (α)
2. if $x \overset{\tau}{\twoheadrightarrow} 1$ then $x' \overset{\tau}{\twoheadrightarrow} 1$ (β)

We need to prove that the $\sqsubseteq_T$ is congruent for the different operators of $CBPA^*_{0,1}$.

- $x + y \sqsubseteq_T x' + y$

if $(x + y) \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 4.2 $|\!\}$

$x \downarrow \tau$ or $y \downarrow \tau$

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$x' \downarrow \tau$ or $y \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 4.2. $|\!\}$

$(x'+y) \downarrow \tau$ (1)

if $x+y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 4.2 $|\!\}$

$x \overset{\tau}{\twoheadrightarrow} 1$ or $y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$x' \overset{\tau}{\twoheadrightarrow} 1$ or $y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 4.2. $|\!\}$

$x'+y \overset{\tau}{\twoheadrightarrow} 1$ (2)

From (1), (2) and Definition 4.4.2 ($\sqsubseteq_T$), we can say that $x+y \sqsubseteq_T x'+y$.

- $y+x \sqsubseteq_T y+x'$
  Same with above.
- $x.y \sqsubseteq_T x'.y$

if $x.y \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 4.2, Definition 4.4.1 and $\tau = \tau_1.\tau_2 |\!\}$

$x \downarrow \tau$ or $x \overset{\tau_1}{\twoheadrightarrow} 1, y \downarrow \tau_2$

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$x' \downarrow \tau$ or $x \overset{\tau_1}{\twoheadrightarrow} 1, y \downarrow \tau_2$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$x' \downarrow \tau$ or $x' \overset{\tau_1}{\twoheadrightarrow} 1, y \downarrow \tau_2$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 4.2 $|\!\}$

$x'.y \downarrow \tau$ or $x'.y \downarrow (\tau_1.\tau_2)$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2\ |\!\}$

$x'.y \downarrow \tau$ (1)

if $x.y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2, Definition 4.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$x \overset{\tau_1}{\twoheadrightarrow} 1, y \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)|\!\}$

$x' \overset{\tau_1}{\twoheadrightarrow} 1, y \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2 and $\tau = \tau_1.\tau_2|\!\}$

$x'.y \overset{\tau}{\twoheadrightarrow} 1$ (2)

From (1), (2) and Definition 4.4.2 ($\sqsubseteq_T$), we can say that $x.y \sqsubseteq_T x'.y$.

- $y.x \sqsubseteq_T y.x'$

if $y.x \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2, Definition 4.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$y \downarrow \tau$ or $y \overset{\tau_1}{\twoheadrightarrow} 1, x \downarrow \tau_2$

$\Rightarrow$ $\{\!|(\alpha)|\!\}$

$y \downarrow \tau$ or $y \overset{\tau_1}{\twoheadrightarrow} 1, x' \downarrow \tau_2$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2 $|\!\}$

$y.x' \downarrow \tau$ or $y.x' \downarrow (\tau_1.\tau_2)$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2|\!\}$

$y.x' \downarrow \tau$ (1)

if $y.x \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2, Definition 4.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$y \overset{\tau_1}{\twoheadrightarrow} 1, x \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)|\!\}$

$y \overset{\tau_1}{\twoheadrightarrow} 1, x' \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{\cdot}), (R_r^{\cdot})$ from Table 4.2 and $\tau = \tau_1.\tau_2|\!\}$

$y.x' \overset{\tau}{\twoheadrightarrow} 1$ (2)

From (1), (2) and Definition 4.4.2 ($\sqsubseteq_T$), we can say that $y.x \sqsubseteq_T y.x'$.

- $x^*y \sqsubseteq_T x'^*y$

if $x^*y \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

There are only four possibilities:
(a): $x \downarrow \tau$
(b): $y \downarrow \tau$
(c): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x \downarrow \tau''$
(d): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \downarrow \tau''$

For (a):
$x \downarrow \tau$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$x' \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^*)$ from Table 4.2 $|\!\}$
$x'^*y \downarrow \tau$

For (b):
$y \downarrow \tau$
$\Rightarrow$ $\{\!|(R_r^*)$ from Table 4.2 $|\!\}$
$x'^*y \downarrow \tau$

For (c):

$\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x \downarrow \tau''$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$x' \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x' \downarrow \tau''$

$\Rightarrow$ $\{\!|(R_l^*)$ from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau_1'...\tau_n'$ $|\!\}$

$x'^*y \downarrow \tau$
For (d):

$\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \downarrow \tau''$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$x' \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \downarrow \tau''$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau_1'...\tau_n'$ $|\!\}$

$x'^*y \downarrow \tau$

From the discussion of (a), (b), (c) and (d), we can conclude that:
if $x^*y \downarrow \tau$, then $x'^*y \downarrow \tau$ (1)

if $x^*y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

There are only two possibilities:
(a): $y \overset{\tau}{\twoheadrightarrow} 1$
(b): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, then $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \overset{\tau''}{\twoheadrightarrow} 1$

For (a):
$y \overset{\tau}{\twoheadrightarrow} 1$
$\Rightarrow$ $\{\!|(R_r^*)$ from Table 4.2 $|\!\}$
$x'^*y \overset{\tau}{\twoheadrightarrow} 1$
For (b):
$\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, then $x \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \overset{\tau''}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$x' \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \overset{\tau''}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau_1'...\tau_n'$ $|\!\}$

$x'^*y \overset{\tau}{\twoheadrightarrow} 1$

From the discussion of (a) and (b), we can conclude that:
if $x^*y \overset{\tau}{\twoheadrightarrow} 1$, then $x'^*y \overset{\tau}{\twoheadrightarrow} 1$ (2)

From (1), (2) and Definition 4.4.2 ($\sqsubseteq_T$), we can say that $x^*y \sqsubseteq_T x'^*y$.

- $y^*x \sqsubseteq_T y^*x'$

if $y^*x \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

There are only four possibilities:
(a): $x \downarrow \tau$
(b): $y \downarrow \tau$
(c): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \downarrow \tau''$
(d): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x \downarrow \tau''$

For (a):
$x \downarrow \tau$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$x' \downarrow \tau$

$\Rightarrow$ $\{\!|\, (R_r^*)$ from Table 4.2 $|\!\}$

$y^* x' \downarrow \tau$

For (b):

$y \downarrow \tau$

$\Rightarrow$ $\{\!|\, (R_l^*)$ from Table 4.2 $|\!\}$

$y^* x' \downarrow \tau$

For (c):

$\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $y \downarrow \tau''$

$\Rightarrow$ $\{\!|\, (R_l^*)$ from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau_1'...\tau_n'$ $|\!\}$

$y^* x' \downarrow \tau$

For (d):

$\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, $y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x \downarrow \tau''$

$\Rightarrow$ $\{\!|\, (\alpha)\ (\beta) |\!\}$

$y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x' \downarrow \tau''$

$\Rightarrow$ $\{\!|\, (R_l^*)$, (R_r^*) from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau_1'...\tau_n'$ $|\!\}$

$y^* x' \downarrow \tau$

From the discussion of (a), (b), (c) and (d), we can conclude that:

if $y^* x \downarrow \tau$, then $y^* x' \downarrow \tau$ $\quad (1)$

if $y^* x \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|\, (R_l^*)$, (R_r^*) from Table 4.2 and Definition 4.4.1 $|\!\}$

There are only two possibilities:

(a): $x \overset{\tau}{\twoheadrightarrow} 1$

(b): $\tau = \tau'.\tau''$, $\tau' = \tau_1'...\tau_n'$, for all i, $1 \leq i \leq n$, then $y \overset{\tau_i'}{\twoheadrightarrow} 1$ and $x \overset{\tau''}{\twoheadrightarrow} 1$

For (a):

$x \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|\, (\alpha)\ (\beta) |\!\}$

$x' \overset{\tau_i'}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|\, (R_r^*)$ from Table 4.2 $|\!\}$

$y^* x' \overset{\tau}{\twoheadrightarrow} 1$

For (b):

$\tau = \tau'.\tau''$, $\tau' = \tau'_1 ... \tau'_n$, for all i, $1 \leq i \leq n$, then $y \overset{\tau'_i}{\twoheadrightarrow} 1$ and $x \overset{\tau''}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\alpha)\ (\beta)|\!\}$

$y \overset{\tau'_i}{\twoheadrightarrow} 1$ and $x' \overset{\tau''}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R^*_l)$, (R^*_r) from Table 4.2, Definition 4.4.1,
$\tau = \tau'.\tau''$ and $\tau' = \tau'_1 ... \tau'_n$ $|\!\}$

$y^* x' \overset{\tau}{\twoheadrightarrow} 1$

From the discussion of (a) and (b), we can conclude that:
if $y^* x \overset{\tau}{\twoheadrightarrow} 1$, then $y^* x' \overset{\tau}{\twoheadrightarrow} 1$ (2)

From (1), (2) and Definition 4.4.2 ($\sqsubseteq_T$), we can say that $y^* x \sqsubseteq_T y^* x'$.

- Conclusion: $\sqsubseteq_T$ is a congruence relation.

□

4.6.2 Proof of Proposition 4.4.7 B_{12}

Lemma 4.6.1 *Let x and y be two processes, τ,τ_1... τ_n be $n+1$ traces, such that $\tau = \tau_1 ... \tau_n$ and for all τ_i, $1 \leq i \leq n$, we have $x \overset{\tau_i}{\twoheadrightarrow} 1$. If $x.y \sqsubseteq_T y$, then: there exists y_n such that $y \overset{\tau}{\twoheadrightarrow} y_n$ and $y \sqsubseteq_T y_n$*

Proof:

The proof is done by induction on "n".

- When n = 1: we have $\tau = \tau_1$, $x \overset{\tau_1}{\twoheadrightarrow} 1$. $x.y \sqsubseteq_T y$, now we need to prove that there exists y_1 such that $y \overset{\tau_1}{\twoheadrightarrow} y_1$, and $y \sqsubseteq_T y_1$.

$x \overset{\tau_1}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R^*_l)$, (R^*_r) from Table 4.2 and Definition 4.4.1 $|\!\}$
$x.y \overset{\tau_1}{\twoheadrightarrow} y$

$\Rightarrow$ $\{\!|x.y \sqsubseteq_T y\ |\!\}$

There exists y_1 such that $y \overset{\tau_1}{\twoheadrightarrow} y_1$ (α)

Now for $y \sqsubseteq_T y_1$, assume this is false, then from Definition 4.4.2, either **(1)** or **(2)** below must be true:

(1) $\exists \tau_y$, such that $y \downarrow \tau_y$, and we can not find y'_1 such that $y_1 \overset{\tau_y}{\twoheadrightarrow} y'_1$

$\Rightarrow$ $\{\!|$ $(R_l^;)$, $(R_r^;)$ from Table 4.2, (α) and $x \overset{\tau_1}{\twoheadrightarrow} 1$ $|\!\}$

$x.y \downarrow (\tau_1.\tau_y)$, and we can not find y_1' such that $y \overset{\tau_1}{\twoheadrightarrow} y_1 \overset{\tau_y}{\twoheadrightarrow} y_1'$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \downarrow (\tau_1.\tau_y)$, and we can not find y_1' such that $y \overset{\tau_1.\tau_y}{\twoheadrightarrow} y_1'$. This contradict the fact that $x.y \sqsubseteq_T y$

(2) $\exists \tau_y$, such that $y \overset{\tau_y}{\twoheadrightarrow} 1$, but $y_1 \overset{\tau_y}{\twoheadrightarrow} 1$ is not true

$\Rightarrow$ $\{\!|$ $(R_l^;)$, $(R_r^;)$ from Table 4.2, (α) and $x \overset{\tau_1}{\twoheadrightarrow} 1$ $|\!\}$

$x.y \overset{\tau_1}{\twoheadrightarrow} y \overset{\tau_y}{\twoheadrightarrow} 1$, but $y \overset{\tau_1}{\twoheadrightarrow} y_1 \overset{\tau_y}{\twoheadrightarrow} 1$ is not true

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_1.\tau_y}{\twoheadrightarrow} 1$, but $y \overset{\tau_1.\tau_y}{\twoheadrightarrow} 1$ is not true. This also contradict the fact that $x.y \sqsubseteq_T y$.

So $y \sqsubseteq_T y_1$, since we have already proved (α) then when n = 1, Lemma is true.

- Assume that when n = k, Lemma is true, which means that for $k+1$ traces Let τ,τ_1... τ_k, such that $\tau = \tau_1 ... \tau_k$, for all τ_i, $1 \le i \le k$ $x \overset{\tau_i}{\twoheadrightarrow} 1$, if $x.y \sqsubseteq_T y$ then there exists y_k such that $y \overset{\tau}{\twoheadrightarrow} y_k$ and $y \sqsubseteq_T y_{k+1}$

  Now we need to prove that when $n = k+1$, Lemma is also true, so we have $k+1$ traces $\tau_1 ... \tau_{k+1}$ such that for all $1 \le i \le k+1$: $x \overset{\tau_i}{\twoheadrightarrow} 1$ and $x.y \sqsubseteq_T y$.

$x \overset{\tau_{k+1}}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|$ $(R_l^;)$, $(R_r^;)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_{k+1}}{\twoheadrightarrow} y$

$\Rightarrow$ $\{\!|$ $x.y \sqsubseteq_T y$ $|\!\}$

$y \downarrow \tau_{k+1}$

$\Rightarrow$ $\{\!|$ $y \sqsubseteq_T y_k$ $|\!\}$

There exists y_{k+1} such that $y_k \overset{\tau_{k+1}}{\twoheadrightarrow} y_{k+1}$

$\Rightarrow$ $\{\!|$ $y \overset{\tau_1 ... \tau_k}{\twoheadrightarrow} y_k$ and Definition 4.4.1 $|\!\}$

There exists y_{k+1} such that $y \overset{\tau_1 ... \tau_{k+1}}{\twoheadrightarrow} y_{k+1}$ (β)

Now we want to prove that $y \sqsubseteq_T y_{k+1}$. Assume this is false, then from Definition 4.4.2 either **(1)** or **(2)** below must be true:

(1) $\exists \tau_y$, such that $y \downarrow \tau_y$, and we can not find y'_{k+1} such that $y_{k+1} \overset{\tau_y}{\twoheadrightarrow} y'_{k+1}$

$x \overset{\tau_1}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 4.2 $|\!\}$

$x.y \overset{\tau_1}{\twoheadrightarrow} y$

$\Rightarrow$ $\{\!|$When $n = k$, Lemma is true and $(k+1) - 2 + 1 = k$ $|\!\}$

$x.y \overset{\tau_1}{\twoheadrightarrow} y$, $\exists y''_{k+1}\ y \overset{\tau_2 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y \sqsubseteq_T y''_{k+1}$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y \sqsubseteq_T y''_{k+1}$

$\Rightarrow$ $\{\!|y \downarrow \tau_y$ and Definition 4.4.2 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y''_{k+1} \downarrow \tau_y$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \downarrow (\tau_1 \ldots \tau_{k+1}.\tau_y)$

$\Rightarrow$ $\{\!|$**(1)** $|\!\}$

$x.y \downarrow (\tau_1 \ldots \tau_{k+1}.\tau_y)$, and we can not find y'_{k+1} such that $y_{k+1} \overset{\tau_y}{\twoheadrightarrow} y'_{k+1}$

$\Rightarrow$ $\{\!|\beta$ and Definition 4.4.1 $|\!\}$

$x.y \downarrow (\tau_1 \ldots \tau_{k+1}.\tau_y)$, and we can not find y'_{k+1} such that $y \overset{\tau_1 \ldots \tau_{k+1}.\tau_y}{\twoheadrightarrow} y'_{k+1}$.

This contradict the fact that $x.y \sqsubseteq_T y$.

(2) $\exists \tau_y$, such that $y \overset{\tau_y}{\twoheadrightarrow} 1$, and $y_{k+1} \overset{\tau_y}{\twoheadrightarrow} 1$ is not true.

$x \overset{\tau_1}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 4.2 $|\!\}$

$x.y \overset{\tau_1}{\twoheadrightarrow} y$

$\Rightarrow$ $\{\!|$When $n = k$, Lemma is true and $(k+1) - 2 + 1 = k$ $|\!\}$

$x.y \overset{\tau_1}{\twoheadrightarrow} y$, $\exists y''_{k+1}\ y \overset{\tau_2 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y \sqsubseteq_T y''_{k+1}$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y \sqsubseteq_T y''_{k+1}$

$\Rightarrow$ $\{\!|\, y \overset{\tau_y}{\twoheadrightarrow} 1$ and Definition 4.4.2 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}}{\twoheadrightarrow} y''_{k+1}$ and $y''_{k+1} \overset{\tau_y}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}.\tau_y}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|$**(2)** $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}.\tau_y}{\twoheadrightarrow} 1$, and $y_{k+1} \overset{\tau_y}{\twoheadrightarrow} 1$ is not true.

$\Rightarrow$ $\{\!|\, y \overset{\tau_1 \ldots \tau_{k+1}}{\twoheadrightarrow} y_{k+1}$ and Definition 4.4.1 $|\!\}$

$x.y \overset{\tau_1 \ldots \tau_{k+1}.\tau_y}{\twoheadrightarrow} 1$, and $y \overset{\tau_1 \ldots \tau_{k+1}.\tau_y}{\twoheadrightarrow} 1$ is not true.

This is also contradict the fact that $x.y \sqsubseteq_T y$.

So $y \sqsubseteq_T y_{k+1}$, since we have already proved (β), then when n = k + 1, Lemma still is true.

-To sum up, for all the nature number n, Lemma is true.

□

Lemma 4.6.2 *Let x and y be two processes in $BPA^*_{0,1}$, then*

$$x.y \sqsubseteq_T y \Rightarrow x^*y \sqsubseteq_T y$$

Proof: The proof is directly from the definition of $\sqsubseteq_T$.

$x^*y \downarrow \tau$

$\Rightarrow$ $\{\!|(R^*_l), (R^*_r)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

There must exist τ'', such that: $x \downarrow \tau''$ or $y \downarrow \tau''$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1 \ldots \tau'_n$, $1 \leq i \leq n$, $x \overset{\tau'_i}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R^{;}_l), (R^{;}_r)$ from Table 4.2 $|\!\}$

There must exist τ'', such that: $x.y \downarrow \tau''$ or $y \downarrow \tau''$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1 \ldots \tau'_n$, $1 \leq i \leq n$, $x \overset{\tau'_i}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|\, x.y \sqsubseteq_T y \,|\!\}$

There must exist τ'', such that: $y \downarrow \tau''$ or $y \downarrow \tau''$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1 \ldots \tau'_n$, $1 \leq i \leq n$, $x \overset{\tau'_i}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|x.y \sqsubseteq_T y$ and Lemma 4.6.1 $|\!\}$

There must exist τ'', such that: $y \downarrow \tau''$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1...\tau'_n$, $1 \le i \le n$, $\exists y_n$ such that $y \overset{\tau'}{\twoheadrightarrow} y_n$ $y \sqsubseteq_T y_n$

$\Rightarrow$ $\{\!|$Definition 4.4.2 $|\!\}$

$y \overset{\tau'}{\twoheadrightarrow} y_n$ and $y_n \downarrow \tau''$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$y \downarrow \tau$ $\qquad\qquad (1)$

if $x^*y \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^*), (R_r^*)$ from Table 4.2 and Definition 4.4.1 $|\!\}$

There must exist τ'', such that: $y \overset{\tau''}{\twoheadrightarrow} 1$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1...\tau'_n$, $1 \le i \le n$, $x \overset{\tau'_i}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|x.y \sqsubseteq_T y$ and Lemma 4.6.1 $|\!\}$

There must exist τ'', such that: $y \overset{\tau''}{\twoheadrightarrow} 1$. Here $\tau = \tau'.\tau''$,
$\tau' = \tau'_1...\tau'_n$, $1 \le i \le n$, $\exists y_n$ such that $y \overset{\tau'}{\twoheadrightarrow} y_n$ $y \sqsubseteq_T y_n$

$\Rightarrow$ $\{\!|$Definition 4.4.2 $|\!\}$

$y \overset{\tau'}{\twoheadrightarrow} y_n$ and $y_n \overset{\tau''}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|$Definition 4.4.1 $|\!\}$

$y \overset{\tau}{\twoheadrightarrow} 1$ $\qquad\qquad (2)$

- From definition 4.4.1, **(1)** and **(2)**, Lemma is true.

□

Proposition 4.4.7 B_{12}: $x.z + y \sqsubseteq_T z \Rightarrow x^*y \sqsubseteq_T z$

Proof:

Now we need to prove that Lemma 4.6.2 is equivalence to Proposition 4.4.7 B_{12}, which is:

$$x.y \sqsubseteq_T y \Rightarrow x^*y \sqsubseteq_T y \qquad \Longleftrightarrow \qquad x.z + y \sqsubseteq_T z \Rightarrow x^*y \sqsubseteq_T z$$

- $\Longrightarrow$

$x.y \sqsubseteq_T y \Rightarrow x^*y \sqsubseteq_T y$

$\Rightarrow$ {|Let z replace y. |}

$x.z \sqsubseteq_T z \Rightarrow x^*z \sqsubseteq_T z$

$\Rightarrow$ {|$x.z + y \sqsubseteq_T z$ and Proposition 4.4.7 B_{11} |}

$x.z + y \sqsubseteq_T z \Rightarrow y \sqsubseteq_T z$

$\Rightarrow$ {|$\sqsubseteq_T$ is a congruence relationship |}

$x.z + y \sqsubseteq_T z \Rightarrow x^*y \sqsubseteq_T x^*z$

$\Rightarrow$ {|$x^*z \sqsubseteq_T z$ and $\sqsubseteq_T$ is a transitive relation |}

$x.z + y \sqsubseteq_T z \Rightarrow x^*y \sqsubseteq_T z$

- $\Longleftarrow$

$x.z + y \sqsubseteq_T z \Rightarrow x^*y \sqsubseteq_T z$

$\Rightarrow$ {|Let y replace z. |}

$x.y + y \sqsubseteq_T y \Rightarrow x^*y \sqsubseteq_T y$

$\Rightarrow$ {|Proposition 4.4.7 B_{11} |}

$x.y \sqsubseteq_T y \Rightarrow x^*y \sqsubseteq_T y$

- To sum up Proposition 4.4.7 B_{12} on page 59 is true.

□

4.6.3 Proof of Proposition 4.5.6

Lemma 4.6.3 *Let x and y be two processes in $BPA^*_{0,1}$, then $y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y \sim x^*y$, when $x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$ and o(x) = 1.*

Proof:

x^*y

$\sim$ {|$x^*y \sim y + x.x^*y$ |}

$y + x.x^*y$

$\sim$ {|$x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$ *and* o(x) = 1 and "$\sim$" is congruence. |}

$$y + (1 + \sum_{a\in\delta(x)} a.\partial_a(x)).x^*y$$

$\sim$ $\{\!|(x+y).z \sim x.z + y.z$ and $x + y \sim y + x\ |\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y + x^*y$$

Let $\beta = y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$, from above we can have $x^*y \sim \beta + x^*y \Rightarrow \beta \sqsubseteq_T x^*y$,
Now we need to prove $x^*y \sqsubseteq_T \beta$ to get $x^*y \sim \beta$.

$$\beta = y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$$

$\sim$ $\{\!|x^*y \sim y + x.x^*y\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).(y + x.x^*y)$$

$\sim$ $\{\!|x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)\ \ \ and\ \ \ o(x) = 1\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).(y + (1 + \sum_{a\in\delta(x)} a.\partial_a(x)).x^*y)$$

$\sim$ $\{\!|(x+y).z \sim x.z + y.z\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).(y + x^*y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y)$$

$\sim$ $\{\!|x.(y+z) \sim x.y + x.z\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y + \sum_{a\in\delta(x)} a.\partial_a(x). \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$$

$\sim$ $\{\!|x \sim x + x\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + y + y + \sum_{a\in\delta(x)} a.\partial_a(x).y +$$

$$\sum_{a\in\delta(x)} a.\partial_a(x).x^*y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y + \sum_{a\in\delta(x)} a.\partial_a(x). \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$$

$\sim$ $\{\!|x + y \sim y + x\ |\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y +$$

$$y + y + \sum_{a\in\delta(x)} a.\partial_a(x).y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y + \sum_{a\in\delta(x)} a.\partial_a(x). \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$$

$\sim$ $\{\!|x.z + y.z \sim (x+y).z\ \ and\ \ " \sim "$ is congruence. $|\!\}$

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y +$$

$y + (1 + \sum_{a\in\delta(x)} a.\partial_a(x)).y + (1 + \sum_{a\in\delta(x)} a.\partial_a(x)). \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$

$\sim$ $\{\!|y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y = \beta \;\; and \;\; o(x) = 1|\!\}$

$\beta + y + (o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)).y + (o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)). \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$

$\sim$ $\{\!|x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x) \;\; and \;\; " \sim "$ is congruence. $|\!\}$

$\beta + y + x.y + x. \sum_{a\in\delta(x)} a.\partial_a(x).x^*y$

$\sim$ $\{\!|x.y + x.z \sim x.(y + z) \;\; and \;\; " \sim "$ is congruence. $|\!\}$

$\beta + y + x.(y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y)$

$\sim$ $\{\!|y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y = \beta \; |\!\}$

$\beta + y + x.\beta$

Now we have:

$\beta \sim \beta + y + x.\beta$

$\sim$ $\{\!|x \sim y + x \Rightarrow y \sqsubseteq_T x \; |\!\}$

$y + x.\beta \sqsubseteq_T \beta$

$\sim$ $\{\!|B_{12} : b + a.x \sqsubseteq_T x \Rightarrow a^*b \sqsubseteq_T x \; |\!\}$

$x^*y \sqsubseteq_T \beta$

By the definition of trace equivalence, we conclude that $y + \sum_{a\in\delta(x)} a.\partial_a(x).x^*y \sim x^*y$, when $x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$ and o(x) = 1.

□

Proposition 4.5.6 Let x be a process in $BPA^*_{0,1}$, then

$$x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$$

Proof:

From the syntax of $BPA^*_{0,1}$:

$$x, y ::= 0 \mid 1 \mid a \mid x + y \mid x.y \mid x^*y$$

The proof is by structural induction.

- $P = 0$

$o(0) + \sum_{a \in \delta(0)} a.\partial_a(0)$

= $\{\!|o(0) = 0 \;\; and \;\; \delta(0) = \emptyset|\!\}$

0

- $P = 1$

o(1) + $\sum_{a \in \delta(1)} a.\partial_a(1)$

= $\{\!|o(1) = 1 \;\; and \;\; \delta(1) = \emptyset|\!\}$

1

- $P = a$

o(a) + $\sum_{b \in \delta(a)} b.\partial_b(a)$

= $\{\!|o(a) = 0, \delta(a) = \{a\} \;\; and \;\; \partial_a(a) = 1|\!\}$

0 + a.1

$\sim$ $\{\!|0 + x = x \;\; and \;\; x.1 \sim x|\!\}$

a

- $P = x + y$

$o(x + y) + \sum_{a \in \delta(x+y)} a.\partial_a(x + y)$

= $\{\!|o(x + y) = o(x) + o(y), \;\; \delta(x + y) = \delta(x) \cup \delta(y)$
$and \;\; \partial_a(x + y) = \partial_a(x) + \partial_a(y)|\!\}$

$o(x) + o(y) + \sum_{a \in \delta(x) \cup \delta(y)} a.(\partial_a(x) + \partial_a(y))$

$\sim$ $\{\!|$Properity of $\sum$ $|\!\}$

$o(x) + o(y) + \sum_{a \in \delta(x) \; and \; a \notin \delta(y)} a.(\partial_a(x) + \partial_a(y))$ + $\sum_{a \in \delta(y) \; and \; a \notin \delta(x)} a.(\partial_a(x) + \partial_a(y))$

$$+ \sum_{a\in\delta(x)\cap\delta(y)} a.(\partial_a(x) + \partial_a(y))$$

$\sim$ {|Definition 5.5.1, $x.(y+z) \sim x.y + x.z$ and "$\sim$" is congruence. |}

$$o(x) + o(y) + \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.(\partial_a(x) + 0) + \sum_{a\in\delta(y)\ and\ a\notin\delta(x)} a.(0 + \partial_a(y))$$

$$+ \sum_{a\in\delta(x)\cap\delta(y)} (a.\partial_a(x) + a.\partial_a(y))$$

$\sim$ {|Properity of $\sum$, $x + 0 \sim x$ and "$\sim$" is congruence. |}

$$o(x) + o(y) + \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.\partial_a(x) + \sum_{a\in\delta(y)\ and\ a\notin\delta(x)} a.\partial_a(y)$$

$$+ \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x) + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$$

$\sim$ {|$x + y \sim y + x$ |}

$$o(x) + \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.\partial_a(x) + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x)$$

$$+ o(y) + \sum_{a\in\delta(y)\ and\ a\notin\delta(x)} a.\partial_a(y) + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$$

$\sim$ {|Property of $\sum$ and "$\sim$" is congruence.|}

$$o(x) + \sum_{a\in\delta(x)} a.\partial_a(x) + (\text{o(y)} + \sum_{a\in\delta(y)} a.\partial_a(y)\ )$$

$\sim$ {|By induction and "$\sim$" is congruence.|}

$$x + y$$

- $P = x.y$

$$o(x.y) + \sum_{a\in\delta(x.y)} a.\partial_a(x.y)$$

$=$ {|$o(x.y) = o(x) \times o(y)$, $\delta(x.y) = \delta(x) \cup o(x) \otimes \delta(y)$
$and\ \ \partial_a(x.y) = \partial_a(x).y + o(x)\partial_a(y)$|}

$$o(x) \times o(y) + \sum_{a\in(\delta(x)\cup o(x)\otimes\delta(y))} a.(\partial_a(x).y + o(x)\partial_a(y)) \qquad (\alpha)$$

a) When o(x)= 0

(α)

$\sim$ {|$o(x) = 0$, $0 + x \sim x\ \ and\ \ 0.x \sim 0$
and "$\sim$" is congruence. |}

$$\sum_{a\in\delta(x)} a.\partial_a(x).y$$

$\sim$ $\{\!|o(x) = 0 \;\; and \;\; 0 + x \sim x \,|\!\}$

$(o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)).y$

$\sim$ {|By induction. and "$\sim$" is congruence. |}

$x.y$

b) When o(x)= 1

(α)

$\sim$ $\{\!|o(x) = 1, \;\; 1.x \sim x \;\; and$ Definition of $\otimes|\!\}$

$o(y) + \sum_{a\in\delta(x)\cup\delta(y)} a.(\partial_a(x).y + \partial_a(y))$

$\sim$ {|Property of $\sum$ |}

$o(y) + \sum_{a\in\delta(x)\; and\; a\notin\delta(y)} a.(\partial_a(x).y + \partial_a(y))$ + $\sum_{a\in\delta(y)\; and\; a\notin\delta(x)} a.(\partial_a(x).y + \partial_a(y))$

+ $\sum_{a\in\delta(x)\cap\delta(y)} a.(\partial_a(x).y + \partial_a(y))$

$\sim$ {|Definition 5.5.1 $\;\; and \;\; x.(y + z) \sim x.y + x.z$|}

$o(y) + \sum_{a\in\delta(x)\; and\; a\notin\delta(y)} a.(\partial_a(x).y + 0)$ + $\sum_{a\in\delta(y)\; and\; a\notin\delta(x)} a.(0.y + \partial_a(y))$

+ $\sum_{a\in\delta(x)\cap\delta(y)} (a.\partial_a(x).y + a.\partial_a(y))$

$\sim$ {|Property of $\sum$, $\;\; x + 0 \sim x \;\;$ and "$\sim$" is congruence. |}

$o(y) + \sum_{a\in\delta(x)\; and\; a\notin\delta(y)} a.\partial_a(x).y$ + $\sum_{a\in\delta(y)\; and\; a\notin\delta(x)} a.\partial_a(y)$

+ $\sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x).y$ + $\sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$

$\sim$ $\{\!|x + y \sim y + x \,|\!\}$

$o(y) + \sum_{a\in\delta(y)\; and\; a\notin\delta(x)} a.\partial_a(y)$ + $\sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$

+ $\sum_{a\in\delta(x)\; and\; a\notin\delta(y)} a.\partial_a(x).y$ + $\sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x).y$

$\sim$ {|Property of $\sum$ |}

$o(y) + \sum_{a\in\delta(y)} a.\partial_a(y)$ + $\sum_{a\in\delta(x)} a.\partial_a(x).y$

$\sim$ {|By induction and "$\sim$" is congruence. |}

$$y + \sum_{a\in\delta(x)} a.\partial_a(x).y$$

$\sim$ {|$x.y + z.y \sim (x+z).y$ |}

$$(1 + \sum_{a\in\delta(x)} a.(\partial_a(x)).y$$

$\sim$ {|$o(x) = 1$|}

$$(o(x) + \sum_{a\in\delta(x)} a.(\partial_a(x)).y$$

$\sim$ {|By induction and "$\sim$" is congruence. |}

$$x.y$$

- $P = x^*.y$

o$(x^*.y)$ + $\sum_{a\in\delta(x^*.y)} a.\partial_a(x^*.y)$

= {|$o(x^*.y) = o(y), \;\; \delta(x^*.y) = \delta(x)\cup\delta(y)$ $and\;\; \partial_a(x^*.y) = \partial_a(x).x^*.y + \partial_a(y)$|}

$$o(y) + \sum_{a\in\delta(x)\cup\delta(y)} a.(\partial_a(x).x^*.y + \partial_a(y))$$

$\sim$ {|Definition 5.5.1 , $x.(y+z) \sim x.y + x.z$ and $A\cup B$ = $(A\backslash B)\cup(B\backslash A)\cup(A\cap B)$ |}

$$o(y) + \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.(\partial_a(x)x^*y + 0) + \sum_{a\in\delta(y), a\notin\delta(x)} a.(0.x^*y + \partial_a(y))$$
$$+ \sum_{a\in\delta(x)\cap\delta(y)} (a.\partial_a(x)x^*y + a.\partial_a(y))$$

$\sim$ {|Property of $\sum$, $x + 0 \sim x$ $0.x \sim 0$ and "$\sim$" is congruence. |}

$$o(y) + \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.\partial_a(x)x^*y + \sum_{a\in\delta(y)\ and\ a\notin\delta(x)} a.\partial_a(y)$$
$$+ \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x)x^*y + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$$

$\sim$ {|$x + y \sim y + x$ |}

$$o(y) + \sum_{a\in\delta(y)\ and\ a\notin\delta(x)} a.\partial_a(y) + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(y)$$
$$+ \sum_{a\in\delta(x)\ and\ a\notin\delta(y)} a.\partial_a(x)x^*y + \sum_{a\in\delta(x)\cap\delta(y)} a.\partial_a(x)x^*y$$

$\sim$ {|Property of $\sum$ |}

$$o(y) + \sum_{a\in\delta(y)} a.\partial_a(y) + \sum_{a\in\delta(x)} a.\partial_a(x)x^*y$$

$\sim$ {|By induction and "$\sim$" is congruence. |}

$$y + \sum_{a\in\delta(x)} a.\partial_a(x)x^*y \qquad (\beta)$$

a) When o(x)= 0

(β)

$\sim$ {|$o(x) = 0, \quad 0 + x \sim x$ and "$\sim$" is congruence. |}

$$y + (o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)).x^*y$$

$\sim$ {|By induction and "$\sim$" is congruence. |}

$$y + x.x^*y$$

$\sim$ {|$x^*y \sim y + x.x^*y$ |}

$$x^*y$$

b) When o(x)= 1

(β)

$\sim$ {|Lemma 4.6.3 on page 79 |}

$$x^*y$$

- Conclusion: By structural induction, we conclude that:

$$x \sim o(x) + \sum_{a\in\delta(x)} a.\partial_a(x)$$

□

4.6.4 Proof of Proposition 4.5.7

Proposition 4.5.7: Let x and y be two processes in $BPA^*_{0,1}$, then

$$x \sqcap y \sim o(x) \times o(y) + \sum_{a\in\delta(x)\cap\delta(y)} a.(\partial_a(x) \sqcap \partial_a(y))$$

Proof:

$x \sqcap y$

$\sim \quad \{\!| P \sim P' \rightarrow P \sqcap Q \sim P' \sqcap Q \ \ and \ \ P \sqcap Q \sim Q \sqcap P$
and Proposition 4.5.6 $|\!\}$

$(o(x) + \sum_{a \in \delta(x)} a.\partial_a(x)) \sqcap$ (o(y) + $\sum_{a \in \delta(y)} a.\partial_a(y))$

$\sim \quad \{\!| P \sqcap (Q + R) \sim P \sqcap Q + P \sqcap R \ \ and \ \ P \sqcap Q \sim Q \sqcap P |\!\}$

$$\underbrace{o(x) \sqcap o(y) + o(x) \sqcap \sum_{a \in \delta(y)} a.\partial_a(y) + o(y) \sqcap \sum_{a \in \delta(x)} a.\partial_a(x)}_{\alpha}$$

$$+ \underbrace{\sum_{a \in \delta(x)} a.\partial_a(x) \sqcap \sum_{a \in \delta(y)} a.\partial_a(y)}_{\beta}$$

1. For α

   - When $o(x) = o(y) = 0$, for $P \sqcap 0 \sim 0$, we have $\alpha \sim 0 \sim o(x) \times o(y)$.
   - When $o(x) = o(y) = 1$, $o(x) \sqcap o(y) \sim 1$, $o(x) \sqcap \sum_{a \in \delta(y)} a.\partial_a(y) + o(y) \sqcap \sum_{a \in \delta(x)} a.\partial_a(x)$ is trace equivalence to 1 or 0. And since $1 + 0 \sim 1$ and $1 + 1 \sim 1$, then $\alpha \sim 1 \sim o(x) \times o(y)$.
   - When $o(x) = 1$ and $o(y) = 0$, α = $0 + 1 \sqcap \sum_{a \in \delta(y)} a.\partial_a(y) + \sum_{a \in \delta(y)} a.\partial_a(y) \sqcap 0$
     $= 0 + 1 \sqcap \sum_{a \in \delta(y)} a.\partial_a(y)$
     Form the definition of $\delta(y)$, we can see that $a \in \delta(y)$ can not be 1. We conclude, in this case that $\alpha \sim 0 + 0 \sim 0 \sim o(x) \times o(y)$.
   - When $o(x) = 0$ and $o(y) = 1$, same with above.

   We conclude, $\alpha \sim o(x) \times o(y)$.

2. For β
   β
   $\sim \quad \{\!| a.P \sqcap a.Q \sim a.(P \sqcap Q) \ \ and \ \ a.P \sqcap b.Q \sim 0 |\!\}$

   $\sum_{a \in \delta(x) \cap \delta(y)} a.(\partial_a(x) \sqcap \partial_a(y))$

- Conclusion: $x \sqcap y \sim o(x) \times o(y)$ + $\sum_{a \in \delta(x) \cap \delta(y)} a.(\partial_a(x) \sqcap \partial_a(y))$ holds.

$\square$

4.7 Conclusion

This chapter introduced the mathematical foundation of enforcing security policies in $CBPA^*_{01}$, a process algebra with test. The approach is based on the resolution of linear systems extracted from the computation of the intersection between a process capturing a given security property and another one capturing a sequential program. This intersection is recursively computed based on the notion of derivatives. The generated system could be resolved thanks to the famous Arden's Lemma. In the next chapter, we extend the approach to an extended version of $CBPA^*_{01}$.

Chapter 5

FASER (Formal and Automatic Security Enforcement by Rewriting by algebra) with Environment

This chapter introduces a formal program-rewriting approach that can automatically enforce security policies on untrusted programs. For a program P and a security policy Φ, we generate another program P' that respects the security policy and behaves like P except that it stops any execution path whenever the enforced security policy is about to be violated. The presented approach uses the $\mathcal{E}BPA^*_{0,1}$ algebra which is a variant of BPA (Basic Process Algebra) extended with variables, environments and conditions to formalize and resolve the problem. Finding the expected enforced program P' will turn to resolve a linear system for which we already know how to extract the solution by a polynomial algorithm.

5.1 Introduction

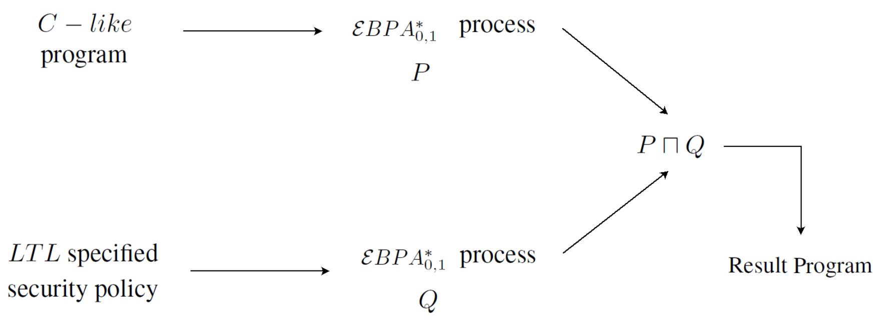
5.1.1 Methodology

To address the main problem introduced in Section 1.2, we present our methodology as following:

- We introduce a new algebra $\mathcal{E}BPA^*_{0,1}$, which extends the one in [48, 24, 60] by taking variables, conditions and environment into consideration.
- We suppose that the program P is a given as a process or it can be transformed to a process in $\mathcal{E}BPA^*_{0,1}$. This process algebra is expressive enough to handle a simplified version of C-like programs.
- We suppose that the security policy Φ is also given as a process or it can be transformed to a process in $\mathcal{E}BPA^*_{0,1}$. In [24], authors showed how a formula in LTL logic can be transformed

to a process in an algebra like $\mathcal{E}BPA^*_{0,1}$.

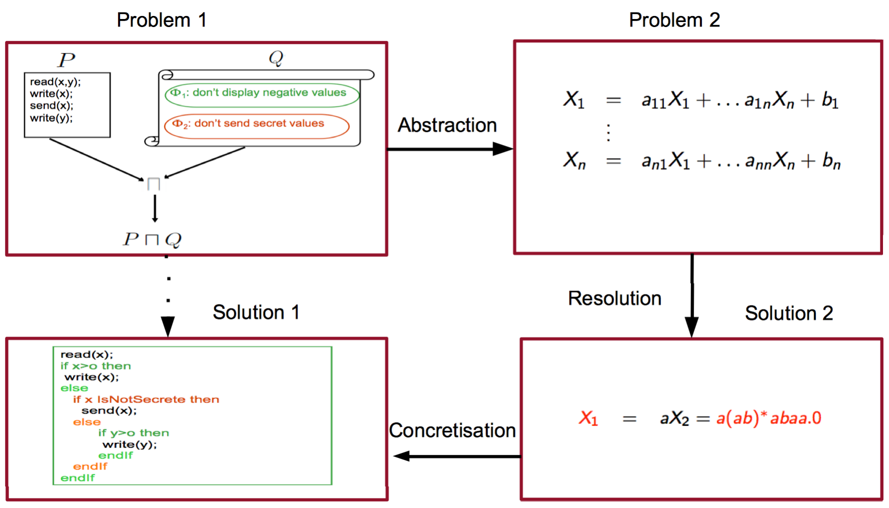
- We define an operator $\sqcap$ that enforces Φ on P as shown in Figure 5.1.



$C - like$ program
$\mathcal{E}BPA^*_{0,1}$ process P
$P \sqcap Q$
LTL specified security policy
$\mathcal{E}BPA^*_{0,1}$ process Q
Result Program

Figure 5.1: Security Policy Enforcement Process.

- Basically, the operator $\sqcap$ transforms the problem of enforcing Φ on P to a problem of generating a set of equations in $\mathcal{E}BPA^*_{0,1}$ as shown by Figure 5.2. We already know how to get the solution of the equations generated by $\sqcap$.



```
read(x,y);
write(x);
send(x);
write(y);
```

```
read(x);
if x>o then
 write(x);
else
   if x IsNotSecrete then
     send(x);
   else
         if y>o then
          write(y);
         endIf
   endIf
endIf
```

Problem 1
P
Q
Φ_1: don't display negative values
Φ_2: don't send secret values
$\sqcap$
$P \sqcap Q$
Abstraction
Problem 2
$X_1 = a_{11}X_1 + \ldots a_{1n}X_n + b_1$
$\vdots$
$X_n = a_{n1}X_1 + \ldots a_{nn}X_n + b_n$
Resolution
Solution 2
$X_1 = aX_2 = a(ab)^*abaa.0$
Concretisation
Solution 1

Figure 5.2: Approach.

- We prove that the secure version of program generated by $\sqcap$ is sound and complete with respect to the inputs. The soundness states that all the traces of the new generated process respect the security policy and they belong to the set of traces of the original insecure process. The completeness property, or transparency, on the other hand, states that any trace of the original program that respects the security policy should be kept in the secure version.

5.1.2 Ingredients and steps to solve the problem

More precisely, to address our problem, we have to follow the steps shown in Figure 5.1 and solve the following sub problems:

- Define a formal language to specify systems.
- Define a formal language to specify security policies.
- Formalize the link between inputs and output of $\sqcap$.
- Resolve the problem, i.e: find $P \sqcap Q$ of Figure 5.2.

5.2 A Formal Language to Specify Systems: $\mathcal{E}BPA^*_{0,1}$

To be able to specify more interesting systems, it is important to endow the algebra with conditions, variables and environments. This section gives syntax and semantics of $\mathcal{E}BPA^*_{0,1}$. We first give the definitions of some necessary ingredients like multi-sorted signature, multi-sorted algebra and environments. After that, we give the syntax and the semantics of $\mathcal{E}BPA^*_{0,1}$, which can be used to specify both programs and security policies. However since this can be difficult some times, we provide a more user-friendly interface to allow an end user to write his program in a $C-$ like program language. This $C-$ like language can be transferred into $\mathcal{E}BPA^*_{0,1}$ by a given function.

5.2.1 Ordered Sorted Algebras

To be able to specify variables with different types (boolean, integer, float, etc.), we introduce an ordered sorted algebra, which is a special case of Many-Sorted Algebras [2, 53].

Definition 5.2.1 (Multi-Sorted Signature) *A signature Ω is a pair $\langle S, F\rangle$, where S is a set of sorts and F is a set of function symbols such that F is equipped with a mapping type: $F \mapsto S^* \times S$, which expresses the type of each function symbol. Given a function $f \in F$, we write $f : \mathrm{s}_1 \times ... \times \mathrm{s}_n \mapsto \mathrm{s}$ to mean that type(f) = $(\mathrm{s}_1...\mathrm{s}_n, \mathrm{s})$.*

Here, we use the definition of Many-Sorted Algebras as given in [53].

Definition 5.2.2 (Many Sorted Algebra) *Let $\Omega = \langle S, F\rangle$ be a multi-sorted signature and $\mathcal{X}$ be a $S-$sorted set of variables ($\{X_\mathrm{s}\}_{\mathrm{s}\in S}$). We define the $T_\Omega(\mathcal{X}) = \bigcup\limits_{\mathrm{s}\in S} T_\Omega(\mathcal{X})_\mathrm{s}$ where $T_\Omega(\mathcal{X})_\mathrm{s}$ as the least set containing:*

1. *every variable of sort* s, *i.e., $\mathcal{X}_\mathrm{s} \subset T_\Omega(\mathcal{X})_\mathrm{s}$.*
2. *every nullary function symbol (constant) $c \in F$ with $c :\mapsto \mathrm{s}$.*

3. *every term $f(t_1, ..., t_n)$ where $f : \mathbf{s}_1 \times ... \times \mathbf{s}_n \mapsto \mathbf{s} \in F$ and each t_i is a term in $T_\Omega(\mathcal{X})_{\mathbf{s}_i}$, $1 \leq i \leq n$.*

To simplify the approach, we assume that the sorts used within the language $\mathcal{E}BPA^*_{0,1}$ form a lattice.

Definition 5.2.3 (Lattice) *A partially ordered set (Poset) $\langle\ S, \subseteq\ \rangle$ is called a lattice if:*

1. *$\forall \mathbf{s}_1, \mathbf{s}_2 \in S$, there exists a least upper bound or join $\mathbf{s}_{join} \in S$, such that $\mathbf{s}_1 \subseteq \mathbf{s}_{joint}$ and $\mathbf{s}_2 \subseteq \mathbf{s}_{joint}$.*
2. *$\forall \mathbf{s}_1, \mathbf{s}_2 \in S$, there exists a greatest lower bound or meet $s_{meet} \in S$, such that $\mathbf{s}_{meet} \subseteq \mathbf{s}_1$ and $\mathbf{s}_{meet} \subseteq \mathbf{s}_2$.*

The lowest upper bound and the greatest lower bound of S are denoted by $\cup S$ and $\cap S$ respectively.

In the sequel, we denote LSA as any many-ordered sorted algebra where its sorts form a lattice.

5.2.2 Environment

Here, we define an environment as a mapping from the set of variables to the set of domains.

Definition 5.2.4 (Environment) *Let $\mathcal{A} = T_\Omega(\mathcal{X})$ be a LSA on $\Omega = \langle S, F\rangle$. A valid environment on $\mathcal{A}$ is a mapping from $\mathcal{X} \longrightarrow S$.*

If for instance $S = \{bool, int, float\}$, then $e = \{x_1 \mapsto bool, x_2 \mapsto bool, x_3 \mapsto int, x_4 \mapsto float\}$ is an example of environment.

Since $\langle\ S, \subseteq\ \rangle$ forms a lattice, we consider that a variable is free in an environment, if its sort is equal to $\cup S$.

Definition 5.2.5 (Free variable) *We say that a variable x is a free in $e \in \mathcal{E}$, if: $(x \mapsto \mathbf{s_x}) \in e$ and $\mathbf{s_x}$ is the least upper bound of the lattice $\langle S, \subseteq\rangle$. i.e: $\mathbf{s_x} = \cup S$*

Notice that an environment (a mapping from $\mathcal{X}$ to S) is different from a substitution (a mapping from $\mathcal{X}$ to $T_\Omega(X)$). We are interested by a particular case of substitutions, called "free substitution", inspired by [49] and defined as follows:

Definition 5.2.6 (Free Substitution) *Let e be an environment. A substitution σ is considered as free in e (or e-free) if for all $x \mapsto t$ in σ, we have $x \in F_v(e)$.*

A substitution σ is considered as an unifier of two terms t_1 and t_2 if $\sigma(t_1)$ (shortly denoted by $t_1\sigma$) is equal to $t_2\sigma$. An unifier σ of two terms t_1 and t_2 is called *mgu* (Most General Unifier) if: $\forall\sigma'$ such that $t_1\sigma' = t_2\sigma'$, there exists σ" such that $\sigma \circ \sigma" = \sigma'$. Since variables are sorted, it's important that mgu respects the order between sorts, i.e. if $x \mapsto t$ belongs to the mgu, then the sort of t should be smaller or equal to the sort of x. We denote by $mgu(t_1, t_2, e)$, the mgu of t_1 and t_2 that respects the sorts of variables given within e. In Section 5.7, we give an algorithm (inspired by Martelli-Montanari reduction rules [45]) allowing computing $mgu(t_1, t_2, e)$ when it exists.

Here we present some useful definition for further discussion. The following definition separates a substitution on two parts.

Definition 5.2.7 *($\sigma_\prec$ and $\sigma_\approx$) Let σ be a substitution, V be a set of variables and e be an environment. We divide σ into two parts $\sigma_\prec^{V,e}$ and $\sigma_\approx^{V,e}$ (or shortly $\sigma_\prec$ and $\sigma_\approx$, when V and e is clear for environment) as following:*

$$\begin{array}{lcl} \sigma & = & \sigma_\prec \cup \sigma_\approx \\ \sigma_\approx & = & \{x \mapsto t | x \textit{ is a free variable in e and } x \notin \mathcal{V}\ \} \\ \sigma_\prec & = & \sigma - \sigma_\approx \end{array}$$

Here, we define a function that transforms a substitution to a condition.

***Definition 5.2.8** ($\ulcorner - \urcorner$) Let $\sigma \in \Gamma$, we define $\ulcorner - \urcorner : \Gamma \rightarrow \mathcal{B}$ as following:*

$$\begin{array}{rcl} \ulcorner \emptyset \urcorner & = & 1 \\ \ulcorner \{x \mapsto t\} \cup \sigma \urcorner & = & (x == t).\ulcorner \sigma \urcorner \end{array}$$

The following operator is used to compute the intersection of two actions.

Definition 5.2.9 (∇_e) *Let a_1, $a_2 \in \mathcal{A}$, $e \in \mathcal{E}$. We define $\nabla_e : \mathcal{A} \times \mathcal{A} \rightarrow \Sigma \times \Gamma$ as following:*

$$a_1 \nabla_e a_2 = \begin{cases} \{(\ulcorner \sigma_\prec \urcorner . a_1\sigma_\approx, \sigma_\approx)\}, & \text{if } \sigma = mgu(a_1, a_2, e) \text{ exists} \\ \{(0, \emptyset)\}, & \text{else} \end{cases}$$

Example 5.2.10 *Let $a_1 = a(x, y, z)$, $a_2 = a(\alpha, z, 3)$ and e be an environment such that $x, y, z \in N_v(e), \alpha \in F_v(e)$. We compute $a_1 \nabla_e a_2$ as following:*

$$
\begin{array}{ll}
 & a_1 \nabla_e a_2 \\
= & \quad \{\!|\sigma = mgu(a_1, a_2, e) = \{\alpha \mapsto x, y \mapsto z, z \mapsto 3\}\ |\!\} \\
 & \{(\ulcorner\sigma_{\prec}\urcorner.a_1, \sigma_{\approx})\} \\
= & \quad \{\!|\alpha \in F_v(e) \text{ and } \sigma = \sigma_{\approx} \cup \sigma_{\prec}\ |\!\} \\
 & \{(\ulcorner\sigma_{\prec}\urcorner.a_1, \sigma_{\approx})\}, \sigma_{\approx} = \{\alpha \mapsto x\} \\
= & \quad \{\!|\sigma_{\prec} = \sigma - \sigma_{\approx} = \{y \mapsto z, z \mapsto 3\}\ |\!\} \\
 & \{(\ulcorner\{y \mapsto z, z \mapsto 3\}\urcorner.a_1, \{\alpha \mapsto x\})\} \\
= & \quad \{\!|\ulcorner\{y \mapsto z, z \mapsto 3\}\urcorner = (y == z).(z == 3) \\
 & \quad \text{and } a_1 = a(x, y, z)\ |\!\} \\
 & \{((y == z).(z == 3).a(x, y, z), \{\alpha \mapsto x\})\}
\end{array}
$$

Let $2^{\mathcal{A}}$ be the set of all sets of actions in $\mathcal{A}$, then ∇_e can be extended to accept sets of actions as input:

Let A_1, $A_2 \in 2^{\mathcal{A}}$, $e \in \mathcal{E}$. We define $\nabla_e : 2^{\mathcal{A}} \times 2^{\mathcal{A}} \to \Sigma \times \Gamma$ as following:

$$(\{a_1\} \cup A_1) \nabla_e (\{a_2\} \cup A_2) =$$

$$
\begin{cases}
\{(\ulcorner\sigma_{\prec}\urcorner.a_1\sigma_{\approx}, \sigma_{\approx})\} \cup A_1 \nabla_e A_2, & \text{if } \sigma = mgu(a_1, a_2, e) \text{ exists} \\
 & \\
A_1 \nabla_e A_2, & \text{else}
\end{cases}
$$

In the remaining part of this paper, we adopt the following notations:

- $\mathcal{E}$: We use $\mathcal{E}_{\mathcal{A}}$ (or shortly $\mathcal{E}$ when $\mathcal{A}$ is clear from the context) to denote the set of all valid environments of $\mathcal{A}$.
- $\mathcal{V}(e)$: If e is an environment, $\mathcal{V}(e)$ denotes the set of variables in environment e.
- $\mathcal{V}(t)$: If t is a term, $\mathcal{V}(t)$ denotes the set of variables in t.
- $F_v(e)$: If e is an environment, $F_v(e)$ denotes the set of free variables in e.
- $N_v(e)$: If e is an environment, $N_v(e)$ denotes the set of non-free variables in e (i.e $N_v(e) = \mathcal{V}(e) - F_v(e)$).
- $s_e(x)$: If e is an environment, $s_e(x)$ denotes the sort of the variable x in e.

5.2.3 Syntax of $\mathcal{E}BPA^*_{0,1}$

In the sequel, we define $\mathcal{E}BPA^*_{0,1}$ as an extension of $BPA^*_{0,1}$ [9, 10, 48] that allows having conditions, variables and environment. More precisely:

- Let $\mathcal{A}$ be a Lattice Sorted Algebra (LSA $T_\Omega(\mathcal{X})$).
- Let a, a_1 and a_2 range over $\mathcal{A}$.
- Let $\mathcal{B}$ be defined by the following BNF-grammar.

  $b, b_1, b_2 ::= 0 \mid 1 \mid a_1 == a_2 \mid a_1 < a_2 \mid \bar{b} \mid b_1.b_2 \mid \quad b_1 + b_2$

  such that $(\mathcal{B}, +, ., \bar{}, 0, 1)$ is a boolean algebra
- Let c range over $\mathcal{B}$.
- Let e range over $\mathcal{E}$.

The syntax of $\mathcal{E}BPA^*_{0,1}$ ($BPA^*_{0,1}$ with boolean algebra and variable) is given by the following BNF-grammar:

$$P, Q ::= \; a \mid c \mid P + Q \mid P.Q \mid P^*Q \mid \lambda x P \mid [P]_e$$

Informally, the semantics of $\mathcal{E}BPA^*_{0,1}$ is as follows:

- a is a process that executes the atomic action a. We used to write $a(x_1, \ldots, x_n)$ to indicate that $x_1, \ldots, x_n$ are variables in a.
- c is a boolean process that is in a deadlock state if it is evaluated to 0 and finishes normally if it is evaluated to 1.
- $P + Q$ is a choice between two processes P and Q.
- $P.Q$ is a sequential composition of P and Q.
- P^*Q is the process that behaves like $P.(P^*Q) + Q$. It is a binary version of the Kleene star operator [40].
- $\lambda x P$ behaves like P except that we limit the scope of the variable x to P. For example, if we have $(\lambda x P).Q$, then the variable x in P will be different from any variable x in Q.
- $[P]_e$ defines the process P running in the environment e.

To reduce the number of parentheses, we use the following priorities between operators (from high to low): "[]", "λ", "$*$", ".", "$+$". Also, when there is no ambiguity, "." can be omitted and we use $\lambda(x.y)P$ as a shortcut for $\lambda x \lambda y P$.

The set of processes in $\mathcal{E}BPA^*_{0,1}$ will be denoted by $\mathcal{P}$ in the remaining part of this chapter.

5.2.4 Semantics of $\mathcal{E}BPA^*_{0,1}$

The operational semantics of $\mathcal{E}BPA^*_{0,1}$ is defined by the transition relation $\rightarrow \in \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ given by Table 5.2, where:

- $[\![-]\!]_{\mathcal{B}}$ is an evaluation function from $\mathcal{B}$ to $\{0, 1\}$.
- "$\downarrow$" is the unary relation on $\mathcal{P}$ and it allows knowing whether a process P can immediately terminate with success or not, defined by the inference rules given in Table 5.1.

Table 5.1: Definition of the Operator $\downarrow$.

$$(R^1)\frac{\square}{1\downarrow} \qquad\qquad (R^{[]_e})\frac{P\downarrow}{[P]_e\downarrow}$$

$$(R^*_{r\downarrow})\frac{Q\downarrow}{(P^*Q)\downarrow} \qquad\qquad (R^{\cdot}_{\downarrow})\frac{P\downarrow \quad Q\downarrow}{(P.Q)\downarrow}$$

$$(R^+_{l\downarrow})\frac{P\downarrow}{(P+Q)\downarrow} \qquad\qquad (R^+_{r\downarrow})\frac{Q\downarrow}{(P+Q)\downarrow}$$

$$(R^c)\frac{[\![c(v_1...v_n)]\!]_{\mathcal{B}}=1}{[c(x_1...x_n)]_e\downarrow} \quad (v_1...v_n) \in (s_e(x_1)...s_e(x_n))$$

- eff: $\mathcal{A} \times \mathcal{E} \longrightarrow \mathcal{E}$ is a function that updates an environment based on the effect of a given action. Right now, it is not necessary to have more precision on the function eff.

For rule $(R^{[]_e}_a)$, a process P in an environment e moves according to the moving possibility of process P and the environment e. For example if P can move to process P' by action $a(x_1...x_n)$ and for $1 \leq i \leq n$, v_i is the value for variable x_i in e, then a process P in e ($[P]_e$) moves to process $[P']_{eff(a(v_1...v_n),e)}$ by action $a(v_1...v_n)$, here $eff(a(v_1...v_n), e)$ returns a new environment, which is the result of applying the effect of the action $a(v_1...v_n)$ on e.

5.2.5 Handling a C-Like Programming Language

Now, we show how to transform a simplified version of C-like programs into $\mathcal{E}BPA^*_{0,1}$.

We consider the following C-like programming language:

-
  ```
  P::= ;| exit() | a| P;P′ | if (c) {P} else {P′} | while (c) do {P}
  | do {P} while (c)
  ```

Any program written in this language can be translated to $\mathcal{E}BPA^*_{0,1}$ by a function given in Table 5.3.

Table 5.2: Operational Semantics of $\mathcal{E}BPA^*_{0,1}$.

$$(R^a)\frac{\Box}{a\xrightarrow{a}1} \qquad (R^c)\frac{\Box}{c(x_1,\ldots,x_n)\xrightarrow{c(x_1,\ldots,x_n)}1}$$

$$(R^{\cdot}_l)\frac{P\downarrow \quad Q\xrightarrow{b}Q'}{P.Q\xrightarrow{b}Q'} \qquad (R^{\cdot}_r)\frac{P\xrightarrow{b}P'}{P.Q\xrightarrow{b}P'Q}$$

$$(R^+_l)\frac{P\xrightarrow{b}P'}{P+Q\xrightarrow{b}P'} \qquad (R^+_r)\frac{Q\xrightarrow{b}Q'}{P+Q\xrightarrow{b}Q'}$$

$$(R^*_l)\frac{P\xrightarrow{b}P'}{P^*Q\xrightarrow{b}P'.P^*Q} \qquad (R^*_r)\frac{Q\xrightarrow{b}Q'}{P^*Q\xrightarrow{b}Q'}$$

$$(R^\lambda)\frac{P[x\mapsto y]\xrightarrow{\alpha}P'}{\lambda xP\xrightarrow{\alpha}P'} : y\, is\, a\, fresh\, variable.$$

$$(R^{[]_e}_a)\frac{P \quad \overset{a(x_1\ldots x_n)}{\longrightarrow} \quad P'}{[P]_e \quad \overset{a(v_1\ldots v_n)}{\longrightarrow}[P']_{eff(a(v_1\ldots v_n),e)}}\ (v_1\ldots v_n)\in(s_e(x_1)\ldots s_e(x_n))$$

$$(R^{[]_e}_c)\frac{P\overset{c(x_1\ldots x_n)}{\longrightarrow}P',\ \ [c(x_1\ldots x_n)]_e\downarrow,\ \ [P']_e\xrightarrow{b}[Q]_{e'}}{[P]_e\xrightarrow{b}[Q]_{e'}}$$

Table 5.3: C-Like Language Translate Function.

$\lceil - \rceil$: C-Like $\longrightarrow \mathcal{E}BPA^*_{0,1}$		
$\lceil ; \rceil$	$=$	1
$\lceil exit() \rceil$	$=$	0
$\lceil P;P' \rceil$	$=$	$\lceil P\rceil.\lceil P'\rceil$
$\lceil a \rceil$	$=$	a
$\lceil if(c)\ \{P\}\ else\ \{P'\}\rceil$	$=$	$c.\lceil P\rceil + \overline{c}.\lceil P'\rceil$
$\lceil while(c)\ do\ \{P\}\rceil$	$=$	$(c.\lceil P\rceil)^*\overline{c}$
$do\ \{P\}\lceil while(c)\ \rceil$	$=$	$(\lceil P\rceil.c)^*\overline{c}$

5.3 A Formal Language to Specify Security Policies ($\mathcal{V}LTL$: LTL with variables)

We have already introduced a LTL-like logic language, now we can extend it to specify security policies with variables.

5.3.1 LTL-Like Logic for Specifying Security Policies

Suppose that a ranges over a finite set of actions $\mathcal{A}$ and c ranges over a finite set of conditions $\mathcal{C}$, then the syntax of $\mathcal{V}LTL$ is as following:

$$\Phi, \Phi_1, \Phi_2 \quad ::= \quad \top \mid \bot \mid 1 \mid a \mid c \mid \neg\Phi \mid \lambda x\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid X\Phi \mid \Phi_1 U \Phi_2 \mid \Phi_1.\Phi_2$$

Let τ be a trace and let τ^i be its suffix starting from the action at the i^{th} position. The semantics of a formula is given by $\models$ as follows:

- $\tau \models \top$, $\tau \not\models \bot$, $\epsilon \models 1$,
- $a \models b$ if there exists σ such that $\sigma(a) = b$,
- $c \models c'$ if there exists σ such that $\sigma(c) \geq c'$,
- $\tau \models \neg\Phi$ if $\tau \not\models \Phi$,
- $\tau \models \lambda x\Phi$ if $\tau \models \Phi[x \mapsto y]$, where y is a fresh variable.
- $\tau \models \Phi_1 \vee \Phi_2$ if $\tau \models \Phi_1$ or $\tau \models \Phi_2$
- $\tau \models \Phi_1 \wedge \Phi_2$ if $\tau \models \Phi_1$ and $\tau \models \Phi_2$
- $\tau \models X\Phi$ if $\tau^1 \models \Phi$
- $\tau \models \Phi_1 U \Phi_2$ if there exist k and σ such that $\tau^k\sigma \models \Phi_2$ and for all $0 \leq i < k$ $\tau^i\sigma \models \Phi_1$.
- $\tau \models \Phi_1.\Phi_2$ if there exist τ_1, τ_2 and σ such that $\tau = \tau_1.\tau_2$, $\tau_1\sigma \models \Phi_1$ and $\tau_2\sigma \models \Phi_2$.

Any formula Φ in the LTL-like logic can be translated to $\mathcal{E}BPA^*_{0,1}$ using the following four steps:

1. We keep applying the following rules on Φ, until all "$\neg$" operators are moved in front of atomic actions or conditions.
$$\begin{array}{rcl} \neg\bot & \longrightarrow & \top \\ \neg\top & \longrightarrow & \bot \\ \neg\neg\Phi & \longrightarrow & \Phi \\ \neg\lambda x\Phi & \longrightarrow & \lambda x\neg\Phi \end{array} \qquad \begin{array}{rcl} \neg(X\Phi) & \longrightarrow & X(\neg\Phi) \\ \neg(\Phi_1 \wedge \Phi_2) & \longrightarrow & \neg\Phi_1 \vee \neg\Phi_2 \\ \neg(\Phi_1 U \Phi_2) & \longrightarrow & (\neg\Phi_1) \vee (\Phi_1 U \neg\Phi_2) \\ \neg(\Phi_1.\Phi_2) & \longrightarrow & (\neg\Phi_1) \vee (\Phi_1.\neg\Phi_2) \end{array}$$

2. We keep applying the following rules until Φ becomes in a conjunctive normal form (CNF).
$$\begin{array}{rcl} X(\wedge_i\Phi_i) & \longrightarrow & \wedge_i(X\Phi_i) \\ ((\wedge_i\Phi_i)U(\wedge_j\Phi_j)) & \longrightarrow & \wedge_i \wedge_j (\Phi_i U \Phi_j) \\ (\Phi_1 \wedge \Phi_2) \vee \Phi_3 & \longrightarrow & (\Phi_1 \vee \Phi_3) \wedge (\Phi_2 \vee \Phi_3) \\ \Phi_1 \vee (\Phi_2 \wedge \Phi_3) & \longrightarrow & (\Phi_1 \vee \Phi_2) \wedge (\Phi_1 \vee \Phi_3) \end{array}$$

3. The previous steps transform any Φ to a form like $\wedge_{i=1}^{n}\Phi_i$, where each Φ_i does not contain the "$\wedge$" operator and all "$\neg$" operators are in front of atomic actions. Here, we use $\mathcal{A}_p$ to denote all the actions in program P. Now, we translate each Φ_i as a process in $\mathcal{E}BPA^*_{0,1}$ using the following function:

$\lceil - \rceil : LTL \longrightarrow \mathcal{E}BPA^*_{0,1}$					
$\lceil \bot \rceil$	$=$	0	$\lceil \neg c \rceil$	$=$	$\bar{c}$
$\lceil \top \rceil$	$=$	$(\sum_{a\in\mathcal{A}} a)^*1$	$\lceil X\Phi \rceil$	$=$	$(\sum_{a\in\mathcal{A}} a).\lceil \Phi \rceil$
$\lceil 1 \rceil$	$=$	1	$\lceil \lambda x\Phi \rceil$	$=$	$\lambda x \lceil \Phi \rceil$
$\lceil \neg 1 \rceil$	$=$	$(\sum_{a\in\mathcal{A}} a)^*(\sum_{a\in\mathcal{A}} a)$	$\lceil \Phi_1 \vee \Phi_2 \rceil$	$=$	$\lceil \Phi_1 \rceil + \lceil \Phi_2 \rceil$
$\lceil a \rceil$	$=$	a	$\lceil \Phi_1.\Phi_2 \rceil$	$=$	$\lceil \Phi_1 \rceil.\lceil \Phi_2 \rceil$
$\lceil \neg a \rceil$	$=$	$\sum_{a_i\in(\mathcal{A}/a)} a_i$	$\lceil \Phi_1 U \Phi_2 \rceil$	$=$	$\lceil \Phi_1 \rceil^* \lceil \Phi_2 \rceil$

Here, different from previous chapter, we define $A/a : 2^{\mathcal{A}} \times \mathcal{A} \rightarrow 2^{\Sigma^*}$ as following:

- $\emptyset/a = \emptyset$
- $$(\{x\} \cup A)/a = \begin{cases} \{x\} \cup (A/a) & \text{if mgu}(x, a) \text{ doesn't exist} \\ \{\neg\ulcorner\sigma_\prec\urcorner.x\} \cup (A/a) & \text{if } (\alpha,\sigma) \in x\nabla_e a, \sigma = \sigma_\prec \cup \sigma_\approx \text{ and } \sigma_\prec \neq \emptyset, \sigma_\approx = \emptyset \\ A/a & \text{else} \end{cases}$$

4. Finally, the enforcement of Φ on a process P becomes as following:

$$P \sqcap \Phi = P \sqcap \wedge_{i=1}^{n}\Phi_i = (P \sqcap \lceil\Phi_1\rceil)... \sqcap \lceil\Phi_n\rceil$$

Example 5.3.1 *Suppose $\mathcal{A} = \{s(x), s(y), p(x)\}$ we have a security policy Q in $\mathcal{V}LTL$ logic as following:*

$Q = \{[\neg s(3)] \wedge \neg[\lambda\beta((\beta > 3).s(\beta))]\}U\bot$

Policy Q is written in $\mathcal{V}LTL$ logic, we use the above rules to transform it into $\mathcal{E}BPA^*_{0,1}$ as following:

1. *Move all $\neg$ operators of Q in front of atomic actions or conditions.*

$$\begin{array}{ll} & Q = \{[\neg s(3)] \wedge \neg[\lambda\beta((\beta > 3).s(\beta))]\}U\bot \\ = & \quad \{\!|\neg\lambda x\Phi \longrightarrow \lambda x\neg\Phi\ |\!\} \\ & \{[\neg s(3)] \wedge \lambda\beta[\neg((\beta > 3).s(\beta))]\}U\bot \\ = & \quad \{\!|\neg(\Phi_1.\Phi_2) \longrightarrow (\neg\Phi_1) \vee (\Phi_1.\neg\Phi_2)\ |\!\} \\ & \{[\neg s(3)] \wedge \lambda\beta\Big(\neg(\beta > 3) \vee [(\beta > 3).\neg s(\beta)]\Big)\}U\bot \end{array}$$

2. *Make Q in a conjunctive normal form (CNF)*

$$
\begin{aligned}
& Q=\{[\neg s(3)] \wedge \lambda\beta\Big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\Big)\}U\bot \\
= \quad & \{\!|((\wedge_i \Phi_i)U(\wedge_j \Phi_j)) \longrightarrow \wedge_i \wedge_j \,(\Phi_i U \Phi_j)\ |\!\} \\
& \Big(\neg s(3)U\bot\Big) \wedge \Big(\{\lambda\beta\big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\big)\}U\bot\Big) \\
= \quad & \{\!|Q_1=[\neg s(3)]U\bot \text{ and } Q_2=\{\lambda\beta\big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\big)\}U\bot\ |\!\} \\
& Q_1 \wedge Q_2
\end{aligned}
$$

3. *Use function $\lceil - \rceil : \mathcal{V}LTL \longrightarrow \mathcal{E}BPA^*_{0,1}$ to transfer Q_1 and Q_2.*

$$
\begin{aligned}
& \lceil Q_1 \rceil = \lceil [\neg s(3)]U\bot \rceil \\
= \quad & \{\!|\lceil \Phi_1 U \Phi_2 \rceil = \lceil \Phi_1 \rceil^* \lceil \Phi_2 \rceil |\!\} \\
& \lceil \neg s(3) \rceil * \lceil \bot \rceil \\
= \quad & \{\!|\lceil \bot \rceil = 0\ |\!\} \\
& \lceil \neg s(3) \rceil * 0 \\
= \quad & \{\!|\lceil \neg a \rceil = \textstyle\sum_{a_i \in \lceil \mathcal{A}-\{a\} \rceil} a_i \text{ and } \mathcal{A}=\{s(x), s(y), p(x)\}\ |\!\} \\
& \{(x \neq 3).s(x) + (y \neq 3).s(y) + p(x)\}^* 0 \\
& \lceil Q_2 \rceil = \lceil \{\lambda\beta\big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\big)\}U\bot \rceil \\
= \quad & \{\!|\lceil \Phi_1 U \Phi_2 \rceil = \lceil \Phi_1 \rceil^* \lceil \Phi_2 \rceil |\!\} \\
& \{\lceil \lambda\beta\big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\big) \rceil\} * \lceil \bot \rceil \\
= \quad & \{\!|\lceil \bot \rceil = 0\ |\!\} \\
& \{\lceil \lambda\beta\big(\neg(\beta>3) \vee [(\beta>3).\neg s(\beta)]\big) \rceil\} * 0 \\
= \quad & \{\!|\lceil \lambda x \Phi \rceil = \lambda x \lceil \Phi \rceil\ |\!\} \\
& \{\lambda\beta\Big(\lceil \neg(\beta>3) \vee [(\beta>3).\neg s(\beta)] \rceil\Big)\} * 0 \\
= \quad & \{\!|\lceil \Phi_1 \vee \Phi_2 \rceil = \lceil \Phi_1 \rceil + \lceil \Phi_2 \rceil \text{ and } \lceil \Phi_1.\Phi_2 \rceil = \lceil \Phi_1 \rceil.\lceil \Phi_2 \rceil\ |\!\} \\
& \{\lambda\beta\Big(\lceil \neg(\beta>3) \rceil + [(\beta>3).\lceil \neg s(\beta) \rceil]\Big)\} * 0 \\
= \quad & \{\!|\lceil \neg c \rceil = \bar{c}\ |\!\} \\
& \{\lambda\beta\Big((\beta \leq 3) + [(\beta>3).\lceil \neg s(\beta) \rceil]\Big)\} * 0 \\
= \quad & \{\!|\lceil \neg a \rceil = \textstyle\sum_{a_i \in \lceil \mathcal{A}-\{a\} \rceil} a_i \text{ and } \mathcal{A}=\{s(x), s(y), p(x)\}\ |\!\} \\
& \{\lambda\beta\Big((\beta \leq 3) + [(\beta>3).p(x)]\Big)\} * 0
\end{aligned}
$$

4. *Finally, Q_1 and Q_2 are transformed into $\mathcal{E}BPA^*_{0,1}$ and the enforcement of Q on a process P becomes as following:*

$$P \sqcap_e Q = ((P \sqcap_e \lceil Q_1 \rceil) \sqcap_e \lceil Q_2 \rceil)$$

5.4 Formalize the Link Between Inputs and Output of $\sqcap$

5.4.1 Trace Based Equivalence

In this section, we introduce variants of trace-based equivalence that will be used to compare processes. We denote by Σ the closure of $\mathcal{A} \cup \{\epsilon\}$ using the operator "." such that $(\Sigma, ., \epsilon)$ is a monoid.

If τ and τ' are in Σ and they are equivalent with respect to the monoid properties ($x = \epsilon.x$, $x = x.\epsilon$, $(x.y).z = x.(y.z)$), we write $\tau \simeq \tau'$.

Definition 5.4.1 (Traces of a process) *Let x, x' and x'' be processes, α be an action or a condition. An element $\tau \in \Sigma$ is a trace for x if there exists a process y such that $x \overset{\tau}{\twoheadrightarrow} y$, where $\twoheadrightarrow$ is a relation in $\mathcal{P} \times \Sigma \times \mathcal{P}$ defined as following:*

$$\frac{\square}{x \overset{\epsilon}{\twoheadrightarrow} x}$$

$$\frac{x \overset{\tau}{\twoheadrightarrow} x' \qquad x' \overset{\alpha}{\rightarrow} x''}{x \overset{\tau.\alpha}{\twoheadrightarrow} x''}$$

The following ordering relation will be used to compare two traces.

Definition 5.4.2 (Trace Ordering ($\preceq$)) *Let τ and τ' be traces, a be an action, c be a condition, $x, x_1, ..., x_n$ be variables and β be a variable or a constant. We define $\preceq$ as the smallest ordering relation (transitivity, reflexivity and anti-symmetry) respecting the following properties:*

1. *If $\tau \simeq \tau'$, then $\tau \preceq \tau'$.*
2. *$\tau.c.\tau' \preceq \tau.\tau'$.*
3. *$\tau.(x == \beta).a(x_1...x...x_n).\tau' \preceq \tau.a(x_1...\beta...x_n).\tau'$.*

We are also interested in comparing traces in a specified environment using the following ordering relation.

Definition 5.4.3 (Trace Ordering in an Environment e. ($\sqsubseteq_e$)) *Let τ, τ' be traces and $e \in \mathcal{E}$. We define $\sqsubseteq_e$ as the smallest ordering relation (transitivity, reflexivity and anti-symmetry) respecting the following properties:*

1. *If $\tau \preceq \tau'$, then $\tau \sqsubseteq_e \tau'$.*
2. *$\tau\sigma \sqsubseteq_e \tau$ for any e-free substitution σ.*

Now, we extend the definitions of $\preceq$ and $\sqsubseteq_e$ to processes as follows.

Definition 5.4.4 (Process Ordering. ($\preceq$)) *Let P and Q be two processes. We say that $P \prec Q$, if $\forall \tau \in \Sigma$, we have:*

1. *$P{\downarrow}\tau$ then $\exists \tau' \in \Sigma$, such that:$Q{\downarrow}\tau'$ and $\tau \preceq \tau'$.*

2. $P \overset{\tau}{\twoheadrightarrow} 1$ *then* $\exists \tau' \in \Sigma$, *such that:*$Q \overset{\tau'}{\twoheadrightarrow} 1$ *and* $\tau \preceq \tau'$.

Definition 5.4.5 (Process Ordering in an Environment e **(** $(\sqsubseteq_e)$ **)** *Let P and Q be two processes, $e \in \mathcal{E}$. We say that $P \sqsubseteq_e Q$, if $\forall \tau \in \Sigma$, we have:*

1. $P{\downarrow}\tau$ *then* $\exists \tau' \in \Sigma$, *such that:*$Q{\downarrow}\tau'$ *and* $\tau \sqsubseteq_e \tau'$.

2. $P \overset{\tau}{\twoheadrightarrow} 1$ *then* $\exists \tau' \in \Sigma$, *such that:*$Q \overset{\tau'}{\twoheadrightarrow} 1$ *and* $\tau \sqsubseteq_e \tau'$.

It is clear from the previous definitions that $\preceq \subseteq \sqsubseteq_e$. Also $\preceq$ has many interesting properties such as those given by the following proposition.

Proposition 5.4.6 (Some Properties of $\preceq$) *The ordering $\preceq$ is a compatible with the operators ("$+$", "$.$" and "$*$"), i.e. for any processes P, P' and Q, we have:*

$P \preceq P' \Longrightarrow$

$$\begin{array}{llcllll} (1) & P+Q & \preceq & P'+Q & (2) & Q+P & \preceq & Q+P' \\ (3) & P.Q & \preceq & P'.Q & (4) & Q.P & \preceq & Q.P' \\ (5) & P^{*}Q & \preceq & P'^{*}Q & (6) & Q^{*}P & \preceq & Q^{*}P' \end{array}$$

Proof: See Section 5.7 (page 135). □

When a process runs in its environment, it produces closed traces. We use the following definition to compare processes running in environments.

Definition 5.4.7 (Ordering processes running in environments ($\sqsubseteq$ **))** *Let P and Q be two processes, $ep, eq \in \mathcal{E}$ and $\tau \in \Sigma$. We say that $[P]_{ep} \sqsubseteq [Q]_{eq}$, if:*

- $[P]_{ep}{\downarrow}\tau$ *then* $\exists \tau' \in \Sigma$ *such that* $[Q]_{eq}{\downarrow}\tau'$ *and* $\tau \simeq \tau'$.

Based on the previous ordering, we define the following equivalence relations:

Definition 5.4.8 ($\approx$) *Let P and Q be two processes. We say that $P \approx Q$, if $P \preceq Q$ and $Q \preceq P$.*

Definition 5.4.9 ($\sim_e$) *We say that two processes P and Q are trace-equivalent in $e \in \mathcal{E}$, denoted by $P \sim_e Q$, if $P \sqsubseteq_e Q$ and $Q \sqsubseteq_e P$.*

Definition 5.4.10 ($\sim$) *Let* e *and* e' *be in* $\mathcal{E}$. *We say that two processes* $[P]_e$ *and* $[Q]_{e'}$ *are trace equivalent and we write* $[P]_{e'} \sim [Q]_{e'}$, *if* $[P]_e \sqsubseteq [Q]_{e'}$ *and* $[Q]_{e'} \sqsubseteq [P]_e$.

The following proposition shows some interesting properties of $\approx$.

Proposition 5.4.11 $\approx$ *is a congruence relation with respect to operators ("+", "." and " *").*

Proof: Directly from Proposition 5.4.6 and Definition 5.4.8. □

More useful properties of $\approx$ are given by the following proposition.

Proposition 5.4.12 *Given three processes x, y and z, the following properties hold.*

$$\begin{array}{llcllllcl}
(B_1) & x+(y+z) & \approx & (x+y)+z & & (B_7) & 0.x & \approx & 0 \\
(B_2) & x.(y.z) & \approx & (x.y).z & & (B_8) & x.1 & \approx & x \approx 1.x \\
(B_3) & x+y & \approx & y+x & & (B_9) & x+0 & \approx & x \\
(B_4) & (x+y).z & \approx & x.z+y.z & & (B_{10}) & x^*y & \approx & y+xx^*y \\
(B_5) & x.(y+z) & \approx & x.y+x.z & & (B_{11}) & x+y \preceq z & \Rightarrow & x \preceq z,\ y \preceq z \\
(B_6) & x+x & \approx & x & & (B_{12}) & x.z+y \preceq z & \Rightarrow & x^*y \preceq z
\end{array}$$

Proof: similar to the proof of Proposition 4.4.7.

Given a program P and a security policy Φ, the goal of this work is to generate another program P' that respects the security policy Φ and preserves the good behaviors of P without introducing new ones. More precisely and as already stated in [48], P' should respect the two following conditions:

1. **Correctness:**
   - $P' \sqsubseteq P$: all the traces of P' are traces in P.
   - $P' \sqsubseteq \Phi$: all the traces of P' respect the security policy.
2. **Completeness:** If there exists P" such that P" $\sqsubseteq P$ and P" $\sqsubseteq \Phi$, then P" $\sqsubseteq P'$. This property involves that all the traces in P that respect the security policy are also in P'.

To simplify the presentation, we integrate the two previous conditions in one operator called greatest common factor with respect to an ordering relation $\sqsubseteq$ (denoted $gcf(\sqsubseteq)$) or shortly gcf if $\sqsubseteq$ is clear from the context) defined as follows:

Definition 5.4.13 (Greatest Common Factor ($gcf(\sqsubseteq)$)) *Let P and Q be two processes. The gcf of P and Q with respect to $\sqsubseteq$, denoted by $P \sqcap Q$, is a process R that respects the following three conditions:*

- $R \sqsubseteq P$.
- $R \sqsubseteq Q$.
- For all R' such that $R' \sqsubseteq P$ and $R' \sqsubseteq Q$, we have $R' \sqsubseteq R$.

Now, the problem of enforcing a security property Q on a program P in an environment e turns to a problem of finding $[P]_e \sqcap [Q]_e$.

To simplify the resolution of the problem, we extend the definition of a gcf to two ordering relations $\sqsubseteq_1$ and $\sqsubseteq_2$ as follows:

Definition 5.4.14 *($gcf(\sqsubseteq_1, \sqsubseteq_2)$) The $gcf(\sqsubseteq_1, \sqsubseteq_2)$ of two processes P and Q is a process R respecting the following conditions:*

- $R \sqsubseteq_1 P$
- $R \sqsubseteq_2 Q$
- *For all R' such that $R' \sqsubseteq_1 P$ and $R' \sqsubseteq_2 Q$, we have $R' \sqsubseteq_1 R$.*

Definition 5.4.15 ($\sqcap_e$) *Let e be an environment. We introduce $\sqcap_e$ as a shortcut for $gcf(\preceq, \sqsubseteq_e)$ ($\sqcap_e = gcf(\preceq, \sqsubseteq_e)$).*

Notice that a gcf is not unique, but they form an equivalence class as shown by the following proposition.

Proposition 5.4.16 *If $R_1 = P \sqcap_e Q$ and $R_2 = P \sqcap_e Q$, then $R_1 \approx R_2$*

Proof: From Definition 5.4.15, we have $R_1 \preceq R_2$ and $R_2 \preceq R_1 \Longrightarrow R_1 \approx R_2$. □

The following theorem gives a main result showing the relationship between $([P]_e \sqcap [Q]_e)$ and $[P \sqcap_e Q]_e$

Theorem 5.4.17 *Let P and Q be two processes and $e \in \mathcal{E}$, then we have:*

$$[P]_e \sqcap [Q]_e \sim [P \sqcap_e Q]_e$$

Proof: See Section 5.7 (page 141). □

This theorem reduce the problem of finding $[P]_e \sqcap [Q]_e$ to computing $P \sqcap_e Q$. In the next section, we propose an algorithm allowing to compute $P \sqcap_e Q$, but before that, we give some useful properties of $\sqcap_e$.

Proposition 5.4.18 *let P, Q and R be three processes, $e \in \mathcal{E}$ and a_1, $a_2 \in \mathcal{A}$. The following properties hold:*

$$\begin{aligned}
0 \sqcap_e a &\approx 0 \\
P \sqcap_e P &\approx P \\
P \sqcap_e (Q+R) &\approx P \sqcap_e Q + P \sqcap_e R \\
P \approx P' &\rightarrow P \sqcap_e Q \approx P' \sqcap_e Q \\
a.P \sqcap_e a.Q &\approx a.(P \sqcap_e Q)
\end{aligned}$$

5.5 Resolution of the Problem: Finding $P \sqcap Q$

In this section, we first introduce the notion of derivatives and we clarify the relationship between the intersection of two processes and the intersection of their derivatives. Then, we give the main proposition on which our algorithm for computing $P \sqcap_e Q$ is based. Finally, we give the algorithm that implements the operator $\sqcap_e$ together with some examples.

5.5.1 Linear Systems

Finding $P \sqcap Q$ turns to generate and solve linear equations having the following forms:

$$E_{\mathcal{P}} = \left\{ \begin{array}{lcl} X_1 & = & c_1 + \sum\limits_{a \in \mathcal{A}_1} \sum\limits_{\sigma \in \Gamma_1} a\sigma X_i \\ & \vdots & \\ X_n & = & c_n + \sum\limits_{a \in \mathcal{A}_1} \sum\limits_{\sigma \in \Gamma_n} a\sigma X_i \end{array} \right. \qquad E_{\Upsilon} = \left\{ \begin{array}{lcl} \Gamma_1 & = & \sigma_0^1 \cup \sum\limits_{i \in n} \sigma_i^1 \circ \Gamma_i \\ & \vdots & \\ \Gamma_n & = & \sigma_0^n \cup \sum\limits_{i \in n} \sigma_i^n \circ \Gamma_i \end{array} \right.$$

To solve $E_{\mathcal{P}}$ we need first to solve E_{Υ}. Since equations of E_{Υ} are in the closed semiring $(\Upsilon, \cup, \circ, \emptyset, \{\emptyset\})$ then the solution of any map $x \mapsto a \circ x \cup b$ is a^*b, where a^* satisfies $a = \{\emptyset\} \cup a^* \circ a = \{\emptyset\} \cup a + a \circ a^*$

Recall that a seminring [19] $(S, +, ., 0, 1)$ is a set S and two operators "+" and "." that respect the following properties for any a, b, c in S:

$$\begin{array}{rclcrcl} a+b &=& b+a & \quad & a.0 &=& 0.a \ = \ 0 \\ a+(b+c) &=& (a+b)+c & & a.1 &=& 1.a \ = \ a \\ a+0 &=& a & & a.(b+c) &=& a.b+a.c \\ a.(b.c) &=& (a.b).c & & (a+b).c &=& a.c+b.c \end{array}$$

E_Υ has "$\circ$" as operator ".", "$\cup$" as operator "+", $\emptyset$ as 0 and $\{\emptyset\}$ as 1. It has the form: $Y = D \circ Y \cup V$ where Y is a vector of length n containing sets of variables, D is a constant matrix $n \times n$ where each element is a set of substitutions and V is a vector of length n containing sets of substitutions. The solution is $Y = D^* \circ V$ where D^* can be computed using the generalized version of the Arden Lemma since $(2^\Upsilon, \circ, \{\emptyset\})$ is a monoid. Notice that if $D = d$ (i.e. D is 1×1), and $V = v$ then $D^* \circ V = d^* \circ v = (\{\emptyset\} \cup d \cup d \circ d \ldots) \circ v = v \cup d \circ v \cup d^2 \circ v \cup \ldots$. Also, for almost all the examples of our interest, we have $d^n = d$. In this case $d^* \circ v = d \circ v \cup v$

After we get the result of E_Υ, we can use it to eliminate all the Γ variables from $E_\mathcal{P}$ to get a new linear system $E'_\mathcal{P}$. $E'_\mathcal{P}$ has the following form $AX + B = X$ where A is a constant matrix of size $n \times n$, B is a constant vector of size n and X is a vector of variables of size n.

The system $E'_\mathcal{P}$ can be solved iteratively by using Arden's Lemma [6]. Also, a generalized version of the Arden Lemma shows that the solution of this system is $X = A^*B$, where A^* is computed as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}$$

and the result can be inductively generalized for matrices n by n as follows, where A is a $n-1$ by $n-1$ matrix:

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]^* = \left[\begin{array}{l|l} (A+BD^*C)^* & (A+BD^*C)^*BD^* \\ \hline (D+CA^*B)^*CA^* & (D+CA^*B)^* \end{array}\right]$$

These results hold because, under the trace equivalence, $\mathcal{E}BPA^*_{0,1}$ is a kind of monodic tree Kleene algebra [62] with a binary star Kleene operator.

5.5.2 Derivatives in $\mathcal{E}BPA^*_{0,1}$

We adapt the definition of derivatives introduced by Brzozowski in [11] as following:

Definition 5.5.1 (Derivative of a Process) *The derivative of a process x with respect to an action a, denote by $\partial_a(x)$, is defined as following:*

$\partial : \mathcal{A} \times \mathcal{P} \longrightarrow \mathcal{P}$		
$\partial_a(0)$	$=$	0
$\partial_a(1)$	$=$	0
$\partial_{b_1}(b_2)$	$=$	$\begin{cases} 1, & \text{if } b_1 = b_2 \\ 0, & \text{else} \end{cases}$
$\partial_a(x^*y)$	$=$	$\partial_a(x).x^*y + \partial_a(y)$
$\partial_a(x+y)$	$=$	$\partial_a(x) + \partial_a(y)$
$\partial_a(x.y)$	$=$	$\partial_a(x).y + o(x).\partial_a(y)$
$\partial_a(\lambda x P)$	$=$	$\partial_a(P[x \mapsto y]) : y\ is\ a\ fresh\ variable.$

Informally, the derivative of a process with respect to a given action is the remaining part of the process after the execution of this action, i.e:

$$\partial_a(x) = \sum_{\{x' \in \mathcal{P} \ \mid \ x \overset{a}{\to} x'\}} x'$$

Example 5.5.2 *Let $P = \{p(x).s(x) + s(x).p(y)\}^*0$ and $s(x), p(x), p(y)$ be actions, then we can compute $\partial_{s(x)}P$ as following:*

$$\begin{aligned}
& \partial_{s(x)}P = \partial_{s(x)}\Big(\{p(x).s(x) + s(x).p(y)\}^*0\Big) \\
= \quad & \{\!|\partial_a(x^*y) = \partial_a(x).x^*y + \partial_a(y) \ |\!\} \\
& \Big(\partial_{s(x)}\{p(x).s(x) + s(x).p(y)\}\Big).P + \partial_{s(x)}0 \\
= \quad & \{\!|\partial_a(0) = 0 \text{ and } x + 0 \approx x \ |\!\} \\
& \Big(\partial_{s(x)}\{p(x).s(x) + s(x).p(y)\}\Big).P \\
= \quad & \{\!|\partial_a(x+y) = \partial_a(x) + \partial_a(y) \ |\!\} \\
& \Big(\partial_{s(x)}(p(x).s(x)) + \partial_{s(x)}(s(x).p(y))\Big).P \\
= \quad & \{\!|\partial_a(x.y) = \partial_a(x).y + o(x).\partial_a(y) \ |\!\} \\
& \Big((\partial_{s(x)}p(x)).s(x) + o(p(x)).\partial_{s(x)}s(x) + (\partial_{s(x)}s(x)).p(y) + o(s(x)).\partial_{s(x)}p(y)\Big).P \\
= \quad & \{\!|o(a) = 0 \ |\!\} \\
& \Big((\partial_{s(x)}p(x)).s(x) + 0.\partial_{s(x)}s(x) + (\partial_{s(x)}s(x)).p(y) + 0.\partial_{s(x)}p(y)\Big).P \\
= \quad & \{\!|0.x \approx 0 \text{ and } x + 0 \approx x \ |\!\} \\
& \Big((\partial_{s(x)}p(x)).s(x) + (\partial_{s(x)}s(x)).p(y)\Big).P \\
= \quad & \{\!|\text{if } b_1 = b_2, \partial_{b_1}(b_2) = 1 \text{ ; else } \partial_{b_1}(b_2) = 0 \ |\!\} \\
& \Big(0.s(x) + 1.p(y)\Big).P \\
= \quad & \{\!|0.x \approx 0 \text{ and } 1.x \approx x \ |\!\} \\
& p(y).P
\end{aligned}$$

Let τ, τ_1 and τ_2 be traces in Σ and $\mathcal{T}$ be a subset of Σ. The notion of derivatives can be extended to traces and sets of traces as following:

$$\begin{array}{rcl} \partial_\epsilon(P) & = & P \\ \partial_{\tau_1.\tau_2}(P) & = & \partial_{\tau_2}(\partial_{\tau_1}(P)) \\ \partial_\tau(\mathcal{P}) & = & \bigcup_{P\in\mathcal{P}}\{\partial_\tau(P)\} \\ \partial_{\mathcal{T}}(P) & = & \bigcup_{\tau\in\mathcal{T}}\{\partial_\tau(P)\} \end{array}$$

Definition 5.5.3 (Immediate Successful Termination of a Process) *The function $o(x)$ allows to know whether $x\downarrow$ holds or not and it is defined as follows:*

$o : \mathcal{P} \longrightarrow \{0, c, 1\}$		
$o(0)$	$=$	$o(a) \ = \ o(c) \ = 0$
$o(1)$	$=$	1
$o(x^*y)$	$=$	$o(y)$
$o(x+y)$	$=$	$o(x) + o(y)$
$o(x.y)$	$=$	$o(x).o(y)$
$o(\lambda xP)$	$=$	$o(P[x \mapsto y]) : y\, is\, a\, fresh\, variable.$

Definition 5.5.4 (Immediate Possible Actions of a Process) *The following function δ gives the immediate possible actions of a process:*

$\delta : \mathcal{P} \longrightarrow \mathcal{P}$		
$\delta(0)$	$=$	$\emptyset$
$\delta(1)$	$=$	$\emptyset$
$\delta(a)$	$=$	$\{a\}$
$\delta(c)$	$=$	$\{c\}$
$\delta(x^*y)$	$=$	$\delta(x) \cup \delta(y)$
$\delta(x+y)$	$=$	$\delta(x) \cup \delta(y)$
$\delta(x.y)$	$=$	$\delta(x) \cup o(x) \otimes \delta(y)$
$\delta(\lambda xP)$	$=$	$\delta(P[x \mapsto y]) : y\, is\, a\, fresh\, variable.$

Intuitively, $\delta(x) = \{a | x \overset{a}{\to} x'\}$ where $\otimes$ is defined as following:

$$\begin{array}{rcl} \otimes : \{0,1\} \times 2^\Sigma & \longrightarrow & 2^\Sigma \\ (0, S) & \mapsto & \emptyset \\ (1, S) & \mapsto & S \end{array}$$

Example 5.5.5 *Let $Q = \{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\}^*0$ and $s(\alpha)$ and $p(x)$ be actions, then we can compute $\delta(Q)$ as following:*

$$
\begin{aligned}
& \delta(Q) = \delta\Big(\{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\}^*0\Big) \\
= \quad & \qquad \{\!|\delta(x^*y) = \delta(x) \cup \delta(y) \;|\!\} \\
& \delta\{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\} \cup \delta(0) \\
= \quad & \qquad \{\!|\delta(0) = \emptyset \;|\!\} \\
& \delta\{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\} \\
= \quad & \qquad \{\!|\delta(x + y) = \delta(x) \cup \delta(y) \;|\!\} \\
& \delta\{p(x)\} \cup \delta\{[\lambda\alpha((\alpha > 3).s(\alpha))]\} \\
= \quad & \qquad \{\!|\delta(a) = \{a\} \;|\!\} \\
& \delta\{\lambda\alpha((\alpha > 3).s(\alpha))\} \cup \{p(x)\} \\
= \quad & \qquad \{\!|\delta(\lambda x P) = \delta(P[x \mapsto y]) : y\, is\, a\, fresh\, variable. \;|\!\} \\
& \delta\{(\alpha_1 > 3).s(\alpha_1)\} \cup \{p(x)\} \\
= \quad & \qquad \{\!|\delta(x.y) = \delta(x) \cup o(x) \otimes \delta(y) \;|\!\} \\
& \delta\{(\alpha_1 > 3)\} \cup o((\alpha_1 > 3)) \otimes \delta\{s(\alpha_1)\} \cup \{p(x)\} \\
= \quad & \qquad \{\!|\delta(c) = \{c\} \;|\!\} \\
& o((\alpha_1 > 3)) \otimes \delta\{s(\alpha_1)\} \cup \{p(x), (\alpha_1 > 3)\} \\
= \quad & \qquad \{\!|o(c) = 0 \text{ and } 0 \otimes \mathcal{S} = \emptyset \;|\!\} \\
& \{p(x), (\alpha_1 > 3)\}
\end{aligned}
$$

Here, we define $\delta_{\mathcal{B}}(x) = \{c | c \in \delta(x) \wedge c \in \mathcal{B}\}$ and $\delta_{\mathcal{A}}(x) = \delta(x) - \delta_{\mathcal{B}}(x)$

By the above definitions, we can establish the relationship between a process and its derivatives. This relationship is important to prove a main result given in the sequel.

Proposition 5.5.6 *For a process P, we have: $P \approx o(P) + \sum\limits_{\alpha \in \delta(P)} \alpha\ \partial_\alpha(P)$*

Proof: The proof is available in Section 5.7 (Page 145).

□

5.5.3 Main Proposition

Hereafter, we give the main proposition that shows the relationship between the intersection of two processes and the intersection of their derivatives.

First, we give some useful definition and intermediary results. Before computing the intersection between two processes, we need to transform them to their guarded form. The notion of guarded process is defined as following.

Definition 5.5.7 (Guarded Process and Guarded Intersection) *Let α be an action or a condition, P and Q be two processes and R and S be processes or intersections. We define a guarded process and a guarded intersection as following:*

- *α is a guarded*
- *$\alpha.P$ is guarded*
- *$P \sqcap_e Q$ if P is guarded and Q is guarded*
- *$R + S$ is guarded if R is guarded and S is guarded*

In the following, we give a rewriting system allowing to transform any intersection between processes to an equivalent guarded one.

Definition 5.5.8 (Normalization Rules) *We denote by $\mathcal{NR}$, the following rewriting system.*

- $0.P \to 0, 1.P \to P$
- $P + 0 \to P, 0 + P \to P$
- $P \sqcap_e 0 \to 0, 0 \sqcap_e P \to 0$
- $P \sqcap_e 1 \to 1, 1 \sqcap_e P \to 1$

- $R \sqcap_e P + Q \to R \sqcap_e P + R \sqcap_e Q$
- $P + Q \sqcap_e R \to P \sqcap_e R + Q \sqcap_e R$
- $R \sqcap_e (P + Q).T \to R \sqcap_e P.T + R \sqcap_e Q.T$
- $(P + Q).T \sqcap_e R \to P.T \sqcap_e R + Q.T \sqcap_e R$

- $R \sqcap_e P^*Q \to R \sqcap_e P.(P^*Q) + R \sqcap_e Q$
- $P^*Q \sqcap_e R \to P.(P^*Q) \sqcap_e R + Q \sqcap_e R$
- $R \sqcap_e P^*Q.T \to R \sqcap_e P.(P^*Q).T + R \sqcap_e Q.T$
- $P^*Q.T \sqcap_e R \to P.(P^*Q).T \sqcap_e R + Q.T \sqcap_e R$

- If R is a guarded process, then $R \sqcap_e (\lambda x P) \to R \sqcap_e P[x \mapsto y]$, y is a fresh variable.

- If R is a guarded process, then $(\lambda x P) \sqcap_e R \to P[x \mapsto y] \sqcap_e R$, y is a fresh variable.
- If R is a guarded process, then $R \sqcap_e (\lambda x P).Q \to R \sqcap_e P[x \mapsto y].Q$, y is a fresh variable.
- If R is a guarded process, then $(\lambda x P).Q \sqcap_e R \to P[x \mapsto y].Q \sqcap_e R$, y is a fresh variable.

If R is a process or an intersection, we denote by $R\Downarrow$ its normal form using the rewriting system $\mathcal{NR}$.

Proposition 5.5.9 *Let R be a process or an intersection. We have:*

- *$R\Downarrow$ exists.*
- *$R\Downarrow$ is guarded.*

Proof: The prof follows from Definitions 5.5.7 and 5.5.8 □

Once, we transform a process to its guarded form, we need to know how to compute $\alpha.P \sqcap_e \beta.Q$, to generate intersection. To that end, we need first to compute the intersection between α and β and then compute the intersection between P and Q. However, we need to carefully take care of the following fact: the intersection between α and β can succeed only under a substitution that needs to be propagated forward to the intersection of P and Q, and, on the other side, the intersection of P and Q can succeed only under a substitution that needs to be propagated backward to the intersection of α and β.

Hereafter, we give some definitions that help us to better formalize the issue of forward and backward propagation of substitutions.

Definition 5.5.10
- *Suppose that Γ_1 and Γ_2 are sets of substitutions, we define their composition $(\circ)$ as following:*

  $\Gamma_1 \circ \Gamma_2 = \bigcup_{\sigma \in \Gamma_1} \bigcup_{\sigma' \in \Gamma_2} \{\sigma \circ \sigma'\}$

- *Let Υ be the set of substitutions form $\mathcal{X} \to \mathcal{A}$. We extend the definition of Υ to $\Upsilon(X)$ to include also unknown or partially known substitutions. More precisely, $\Upsilon(X)$ is the smallest set respecting the following conditions:*
  - $\Upsilon \subset \Upsilon(X)$
  - *$z \in \Upsilon(X)$ for any variable z that range over substitutions.*
  - *if σ_x and σ_y are in $\Upsilon(X)$ then $\sigma_x \circ \sigma_y$ is in $\Upsilon(X)$*

Definition 5.5.11 (T) *Let $e \in \mathcal{E}$, $a_1, a_2 \in \mathcal{A}$, $c \in \mathcal{B}$, P and Q be two processes and R and S be processes or intersections. We inductively define the function T as follows:*

$T : \mathcal{P} \longrightarrow 2^{\Upsilon(X)}$			
$T(0)$	$=$	$\emptyset$	
$T(1)$	$=$	$\emptyset$	
$T(R+S)$	$=$	$T(R) \cup T(S)$	
$T(a_1.P \sqcap_e a_2.Q)$	$=$	$\{\sigma\} \circ \Gamma_{P \sqcap_e Q}$	*if* $a_1 \nabla_e a_2 = \{(a,\sigma)\}$
$T(c.P \sqcap_e Q)$	$=$	$\Gamma_{P \sqcap_e Q}$	
$T(P \sqcap_e c.Q)$	$=$	$\Gamma_{P \sqcap_e Q}$	

Example 5.5.12 *Let P, Q be two processes and $s(\alpha_1), s(\alpha_2), s(x), s(y)$ be actions, then we can compute $T\Big(\big(\underline{(y>3).s(x).P \sqcap_e (\alpha_1>5).s(\alpha_1).Q}\big) + \big(\underline{s(y).P \sqcap_e s(\alpha_2).Q}\big)\Big)$ as following:*

$$T\Big(\big(\underline{(y>3).s(x).P \sqcap_e (\alpha_1>5).s(\alpha_1).Q}\big) + \big(\underline{s(y).P \sqcap_e s(\alpha_2).Q}\big)\Big)$$

$$= \qquad \{\!| T(R+S) = T(R) \cup T(S) \,|\!\}$$

$$T\big(\underline{(y>3).s(x).P \sqcap_e (\alpha_1>5).s(\alpha_1).Q}\big) \cup T\big(\underline{s(y).P \sqcap_e s(\alpha_2).Q}\big)$$

$$= \qquad \{\!| T(c.P \sqcap_e Q) = \Gamma_{P \sqcap_e Q} \,|\!\}$$

$$\underline{\Gamma_{s(x).P \sqcap_e (\alpha_1>5).s(\alpha_1).Q}} \cup T\big(\underline{s(y).P \sqcap_e s(\alpha_2).Q}\big)$$

$$= \qquad \{\!| T(a_1.P \sqcap_e a_2.Q) = \{\sigma\} \circ \Gamma_{P \sqcap_e Q} \text{ if } a_1 \nabla_e a_2 = \{(a,\sigma)\} \text{ and } s(y) \nabla_e s(\alpha_2) = \{(s(y), \{\alpha_2 \mapsto y\})\} \,|\!\}$$

$$\underline{\Gamma_{s(x).P \sqcap_e (\alpha_1>5).s(\alpha_1).Q}} \cup \{\alpha_2 \mapsto y\} \circ \underline{\Gamma_{P \sqcap_e Q}}$$

For the sake of clarity, in the remaining part of this chapter, we use the color red and underline to indicate variables that range over sets of substitutions; the color blue and underline to indicate variables that range over intersections between processes.

The following proposition gives the relationship between the intersection of two processes in terms of the intersections between their derivatives for some particular cases.

Proposition 5.5.13 *For $P, Q \in \mathcal{P}, e \in \mathcal{E}$, $a_1, a_2 \in \mathcal{A}, c_2 \in \mathcal{B}$, if $\mathcal{V}(a_1.P) \subseteq N_v(e)$, we have:*

$$1)\begin{cases} \underline{a_1.P \sqcap_e a_2.Q} \approx \sum\limits_{\sigma_\partial \,\in\, \underline{\Gamma_{(P\sigma_\delta \sqcap_e Q\sigma_\delta)}}} a\sigma_\partial.\big(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\big) \\ \qquad\qquad \textit{where } (a, \sigma_\delta) \in a_1 \nabla_e a_2 \\ \underline{\Gamma_{a_1.P \sqcap_e a_2.Q}} = T((\underline{a_1.P \sqcap_e a_2.Q}) \Downarrow) \end{cases}$$

$$2)\begin{cases} \underline{a_1.P \sqcap_e c_2.Q} \approx \sum\limits_{\sigma_\partial \,\in\, \underline{\Gamma_{(a_1 P \sqcap_e Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big) \\ \underline{\Gamma_{a_1.P \sqcap_e c_2.Q}} = T((\underline{a_1.P \sqcap_e c_2.Q}) \Downarrow) \end{cases}$$

For $P, Q \in \mathcal{P}, e \in \mathcal{E}$, $a_2 \in \mathcal{A}, c_1, c_2 \in \mathcal{B}$, if $\mathcal{V}(c_1.P) \subseteq N_v(e)$, we have:

$$3)\begin{cases} \underline{c_1.P \sqcap_e a_2.Q} \approx \sum\limits_{\sigma_\partial \,\in\, \underline{\Gamma_{(P \sqcap_e a_2.Q)}}} c_1\sigma_\partial.\big(\underline{P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial}\big) \\ \underline{\Gamma_{c_1.P \sqcap_e a_2.Q}} = T((\underline{c_1.P \sqcap_e a_2.Q}) \Downarrow) \end{cases}$$

$$4)\begin{cases} \underline{c_1.P \sqcap_e c_2.Q} \approx \sum\limits_{\sigma_\partial \,\in\, \underline{\Gamma_{(P \sqcap_e Q)}}} (c_1.c_2)\sigma_\partial.\big(\underline{P\sigma_\partial \sqcap_e Q\sigma_\partial}\big) \\ \underline{\Gamma_{c_1.P \sqcap_e c_2.Q}} = T((\underline{c_1.P \sqcap_e c_2.Q}) \Downarrow) \end{cases}$$

Proof: The proof is available in Section 5.7 (Page 147).

□

In a special case when both P and Q do not contain free variables, we can simplify proposition 5.5.13 as follows:

Corollary 5.5.14 *For $P, Q \in \mathcal{P}, s \in \mathcal{E}$, $a_1, a_2 \in \mathcal{A}, c_1, c_2 \in \mathcal{B}$, $\mathcal{V}(a_1.P) \subseteq N_v(e), \mathcal{V}(c_1.P) \subseteq N_v(e), \mathcal{V}(a_2.Q) \subseteq N_v(e), \mathcal{V}(c_2.Q) \subseteq N_v(e)$, we have:*

1. $\underline{a_1.P \sqcap_e a_2.Q} \approx a.(\underline{P \sqcap_e Q\sigma_\delta})$ *where* $(a, \sigma_\delta) \in a_1 \nabla_e a_2$.
2. $\underline{c_1.P \sqcap_e a_2.Q} \approx c_1.(\underline{P \sqcap_e a_2.Q})$.
3. $\underline{a_1.P \sqcap_e c_2.Q} \approx c_2.\big(\underline{(a_1.P) \sqcap_e Q}\big)$.
4. $\underline{c_1.P \sqcap_e c_2.Q} \approx (c_1.c_2).\big(\underline{P \sqcap_e Q}\big)$.

Let $E_\Upsilon(x_1, ..., x_n)$ be a linear system include variables $x_1, ..., x_n$, then we introduce our main theorem as follows:

Theorem 5.5.15 *Let $P, Q \in \mathcal{P}$, $e \in \mathcal{E}$ and $\mathcal{V}(P) \subseteq N_v(e)$, we have the two following equations:*

1. $$X = P \sqcap_e Q \approx o(P).o(Q) + \sum_{(a,\sigma_\delta) \in \delta_{\mathcal{A}}(P) \nabla_e \delta_{\mathcal{A}}(Q)} \sum_{\sigma_\partial \in \Gamma_{X_1}} a\sigma_\partial.X_1' +$$
$$\sum_{\substack{(c_1,a_2) \in \\ \delta_{\mathcal{B}}(P) \times \delta_{\mathcal{A}}(Q)}} \sum_{\sigma_\partial \in \Gamma_{X_2}} c_1\sigma_\partial.X_2' + \sum_{\substack{(a_1,c_2) \in \\ \delta_{\mathcal{A}}(P) \times \delta_{\mathcal{B}}(Q)}} \sum_{\sigma_\partial \in \Gamma_{X_3}} c_2\sigma_\partial.X_3' + \sum_{\substack{(c_1,c_2) \in \\ \delta_{\mathcal{B}}(P) \times \delta_{\mathcal{B}}(Q)}} \sum_{\sigma_\partial \in \Gamma_{X_4}} (c_1.c_2)\sigma_\partial.X_4'$$
$$\begin{array}{lcl lcl} X_1' & = & (\partial_{a_1}P)\sigma_\delta\sigma_\partial \sqcap_e (\partial_{a_2}Q)\sigma_\delta\sigma_\partial & X_2' & = & (\partial_{c_1}P)\sigma_\partial \sqcap_e (a_2.\partial_{a_2}Q)\sigma_\partial \\ X_3' & = & (a_1.\partial_{a_1}P)\sigma_\partial \sqcap_e (\partial_{c_2}Q)\sigma_\partial & X_4' & = & (\partial_{c_1}P)\sigma_\partial \sqcap_e (\partial_{c_2}Q)\sigma_\partial \end{array}$$

2. $$\Gamma_X = \Gamma_{P \sqcap_e Q} = T((P \sqcap_e Q)\Downarrow) \in E_\Upsilon(\Gamma_{X_1}, \Gamma_{X_2}, \Gamma_{X_3}, \Gamma_{X_4})$$
$$\begin{array}{lcl lcl} X_1 & = & (\partial_{a_1}P)\sigma_\delta \sqcap_e (\partial_{a_2}Q)\sigma_\delta & X_2 & = & \partial_{c_1}P \sqcap_e a_2.\partial_{a_2}Q \\ X_3 & = & a_1.\partial_{a_1}P \sqcap_e \partial_{c_2}Q & X_4 & = & \partial_{c_1}P \sqcap_e \partial_{c_2}Q \end{array}$$

Proof: Based on Proposition 5.5.6 and 5.5.13, we can do similar proof with the Proposition 4.5.7. □

In above theorem, after we expand the right side of the second equation by applying the function T, the equation will be in the form of $\Gamma_X = d_1 \circ \Gamma_{X_1} \cup d_2 \circ \Gamma_{X_2} \cup ... \cup v$, where $d_1, d_2 ... v$ are constants (sets of substitutions) and $\Gamma_X, \Gamma_{X_1}...$ are variables (if $X = P \sqcap_e Q$ then we say: $\Gamma_X = \Gamma_{P \sqcap_e Q}$).

If we apply the above theorem on the new generated variables of the right side of each equation recursively until we can not find new variables, we will get two linear systems as discussed in Section 5.5.1.

To simplify the presentation of the algorithm, we introduce the following definitions.

Definition 5.5.16 (First Order Intersection) *Let $P, Q \in \mathcal{P}$, $a_1, a_2 \in \mathcal{A}$, $c_1, c_2 \in \mathcal{B}$ and $e \in \mathcal{E}$, we define function $\mathcal{I} : \mathcal{P} \to \mathcal{P}$ as following:*

- $\mathcal{I}(P + Q) = \mathcal{I}(P) + \mathcal{I}(Q)$

- $\mathcal{I}(a_1.P \sqcap_e a_2.Q) = \sum_{\sigma_\partial \in \Gamma_{(P\sigma_\delta \sqcap_e Q\sigma_\delta)}} a\sigma_\partial.(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta)$

  where $(a, \sigma_\delta) \in a_1 \nabla_e a_2$

- $\mathcal{I}(c_1.P \sqcap_e a_2.Q) = \sum_{\sigma_\partial \in \Gamma_{(P \sqcap_e a_2.Q)}} c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)$
- $\mathcal{I}(a_1.P \sqcap_e c_2.Q) = \sum_{\sigma_\partial \in \Gamma_{(a_1 P \sqcap_e Q)}} c_2\sigma_\partial.\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big)$
- $\mathcal{I}(c_1.P \sqcap_e c_2.Q) = \sum_{\sigma_\partial \in \Gamma_{(P \sqcap_e Q)}} (c_1.c_2)\sigma_\partial.\big(P\sigma_\partial \sqcap_e Q\sigma_\partial\big)$

Definition 5.5.17 (($(P \sqcap_e Q)\downharpoonleft$)) *We denote by $(P \sqcap_e Q)\downharpoonleft$ the normal form of $P \sqcap_e Q$ and define it as follows.*
$(P \sqcap_e Q)\downharpoonleft = \mathcal{I}((P \sqcap_e Q) \Downarrow)$

By the help of above definitions, we can simplify Theorem 5.5.15 as following:

Theorem 5.5.18 *Let P and Q be two processes and $e \in \mathcal{E}$, if $\mathcal{V}(P) \subseteq N_v(e)$ then we have:*

1. $P \sqcap_e Q \approx (P \sqcap_e Q)\downharpoonleft$
2. $\Gamma_{P \sqcap_e Q} = T((P \sqcap_e Q)\Downarrow)$

Proof: Based on proposition 5.5.6 and 5.5.13. we can do a similar proof with Proposition 4.5.7. □

5.5.4 Algorithm

Based on Theorem 5.5.18, we write an algorithm allowing generating a linear system where $P \sqcap_e Q$ could be extracted from its solutions. This algorithm is as follows:

Algorithm 1 Let X and Y be two $\mathcal{E}BPA^*_{0,1}$ processes, $e \in \mathcal{E}$, and $\mathcal{V}(X) \subseteq N_v(e)$ calculate $X \sqcap Y$, where $G = \phi$, $E = \phi$, $S = \{(X, Y)\}$

```
1: DO
    Get one element $s = (P, Q)$ from $S$.
    Remove $s$ from $S$.

    $E \longleftarrow E \cup \{e \longleftarrow \big(P \sqcap_e Q \approx (P \sqcap_e Q)\downharpoonleft\big)\}$
    $G \longleftarrow G \cup \{g \longleftarrow \big(\Gamma_{P \sqcap_e Q} = T((P \sqcap_e Q)\Downarrow)\big)\}$

    For each  $P_i \sqcap_e Q_i\sigma_\partial$ in the right side of $e$,
      If  $P_i \sqcap_e Q_i$ does not appear (modulo commutativity of $\sqcap_e$ and ACIT of $+$) in the left side
        of any equation in $E$
      Do $S = S \cup (P_i, Q_i)$ ;
      End If
    End For each
  WHILE  $(S \neq \phi)$

2: Solve equations in $G$ to get all the values of $\Gamma$.
3: Substitute all $\Gamma$ in $E$.

4: While there exists $P_i \sqcap_e Q_i$ in the right side of any equation in $E$ that does not appear
     (modulo commutativity of $\sqcap_e$) in the left side on any equation do
     $E \longleftarrow E \cup \{e \longleftarrow \underline{P_i \sqcap Q_i} \approx (\underline{P_i \sqcap_e Q_i})\lfloor\}$
   End While
5: Return the solution of the linear system $E$.
```

- ACIT of $+$ is an abbreviation of Associativity ($(x + y) + z \approx x + (y + z)$), Commutativity ($x + y \approx y + x$), Identity ($x + x \approx x$) and Triviality ($x + 0 \approx 0 + x \approx x$, $0.x \approx 0$ and ($1.x \approx x.1 \approx x$).

- The algorithm terminates since the number of partial derivatives of regular terms x, denoted by $\mathcal{PD}(x)$ and defined as ∂_{Σ^∞}(x), is finite as shown by Brzozowski [11].

- The complexity of the algorithm for given inputs P and Q is $O(||P|| \times ||Q||)$, where $||P||$ is the size of P, i.e. the number of elements in $\Sigma \cup \{0, 1\}$ in P. In fact, it was proved in [5] that the number of partial derivatives of P is smaller than $||P|| + 1$. Also, since any equation is the intersection of an element from $\mathcal{PD}(P)$ and another from $\mathcal{PD}(Q)$, then the number of equations N is no more than $(||P|| + 1) \times (||Q|| + 1)$ and their resolutions can be done by elimination using Arden's Lemma [6] fewer than N times.

- The system E and G generated by the algorithm are linear systems. They can be solved by using the method we discussed in Section 5.5.1.

Using the previous algorithm, we can present some examples of intersections using our approach.

5.5.5 Simple Example of Intersection between Processes

Suppose that we have $e \in \mathcal{E}$, $F_v(e) = \{\alpha, \beta, \gamma\}$ and $N_v(e) = \{x, y, z\}$ and two processes P and Q as following:

$P = [((y > 3).s(x).s(y)] + [!(y > 3).s(y)]$

$Q = \lambda[(\alpha)][(\alpha > 5).s(\alpha)]^{*}0$

Now, we can compute $P \sqcap_e Q$, where $G = \phi$, $E = \phi$ and $S = \{(P, Q)\}$

- Step 1: Generate E and G:

$\underline{X_1} = \underline{P \sqcap_e Q}$

$\approx$ $\{\!|$Proposition 5.5.18 $|\!\}$

$(\underline{P \sqcap_e Q})\downharpoonleft$

$\approx$ $\{\!| R \sqcap_e P^{*}Q \approx R \sqcap_e P.(P^{*}Q) + R \sqcap_e Q |\!\}$

$P \sqcap_e \lambda(\alpha)[(\alpha > 5).s(\alpha)].Q + P \sqcap_e 0$

$\approx$ $\{\!| P \sqcap_e 0 \approx 0$ and $P + 0 \approx P |\!\}$

$P \sqcap_e \lambda(\alpha)[(\alpha > 5).s(\alpha)].Q$

$\approx$ $\{\!| P + Q \sqcap_e R \approx P \sqcap_e R + Q \sqcap_e R |\!\}$

$(y > 3).s(x).s(y) \sqcap_e \lambda(\alpha)[(\alpha > 5).s(\alpha)].Q + !(y > 3).s(y) \sqcap_e \lambda(\alpha)[(\alpha > 5).s(\alpha)].Q$

$\approx$ $\{\!| R \sqcap_e (\lambda x P).Q \approx R \sqcap_e P[x \mapsto y].Q$, y is a fresh variable. $|\!\}$

$(y > 3).s(x).s(y) \sqcap_e (\alpha_{10} > 5).s(\alpha_{10}).Q + !(y > 3).s(y) \sqcap_e (\alpha_{11} > 5).s(\alpha_{11}).Q$

$\approx$ $\{\!| \underline{c_1.P \sqcap_e c_2.Q} \approx \sum_{\sigma_\partial \in \underline{\Gamma_{(P \sqcap_e Q)}}} (c_1.c_2)\sigma_\partial.(\underline{P\sigma_\partial \sqcap_e Q\sigma_\partial}) |\!\}$

$$\sum_{\sigma_\partial \in \underline{\Gamma_{s(x).s(y) \sqcap_e s(\alpha_{10}).Q}}} [(y > 3).(\alpha_{10} > 5)]\sigma_\partial.[\underline{s(x).s(y) \sqcap_e (s(\alpha_{10}).Q)\sigma_\partial}] +$$
$$\sum_{\sigma_\partial \in \underline{\Gamma_{s(y) \sqcap_e s(\alpha_{11}).Q}}} [!(y > 3).(\alpha_{11} > 5)]\sigma_\partial.[\underline{s(y) \sqcap_e (s(\alpha_{11}).Q)\sigma_\partial}]$$

Here $\underline{\Gamma_{P \sqcap_e Q}} = \underline{\Gamma_{X_1}}$, $\underline{\Gamma_{s(x).s(y) \sqcap_e s(\alpha_{10}).Q}} = \underline{\Gamma_{X_2}}$
and $\underline{\Gamma_{s(y) \sqcap_e s(\alpha_{11}).Q}} = \underline{\Gamma_{X_3}}$:
$\underline{X_1} = \underline{P \sqcap_e Q}$

$$\approx \sum_{\sigma_\partial \in \Gamma_{X_2}} [(y > 3).(\alpha_{10} > 5)]\sigma_\partial.[s(x).s(y) \sqcap_e (s(\alpha_{10}).Q)\sigma_\partial] +$$
$$\sum_{\sigma_\partial \in \Gamma_{X_3}} [!(y > 3).(\alpha_{11} > 5)]\sigma_\partial.[s(y) \sqcap_e (s(\alpha_{11}).Q)\sigma_\partial] \quad (e_1)$$
$$\Gamma_{X_1} = T((P \sqcap_e Q)\Downarrow) = \Gamma_{X_2} \cup \Gamma_{X_3} \quad (g_1)$$

$$X_3 = s(y) \sqcap_e s(\alpha_{11}).Q$$
$$\approx \quad \{\!| \text{Simplification} |\!\}$$
$$s(y) \quad (e_3)$$
$$\Gamma_{X_3} = \{\{\alpha_{11} \mapsto y\}\} \quad (g_3)$$

Here $\Gamma_{s(y) \sqcap_e Q} = \Gamma_{X_4}$,
$$X_2 = s(x).s(y) \sqcap_e s(\alpha_{10}).Q$$

$$\approx \quad \{\!| \text{Simplification} |\!\}$$

$$\sum_{\sigma_\partial \in \Gamma_{X_4}} s(x).[s(y) \sqcap_e Q\sigma_\partial] \quad (e_2)$$
$$\Gamma_{X_2} = \{\{\alpha_{10} \mapsto x\}\} \circ \Gamma_{X_4} \quad (g_2)$$

Here $\Gamma_{s(y) \sqcap_e (s(\alpha_{40}).Q)} = \Gamma_{X_5}$,
$$X_4 = s(y) \sqcap_e Q$$

$$\approx \quad \{\!| \text{Simplification} |\!\}$$

$$\sum_{\sigma_\partial \in \Gamma_{X_5}} [(\alpha_{40} > 5)]\sigma_\partial.[s(y) \sqcap_e (s(\alpha_{40}).Q)\sigma_\partial] \quad (e_4)$$
$$\Gamma_{X_4} = \Gamma_{X_5} \quad (g_4)$$

$$X_5 = s(y) \sqcap_e s(\alpha_{40}).Q$$
$$\approx \quad \{\!| \text{Simplification} |\!\}$$
$$s(y) \quad (e_5)$$
$$\Gamma_{X_5} = \{\{\alpha_{40} \mapsto y\}\} \quad (g_5)$$

We have $E = \{e_1, e_2, e_3, e_4, e_5\}, G = \{g_1, g_2, g_3, g_4, g_5\}$

- Step 2: Solve G to get all the values of Γ.
  $\Gamma_{X_4} = \Gamma_{X_5} = \{\{\alpha_{40} \mapsto y\}\}$

$$\Gamma_{X_2} = \{\alpha_{10} \mapsto x\}\} \circ \Gamma_{X_4} = \{\{\alpha_{10} \mapsto x, \alpha_{40} \mapsto y\}\} = \{\sigma_2\}$$
$$\Gamma_{X_3} = \{\{\alpha_{11} \mapsto y\}\} = \{\sigma_3\}$$

- Step 3: Now, we substitute Γ in the equation set E.

$$X_1 = P \sqcap_e Q$$

$$\approx \sum_{\sigma_\partial \in \Gamma_{X_2}} [(y > 3).(\alpha_{10} > 5)]\sigma_\partial.[s(x).s(y) \sqcap_e (s(\alpha_{10}).Q)\sigma_\partial] +$$
$$\sum_{\sigma_\partial \in \Gamma_{X_3}} [!(y > 3).(\alpha_{11} > 5)]\sigma_\partial.[s(y) \sqcap_e (s(\alpha_{11}).Q)\sigma_\partial] \quad (e_1)$$
$$\approx (y > 3).(x > 5).[s(x).s(y) \sqcap_e (s(x).Q)\sigma_2] + !(y > 3).(y > 5).[s(y) \sqcap_e (s(y).Q)\sigma_3]$$
$$\approx (y > 3).(x > 5).X'_2 + !(y > 3).(y > 5).X'_3$$

- Step 4: Since there are new intersections X'_2, X'_3, we can generate new equations in E.

$$X'_2 = [s(x).s(y) \sqcap_e s(x).Q\sigma_2] = s(x).[s(y) \sqcap_e Q\sigma_2] = s(x).X'_4$$
$$X'_3 = s(y)$$
$$X'_4 = s(y) \sqcap_e Q\sigma_2 \approx [s(y) \sqcap_e ((\alpha_{40} > 5)s(\alpha_{40}).Q)\sigma_2]$$
$$= [s(y) \sqcap_e (y > 5).s(y).Q\sigma_2] = (y > 5).[s(y) \sqcap_e s(y).Q\sigma_2] = (y > 5).X'_5$$

$$X'_5 = s(y) \sqcap_e s(y).Q\sigma_2 \approx s(y)$$

- Step 5: Solve the equations set E by using Arden's Lemma and variable substitution:

$$X'_4 \approx (y > 5).X'_5 \approx (y > 5).s(y)$$
$$X'_2 \approx s(x).X'_4 \approx s(x).(y > 5).s(y)$$
$$X_1 \approx (y > 3).(x > 5).X'_2 + !(y > 3).(y > 5).X'_3$$
$$\approx (y > 3).(x > 5).s(x).(y > 5).s(y) + !(y > 3).(y > 5).s(y)$$

5.6 Examples

With our theoretical foundation, now we can present examples with more interesting programming language and security policies specified in $\mathcal{V}LTL$.

5.6.1 Example for a C-Like Language:

Suppose we have $e \in \mathcal{E}$, $F_v(e) = \{\alpha, \beta\}$ and $N_v(e) = \{x, y\}$.

Let P be the following program:

```
while (x<8)
   {
     if(y>5)
     {  print(x); send(y);  }
     s(x);
   }
```

Let s and p denote the actions $send$ and $print$ respectively. By using the transfer function introduced in Section 5.2.5, The program P can be specified in $\mathcal{E}BPA^*_{0,1}$ by the following process:

$P = \{((x < 8).[(y > 5).p(x).s(y)+!(y > 5).1].s(x)\} * (!(x < 8).1)$

Suppose that we have a security policy Q stating that we can send only values that are bigger than 3. Q can be specified in the $\mathcal{V}LTL$ logic as following:

$Q = \{[\neg s(\beta)] \vee [\lambda\alpha((\alpha > 3).s(\alpha))]\}U\bot$

Here, as defined in 5.2.3, we use λx to define the available scope of variables. By using the rules of Section 5.3 in page 97, security policies Q can be rewrote into $\mathcal{E}BPA^*_{0,1}$ as following:

$Q = \{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\} * 0$

Now, we can compute $P \sqcap_e Q$, where $G = \phi$, $E = \phi$, $E' = \phi$ and $S = \{(P, Q)\}$

- Inputs:
  $P = \{(x < 8).[(y > 5).p(x).s(y)+!(y > 5).1].s(x)\} * (!(x < 8).1)$
  $Q = \{p(x) + [\lambda\alpha((\alpha > 3).s(\alpha))]\}^*0$

- Step 1: Generate E and G.

  $X_1 = P \sqcap_e Q$

$\approx$ $\{|$Proposition 5.5.18 $|\}$

  $(P \sqcap_e Q)\downharpoonleft$

  Let's transform $P \sqcap_e Q$ using the rules in Definition 5.5.8

  $X_1 = P \sqcap_e Q$

$\approx$ $\{\!|$rules in Definition 5.5.8 $|\!\}$

$$(x<8).[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e p(x).Q+$$
$$(x<8).[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e (\alpha_{11}>3).s(\alpha_{11}).Q$$

$\approx$ $\{\!|\underline{c_1.P \sqcap_e a_2.Q} \approx c_1.(\underline{P \sqcap_e a_2.Q})|\!\}$

$$(x<8).\underline{\{[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e p(x).Q\}}+$$
$$(x<8).[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e (\alpha_{11}>3).s(\alpha_{11}).Q$$

$\approx$ $\{\!|\underline{c_1.P \sqcap_e c_2.Q} \approx \sum_{\sigma_\partial \in \underline{\Gamma_{(P\sqcap_e Q)}}} (c_1.c_2)\sigma_\partial.(\underline{P\sigma_\partial \sqcap_e Q\sigma_\partial})|\!\}$

$$(x<8).\underline{\{[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e p(x).Q\}}+$$
$$\sum_{\sigma_\partial \in \underline{\Gamma_{X_3}}} [(x<8).(\alpha_{11}>3)]\sigma_\partial.$$
$$\underline{\{[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e (s(\alpha_{11}).Q)\sigma_\partial\}} \quad (e_1)$$

Here: $\underline{\Gamma_{X_1}} = \underline{\Gamma_{X_2}} \cup \underline{\Gamma_{X_3}} \quad (g_1)$

The rest of the linear system can be generated by repeating the above process.

$\underline{X_2}$ $= \underline{\{[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e p(x).Q\}}$
$\approx (y>5).[\underline{p(x).s(y).s(x).P \sqcap_e p(x).Q}]+!(y>5).[\underline{s(x).P \sqcap_e p(x).Q}]$
$\approx (y>5).\underline{X_4}+!(y>5).X_5 \quad (e_2)$
Here: $\underline{\Gamma_{X_2}} = \underline{\Gamma_{X_4}} \cup \underline{\Gamma_{X_5}} \quad (g_2)$

$\underline{X_3}$ $= \underline{\{[(y>5).p(x).s(y)+!(y>5).1].s(x).P \sqcap_e s(\alpha_{11}).Q\}}$
$\approx (y>5).[\underline{p(x).s(y).s(x).P \sqcap_e s(\alpha_{11}).Q}]+!(y>5).[\underline{s(x).P \sqcap_e s(\alpha_{11}).Q}]$
$\approx (y>5).\underline{X_6}+!(y>5).\underline{X_7} \quad (e_3)$
Here: $\underline{\Gamma_{X_3}} = \underline{\Gamma_{X_6}} \cup \underline{\Gamma_{X_7}} \quad (g_3)$

$\underline{X_4}$ $= p(x).[\underline{s(y).s(x).P \sqcap_e Q}] \approx p(x).\underline{X_8} \quad (e_4)$
Here: $\underline{\Gamma_{X_4}} = \underline{\Gamma_{X_8}} \quad (g_4)$

$\underline{X_5}$ $= 0 \quad (e_5)$
Here: $\underline{\Gamma_{X_5}} = \emptyset \quad (g_5)$

$\underline{X_6}$ $= 0 \quad (e_6)$
Here: $\underline{\Gamma_{X_6}} = \emptyset \quad (g_6)$

$\underline{X_7}$ $= s(x).[\underline{P \sqcap_e Q}] = s(x).\underline{X_1} \quad (e_7)$
Here: $\underline{\Gamma_{X_7}} = \{\{\alpha_{11} \mapsto x\}\} \circ \underline{\Gamma_{X_1}} \quad (g_7)$

$\underline{X_8}$ $= \sum_{\sigma_\partial \in \underline{\Gamma_{X_9}}} (\alpha_{80}>3)\sigma_\partial.[\underline{s(y).s(x).P \sqcap_e (s(\alpha_{80}).Q)\sigma_\partial}] \quad (e_8)$

Here: $\Gamma_{X_8} = \Gamma_{X_9} \quad (g_8)$

$X_9 \quad = s(y).[s(x).P \sqcap_e Q] = s(y).X_{10} \quad (e_9)$

Here: $\Gamma_{X_9} = \{\{\alpha_{80} \mapsto y\}\} \circ \Gamma_{X_{10}} \quad (g_9)$

$X_{10} \quad = \sum_{\sigma_\partial \in \Gamma_{X_{11}}} (\alpha_{100} > 3)\sigma_\partial.[s(x).P \sqcap_e (s(\alpha_{100}).Q)\sigma_\partial] \quad (e_{10})$

Here: $\Gamma_{X_{10}} = \Gamma_{X_{11}} \quad (g_{10})$

$X_{11} \quad = s(x).[P \sqcap_e Q] = s(x).X_1 \quad (e_{11})$

Here: $\Gamma_{X_{11}} = \{\{\alpha_{100} \mapsto x\}\} \circ \Gamma_{X_1} \quad (g_{11})$

We have $E = \{e_1, ..., e_{11}\}, G = \{g_1, ..., g_{11}\}$

- Step 2: Solve G to get all the values of Γ.

  Here: $\Gamma_{X_{10}} = \Gamma_{X_{11}} = \{\{\alpha_{100} \mapsto x\}\} \circ \Gamma_{X_1}$

  $\Gamma_{X_8} = \Gamma_{X_9} = \{\{\alpha_{80} \mapsto y\}\} \circ \Gamma_{X_{10}} = \{\{\alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\} \circ \Gamma_{X_1}$

  $\Gamma_{X_2} = \Gamma_{X_4} \cup \Gamma_{X_5} = \Gamma_{X_4} = \Gamma_{X_8} = \{\{\alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\} \circ \Gamma_{X_1}$

  $\Gamma_{X_3} = \Gamma_{X_6} \cup \Gamma_{X_7} = \Gamma_{X_7} = \{\{\alpha_{11} \mapsto x\}\} \circ \Gamma_{X_1}$

  $\Gamma_{X_1} = \Gamma_{X_2} \cup \Gamma_{X_3} = \{\{\alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\} \circ \Gamma_{X_1} \cup \{\{\alpha_{11} \mapsto x\}\} \circ \Gamma_{X_1}$
  $= \{\{\alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\{\alpha_{11} \mapsto x\}\} \circ \Gamma_{X_1}$

  $\Gamma_{X_1} = \{\{\alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\{\alpha_{11} \mapsto x\}\}$

  $\Gamma_{X_2} = \{\{\alpha_{11} \mapsto x, \alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\}$

  $\Gamma_{X_3} = \{\{\alpha_{11} \mapsto x, \alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}\}$
  Let $\sigma = \{\alpha_{11} \mapsto x, \alpha_{80} \mapsto y, \alpha_{100} \mapsto x\}$

- Step 3: Now we can substitute Γ in E.

$X_1 \quad = P \sqcap_e Q \approx (P \sqcap_e Q)\downharpoonleft$

$$\approx (x < 8).\{[(y > 5).p(x).s(y) + !(y > 5).1].s(x).P \sqcap_e p(x).Q\} + \sum_{\sigma_\partial \in \Gamma_{X_3}} [(x < 8).(\alpha_{11} > 3)]\sigma_\partial.\{[(y > 5).p(x).s(y) + !(y > 5).1].s(x).P \sqcap_e (s(\alpha_{11}).Q)\sigma_\partial\}$$

$$\approx (x < 8).\{[(y > 5).p(x).s(y) + !(y > 5).1].s(x).P \sqcap_e p(x).Q\}$$
$$+(x < 8).(x > 3).\{[(y > 5).p(x).s(y) + !(y > 5).1].s(x).P \sqcap_e s(x).Q\sigma\}$$

$$\approx (x < 8).X_2 + (x < 8).(x > 3).X_3'$$

$X_2 \quad \approx (y > 3).X_4 + !(y > 3).X_5$

$X_4 \quad \approx p(x).X_8$

$$X_5 \approx 0$$

$$X_8 = \sum_{\sigma_\partial \in \Gamma_{X_9}} (\alpha_{80} > 3)\sigma_\partial.[s(y).s(x).P \sqcap_e (s(\alpha_{80}).Q)\sigma_\partial]$$

$$\approx (y > 3).[s(y).s(x).P \sqcap_e s(y).Q\sigma] \approx (y > 3).X_9'$$

- Step 4: Since there are new intersections X_3', X_9', we can generate new equations in E.

$$\begin{aligned}
X_3' &\approx [(y > 5).p(x).s(y)+!(y > 5).1].s(x).P \sqcap_e s(x).Q\sigma \\
&\approx (y > 5).[p(x).s(y).s(x).P \sqcap_e s(x).Q\sigma]+!(y > 5).[s(x).P \sqcap_e s(x).Q\sigma] \\
&\approx (y > 5).X_6'+!(y > 5).X_7' \\
X_6' &= 0 \\
X_7' &= s(x).\big(P \sqcap_e Q\big) = s(x).X_1 \\
X_9' &= s(y).[s(x).P \sqcap_e Q\sigma] = s(y).X_{10}' \\
X_{10}' &= s(x).P \sqcap_e Q\sigma = s(x).P \sqcap_e ((\alpha_{100} > 3).s(\alpha_{100}).Q)\sigma \\
&= s(x).P \sqcap_e (x > 3).s(x).Q\sigma = (x > 3).[s(x).P \sqcap_e s(x).Q\sigma] \\
&= (x > 3).X_{11}' \\
X_{11}' &= s(x).X_1
\end{aligned}$$

- Step 5: Solve the equation set E by using Arden's Lemma and variable substitutions:

$$\begin{aligned}
X_{10}' &\approx (x > 3).X_{11}' \approx (x > 3).s(x).X_1 \\
X_9' &\approx s(y).X_{10}' \approx s(y).(x > 3).s(x).X_1 \\
X_8 &\approx (y > 3).X_9' \approx (y > 3).s(y).(x > 3).s(x).X_1
\end{aligned}$$

$$\begin{aligned}
X_2 &\approx (y > 5).X_4+!(y > 5).X_5 \approx (y > 5).X_4 + 0 \approx (y > 5).p(x)X_8 \\
&\approx (y > 5).p(x).(y > 3).s(y).(x > 3).s(x).X_1
\end{aligned}$$

$$\begin{aligned}
X_3' &\approx (y > 5).X_6'+!(y > 5).X_7' \approx 0+!(y > 5).X_7' \\
&\approx !(y > 5).s(x).X_1
\end{aligned}$$

$$\begin{aligned}
X_1 &\approx (x < 8).X_2 + (x < 8).(x > 3).X_3' \\
&\approx (x < 8).(y > 5).p(x).(y > 3).s(y).(x > 3).s(x).X_1 + (x < 8).(x > 3).!(y > 5).s(x).X_1 \\
&\approx \{(x < 8).(y > 5).p(x).(y > 3).s(y).(x > 3).s(x). + (x < 8).(x > 3).!(y > 5).s(x)\}^*0 \\
&\approx \{(x < 8).[(y > 5)\big(p(x)(y > 3)s(y)(x > 3)s(x)\big)+!(y > 5).\big((x > 3).s(x)\big)]\}^*\{!(x < 8).0\} \\
&\approx \{(x < 8).[(y > 5).\big(p(x).[(y > 3).s(y)+!(y > 3).0]. \\
&\quad [(x > 3).s(x)+!(x > 3).0]\big)+!(y > 5).\big((x > 3).s(x)+!(x > 3).0\big)]\}^*\{!(x < 8).0\}
\end{aligned}$$

- Result:

```
while (x<8)
  {
    if( y>5 )
    {
       print(x);
       if(y>3) send(y); else exit();
       if(x>3) send(x); else exit();
    }
    else
    { if(x>3) send(x); else exit(); }
  }
```

5.6.2 More Complicated Policies

Suppose that a is an action and Q is a security policy, we can use the following $\mathcal{V}LTL$ formula to specify where the policy Q should be enforced in a program according to the position of a.

$\neg aUa.Q$	:	Q holds after action a
$QUa.\top$	:	Q holds before action a
$\neg aUa.QUb.\top$	:	Q holds between action a and b

5.6.3 Example with More Complicated Policies

Now we present an example to enforce a more complex security property.

Suppose that we have $e \in \mathcal{E}$, $F_v(e) = \{\alpha, \beta\}$ and $N_v(e) = \{x, y\}$. Let P be the following program:

```
send(x);
while(x>6)
  {
      link (x);  send (y);
  }
```

Let s and l denote the actions $send$ and $link$ respectively. By using the transformation function introduced in 5.2.5, the program P can be specified in $\mathcal{E}BPA^*_{0,1}$ by the following process:

$P = s(x).\{(x > 6).[l(x).s(y)]\}^*0$

Suppose that we have a security policy Q stating that after a link action, we can send only values that are smaller than 7. Q can be specified in $\mathcal{V}LTL$ as following:

$$Q = [\neg l(\alpha)]Ul(\alpha).\{[(\neg s(\alpha)) \vee [\lambda\beta((\beta < 7).s(\beta))]]U\bot\}$$

Q can be transformed to $\mathcal{E}BPA^*_{0,1}$ as following:

$$Q = [s(x) + s(y)]^* l(\alpha).\{[l(x) + \lambda\beta\big((\beta < 7).s(\beta)\big)]^* 0\}$$

Now, we can compute $P \sqcap_e Q$, where $G = \phi$, $E = \phi$ and $S = \{(P, Q)\}$

- Inputs:
  $P = s(x).\{(x > 6).[l(x).s(y)]\}^* 0 = s(x).P_1$

  $Q = [s(x) + s(y)]^* l(\alpha).\{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}$

- Step 1: Generates E and G:

$\underline{X_1}$ $= \underline{P \sqcap_e Q} \approx s(x).(\underline{P_1 \sqcap_e Q}) = s(x).\underline{X_2}$ (e_1)

Here: $\underline{\Gamma_{X_1}} = \underline{\Gamma_{X_2}}$ (g_1)

$\underline{X_2}$ $= \underline{P_1 \sqcap_e Q}$

$\approx (x > 6).\Big(\underline{l(x).s(y).P_1 \sqcap_e l(\alpha).\{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}}\Big)$

$+(x > 6).\big(\underline{l(x).s(y).P_1 \sqcap_e s(x).Q}\big) + (x > 6).\big(\underline{l(x).s(y).P_1 \sqcap_e s(y).Q}\big)$

$= (x > 6).\underline{X_3} + (x > 6).\underline{X_4} + (x > 6).\underline{X_5}$ (e_2)

Here: $\underline{\Gamma_{X_2}} = \underline{\Gamma_{X_3}} \cup \underline{\Gamma_{X_4}} \cup \underline{\Gamma_{X_5}}$ (g_2)

$\underline{X_3}$ $= \underline{l(x).s(y).P_1 \sqcap_e l(\alpha).\{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}}$

$\approx l(x).\big(\underline{s(y).P_1 \sqcap_e \{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}}\big)$

$= l(x).\underline{X_6}$ (e_3)

Here: $\underline{\Gamma_{X_3}} = \{\{\alpha \mapsto x\}\} \circ \underline{\Gamma_{X_6}}$ (g_3)

$\underline{X_4}$ $= 0$ (e_4)

Here: $\underline{\Gamma_{X_4}} = \emptyset$ (g_4)

$\underline{X_5}$ $= 0$ (e_5)

Here: $\underline{\Gamma_{X_5}} = \emptyset$ (g_5)

Let $Q' = [l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0$

$$\underline{X_6} \quad = \underline{s(y).P_1 \sqcap_e \{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}}$$

$$= \sum_{\sigma_\partial \in \underline{\Gamma_{X_6}}} [(\beta_{60} < 7)]\sigma_\partial.[\underline{s(y).P_1 \sqcap_e (s(\beta_{60}).Q')\sigma_\partial}] \quad (e_6)$$

Here: $\underline{\Gamma_{X_6}} = \underline{\Gamma_{X_7}}$ $\qquad (g_6)$

$$\underline{X_7} \quad = \underline{s(y).P_1 \sqcap_e s(\beta_{60}).Q'} \approx s(y).(\underline{P_1 \sqcap_e Q'}) \quad (e_7)$$

Here: $\underline{\Gamma_{X_7}} = \{\{\beta_{60} \mapsto y\}\} \circ \underline{\Gamma_{X_8}}$ $\qquad (g_7)$

$$\begin{aligned}\underline{X_8} \quad &= \underline{P_1 \sqcap_e Q'} \\ &\approx (x > 6).\left(\underline{l(x).s(y).P_1 \sqcap_e l(x).Q'}\right) \\ &+ \sum_{\sigma_\partial \in \underline{\Gamma_{X_{10}}}} [(x > 6).(\beta_{81} < 7)]\sigma_\partial.[\underline{l(x).s(y).P_1 \sqcap_e (s(\beta_{81}).Q')\sigma_\partial}] \qquad (e_8)\end{aligned}$$

Here: $\underline{\Gamma_{X_8}} = \underline{\Gamma_{X_9}} \cup \underline{\Gamma_{X_{10}}}$ $\qquad (g_8)$

$$\underline{X_9} \quad = \underline{l(x).s(y).P_1 \sqcap_e l(x).Q'} \approx l(x).\left(\underline{s(y).P_1 \sqcap_e Q'}\right) \approx l(x).\underline{X_6} \qquad (e_9)$$

Here: $\underline{\Gamma_{X_9}} = \underline{\Gamma_{X_6}}$ $\qquad (g_9)$

$$\underline{X_{10}} \quad = 0 \qquad (e_{10})$$

Here: $\underline{\Gamma_{X_{10}}} = \emptyset$ $\qquad (g_{10})$

We have $E = \{e_1, ..., e_{10}\}, G = \{g_1, ..., g_{10}\}$.

- Step 2: Solve the equation set G to get all the values of Γ.

Here:

$\underline{\Gamma_{X_8}} = \underline{\Gamma_{X_9}} \cup \underline{\Gamma_{X_{10}}} = \underline{\Gamma_{X_9}} = \underline{\Gamma_{X_6}}$

$\underline{\Gamma_{X_7}} = \{\{\beta_{60} \mapsto y\}\} \circ \underline{\Gamma_{X_8}} = \{\{\beta_{60} \mapsto y\}\} \circ \underline{\Gamma_{X_6}}$

$\underline{\Gamma_{X_6}} = \underline{\Gamma_{X_7}} = \{\{\beta_{60} \mapsto y\}\} \circ \underline{\Gamma_{X_6}}$

$\underline{\Gamma_{X_6}} = \{\{\beta_{60} \mapsto y\}\} = \{\sigma\}$

$\underline{\Gamma_{X_2}} = \underline{\Gamma_{X_3}} == \{\{\alpha \mapsto x, \beta_{60} \mapsto y\}\}$

- Step 3: Now, we can substitute $\underline{\Gamma}$ in E.

$$\begin{aligned}\underline{X_1} \quad &= s(x).\underline{X_2} \\ \underline{X_2} \quad &\approx (x > 6).\left(\underline{l(x).s(y).P_1 \sqcap_e l(\alpha).\{[l(x) + \lambda\beta((\beta < 7).s(\beta))]^* 0\}}\right) \\ &+ (x > 6).\left(\underline{l(x).s(y).P_1 \sqcap_e s(x).Q}\right) + (x > 6).\left(\underline{l(x).s(y).P_1 \sqcap_e s(y).Q}\right) \\ &= (x > 6).\underline{X_3} + (x > 6).\underline{X_4} + (x > 6).\underline{X_5} \\ \underline{X_3} \quad &= l(x).\underline{X_6}\end{aligned}$$

$$\underline{X_4} \quad = 0$$
$$\underline{X_5} \quad = 0$$

$$Q' = [l(x) + \lambda\beta((\beta < 7).s(\beta))]^*0$$

$$\underline{X_6} \quad = \sum_{\sigma_\partial \in \underline{\Gamma_{X_6}}} [(\beta_{60} < 7)]\sigma_\partial.[\underline{s(y).P_1 \sqcap_e (s(\beta_{60}).Q')\sigma_\partial}] \approx (y < 7).[\underline{s(y).P_1 \sqcap_e s(y).Q'\sigma}]$$
$$\approx (y < 7).\underline{X_7'}$$

- Step 4: Since there is a new intersection $\underline{X_7'}$, we can generate new equations in E.

$$\begin{aligned}
\underline{X_7'} &\approx s(y).[\underline{P_1 \sqcap_e Q'\sigma}] = s(y).\underline{X_8'}\\
\underline{X_8'} &\approx \underline{P_1 \sqcap_e Q'\sigma} \approx (x > 6).l(x).s(y).P_1 \sqcap_e l(x).Q'\sigma\\
&+(x > 6).l(x).s(y).P_1 \sqcap_e ((\beta_{81} < 7)s(\beta_{81}).Q')\sigma\\
&\approx (x > 6).[\underline{l(x).s(y).P_1 \sqcap_e l(x).Q'\sigma}] + 0\\
&\approx (x > 6).\underline{X_9'}\\
\underline{X_9'} &\approx \underline{l(x).s(y).P_1 \sqcap_e l(x).Q'\sigma} \approx l(x).[\underline{s(y).P_1 \sqcap_e Q'\sigma}] = l(x).\underline{X_6}
\end{aligned}$$

- Step 5: Solve the equation set E by using Arden's Lemma and variable elimination:

$$\begin{aligned}
\underline{X_2} &\approx (x > 6).\underline{X_3} + (x > 6).\underline{X_4} + (x > 6).\underline{X_5}\\
&\approx (x > 6).\underline{X_3} + (x > 6).0 + (x > 6).0\\
&\approx (x > 6).\underline{X_3} \approx (x > 6).l(x).\underline{X_6}
\end{aligned}$$

$$\underline{X_1} \approx s(x).\underline{X_2} \approx s(x).(x > 6).l(x).\underline{X_6}$$

$$\begin{aligned}
\underline{X_6} &\approx (y < 7).\underline{X_7'}\\
&\approx (y < 7).s(y).\underline{X_8'}\\
&\approx (y < 7).s(y).(x > 6)\underline{X_9'}\\
&\approx (y < 7).s(y).(x > 6).l(x).\underline{X_6}
\end{aligned}$$

$$\underline{X_6} \approx \{(y < 7).s(x).(x > 6).l(x)\}^*0$$

$$\begin{aligned}
\underline{X_1} &\approx s(x).(x > 6).l(x).\{(y < 7).s(y).(x > 6).l(x)\}^*0\\
&\approx s(x).[(x > 6).l(x) + !(x > 6).0].\{(y < 7).s(y).[(x > 6).l(x) + !(x > 6).0]\}^*\{!(y < 7).0\}
\end{aligned}$$

- Result:

```
send(x);
```

```
if (x >6) link(x);  else exit();
while (y <7)
{
   send(y);
   if(x>6); link(x);  else exit();
}
```

5.7 Proof of the Main Result

5.7.1 MGU Under Constraints

To define the intersection of two terms, we need the most general unifier. Two terms a and b over Σ^* are unifiable, if there exists a substitution $\sigma : \mathcal{X} \rightarrow \Sigma^*$ such that $a\sigma = b\sigma$, where $a\sigma$ denotes $\sigma(a)$. The most general unifier between a and b is denoted by $mgu(a, b)$. We also say that a substitution $\sigma = \{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ is a good mapping if and only if t_i are terms and x_i are distinct variables. In the rest of this paper we will use Γ to denote the set of all substitutions.

Since we need to consider mgu in a given environment e and separate all variables into free and non-free ones, the classic mgu algorithm may generate results that we do not want. For example: Let $g(x, f(x))$ and $g(\alpha, \beta)$ be two terms, where x is non-free variable and α, β are free variables. By using the previous algorithm, we generate two substitutions: $\sigma_1 = \{\alpha \mapsto x, \beta \mapsto f(x)\}$ or $\sigma_2 = \{x \mapsto \alpha, \beta \mapsto f(\alpha)\}$. But σ_1 is acceptable while σ_2 is not, because σ_2 substitute a non free variable by a free one. We introduce a new algorithm for finding mgu(a_1, a_2) in $e \in \mathcal{E}$, which is denoted by $mgu(a_1, a_2, e)$. This algorithm is inspired by Martelli-Montanari reduction rules [45].

Algorithm For $a_1, a_2 \in \mathcal{A}$ and $e \in \mathcal{E}$, let $\mathcal{V} = \mathcal{V}(a_1) \cup \mathcal{V}(a_2)$, then we compute the $mgu(a_1, a_2, e)$ as following.

1. $E = \{a_1 = a_2\}$
2. Apply following rewriting rules on E, until we reach a normal form.
   - Decompose:
     $E \cup \{f(s_1, ..., s_n) = f(t_1, ..., t_n)\} \longrightarrow E \cup \{s_1 = t_1, ..., s_n = t_n\}$
   - Delete: $E \cup \{t = t\} \longrightarrow E$
   - Substitute Sorted Variables (SSV):

     $E \cup \{x = t\} \longrightarrow \{x = t\} \cup E\{x \mapsto t\}$, if $x \in \mathcal{V}$, $t \in \mathcal{V}$ and $s_e(t) \subseteq s_e(x)$.

- Variable Term Substitute (VTS):

  $E \cup \{x = t\} \longrightarrow \{x = t\} \cup E\{x \mapsto t\}$, if $x \in \mathcal{V}, t \notin \mathcal{V}$ and $x \notin \mathcal{V}(t)$

- Reverse:

  $E \cup \{t = x\} \longrightarrow \{x = t\} \cup E$, if:

  - $x \in \mathcal{V}, t \notin \mathcal{V}$ OR
  - $x \in \mathcal{V}, t \in \mathcal{V}, s_e(t) \subseteq s_e(x)$

- Occur Check (OC):

  $E \cup \{x = t\} \longrightarrow \perp$ (fail), if $x \in \mathcal{V}$, $x \in \mathcal{V}(t)$.

- Clash:

  $E \cup \{f(s_1, ..., s_n) = g(t_1, ..., t_n)\} \longrightarrow \perp$. if f and g are different.

Here, we introduce two examples using the above algorithm.

Example 5.7.1 *Let* $e \in \mathcal{E}$*,* $F_v(e) = \{\alpha, \beta\}$ *and* $N_v(e) = \{x, y, z\}$*. We calculate the* mgu *of* $a_1 = f(h(x), g(y, z))$ *and* $a_2 = f(\alpha, g(3, \beta))$ *in the environment* e *using the above algorithm.*

$$
\begin{array}{ll}
 & E = \{f(h(x), g(y, z)) = f(\alpha, g(3, \beta))\} \\
=> & \qquad \{\!|\text{Decompose }|\!\} \\
 & \{h(x) = \alpha, g(y, z) = g(3, \beta))\} \\
=> & \qquad \{\!|\text{Decompose }|\!\} \\
 & \{h(x) = \alpha, y = 3, z = \beta)\} \\
=> & \qquad \{\!|\text{Reverse }|\!\} \\
 & \{\alpha = h(x), y = 3, \beta = z)\} \\
=> & \qquad \{\!|\text{VTS }|\!\} \\
 & \{\underline{\alpha = h(x)}, y = 3, \beta = z)\} \\
=> & \qquad \{\!|\text{VTS }|\!\} \\
 & \{\underline{\alpha = h(x)}, \underline{y = 3}, \beta = z)\} \\
=> & \qquad \{\!|\text{SSV }|\!\} \\
 & \{\underline{\alpha = h(x)}, \underline{y = 3}, \underline{\beta = z})\}
\end{array}
$$

Conclusion: returns $\sigma = \{\alpha \mapsto h(x), y \mapsto 3, \beta \mapsto z\}$

Example 5.7.2 *Let $e \in \mathcal{E}$, $F_v(e) = \{\alpha\}$ and $N_v(e) = \{x, y\}$. We calculate the mgu of $a_1 = f(x, y, g(h(x)))$ and $a_2 = f(4, \alpha, g(\alpha))$ in e using the above algorithm.*

$$
\begin{array}{ll}
 & E = \{f(x, y, g(h(x))) = f(4, \alpha, g(\alpha))\} \\
=> & \quad \{\!|\text{Decompose}|\!\} \\
 & \{x = 4, y = \alpha, g(h(x)) = g(\alpha)\} \\
=> & \quad \{\!|\text{Decompose}|\!\} \\
 & \{x = 4, y = \alpha, h(x) = \alpha\} \\
=> & \quad \{\!|\text{Reverse}|\!\} \\
 & \{x = 4, \alpha = y, \alpha = h(x)\} \\
=> & \quad \{\!|\text{VTS}|\!\} \\
 & \{\underline{x = 4}, \alpha = y, \alpha = h(4)\} \\
=> & \quad \{\!|\text{SSV}|\!\} \\
 & \{\underline{x = 4}, \underline{\alpha = y}, y = h(4)\} \\
=> & \quad \{\!|\text{VTS}|\!\} \\
 & \{\underline{x = 4}, \underline{\alpha = y}, \underline{y = h(4)}\}
\end{array}
$$

Conclusion: returns $\sigma = \{x \mapsto 4, \alpha \mapsto y, y \mapsto h(4)\}$

5.7.2 Proof of Theorem 5.4.17

Lemma 5.7.3 *For two processes P and Q, $e \in \mathcal{E}$, we have:*

$P \sqsubseteq_e Q \Longrightarrow [P]_e \sqsubseteq [Q]_e$

Proof:

To prove $[P]_e \sqsubseteq [Q]_e$, we need to prove that:

$\forall \tau_e \in \Sigma^*$, if $[P]_e{\downarrow}\tau_e$ then $\exists \tau'_e \in \Sigma^*$, such that:$[Q]_e{\downarrow}\tau'_e$ and $\tau_e \simeq \tau'_e$.

Let $\tau_e \in \Sigma^*$, and $[P]_e{\downarrow}\tau_e$

$\Rightarrow$ $\quad \{\!|\text{Definition 5.4.1}|\!\}$

$\exists [R_{a_n}]_{e_n} \in \mathcal{P}$ such that: $[P]_e \overset{\tau_e}{\twoheadrightarrow} [R_{a_n}]_{e_n}$

$\Rightarrow$ $\{\!|$Based on Rules $(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2, when a process runs in an environment, it generates closed traces (no variables) without conditions.

Let $\tau_e = a_1(v_{a1}^1, ..., v_{a1}^n)...a_n(v_{an}^1, ..., v_{an}^n)$ $(v_{ai}^i...v_{ai}^n) \in (s_i(x_1)...s_i(x_n))$ for $1 \leq i \leq n$ and $s_1 = eff_{a_1}(e), s_2 = eff_{a_2}(s_1)$ $...s_n = eff_{a_n}(s_{n-1})$ $|\!\}$

$$[P]_e \overset{a_1(v_{a1}^1,...,v_{a1}^n)}{\longrightarrow} [P_1]_{e1} ... \overset{a_n(v_{an}^1,...,v_{an}^n)}{\longrightarrow} [P_n]_{e_n}$$

$\Rightarrow$ $\{\!|(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2 $|\!\}$

For each step $\left([P_i]_{e_i} \overset{a_i(v_{ai}^1,...,v_{ai}^n)}{\longrightarrow} [P_{i+1}]_{e(i+1)}\right)$ of $[P]_e$, there is a corresponding step for process P. Let $\tau = c_1(x_1...x_n).a_1(x_1...x_n)...c_n(x_1...x_n).a_n(x_1...x_n)$, where each $c_i(x_1...x_n) \in \mathcal{B}$, we have:

$$\frac{P \overset{c_1(x_1...x_n)}{\longrightarrow} P_{c_1} \quad [c_1(x_1...x_n)]_e\downarrow \quad \dfrac{P_{c_1} \overset{a_1(x_1...x_n)}{\longrightarrow} P_1}{[P_{c_1}]_e \overset{a_1(v_{a1}^1...v_{a1}^n)}{\longrightarrow} [P_1]_{e_1}}}{[P]_e \overset{a_1(v_{a1}^1...v_{a1}^n)}{\longrightarrow} [P_1]_{e_1}}$$

...

$$\frac{P_{n-1} \overset{c_n(x_1...x_n)}{\longrightarrow} P_{c_n} \quad [c_n(x_1...x_n)]_{e_{n-1}}\downarrow \quad \dfrac{P_{c_n} \overset{a_n(x_1...x_n)}{\longrightarrow} P_n}{[P_{c_n}]_{e_{n-1}} \overset{a_n(v_{a1}^1...v_{a1}^n)}{\longrightarrow} [P_n]_{e_n}}}{[P_{n-1}]_{e_{n-1}} \overset{a_n(v_{a1}^1...v_{a1}^n)}{\longrightarrow} [P_n]_{e_n}} \qquad (\alpha)$$

$\Rightarrow$ $\{\!|$Definition 5.4.1 $|\!\}$

$P \overset{\tau}{\twoheadrightarrow} P_n$, where $\tau = c_1(x_1...x_n).a_1(x_1...x_n)...c_n(x_1...x_n)$ $.a_n(x_1...x_n)$

$\Rightarrow$ $\{\!|$Definition 5.4.5 ($\sqsubseteq_e$) and $P \sqsubseteq_e Q$ $|\!\}$

$\exists \tau' \in \Sigma^*$ such that $Q\downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$

$\Rightarrow$ $\{\!|$Definition 5.4.3 $|\!\}$

There are different possibilities for τ':

(1) From the reflexivity of Definition 5.4.3, we have $\tau \sqsubseteq_e \tau'$, when $\tau = \tau'$. So we have:

$\tau' = \tau = c_1(x_1...x_n).a_1(x_1...x_n)...c_n(x_1...x_n).a_n(x_1...x_n)$

$Q\downarrow\tau'$

$\Rightarrow$ $\{\!|$Definition 5.4.1 $|\!\}$

$\exists Q_n \in \mathcal{P}$ such that: $Q \overset{\tau'}{\twoheadrightarrow} Q_n$

$\Rightarrow$ $\{\!|(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2 $|\!\}$

Now, we can find a trace for $[Q]_e$, for $a_1(v_{a1}^1, ..., v_{a1}^n)...a_n(v_{an}^1, ..., v_{an}^n)$, we still have $s_1 = eff_{a_1}(e)$, $...s_n = eff_{a_n}(s_{n-1})$ and $(v_{ai}^i...v_{ai}^n) \in (s_i(x_1)...s_i(x_n))$ for $1 \leq i \leq n$

$$\frac{Q \xrightarrow{c_1(x_1...x_n)} Q_{c_1} \quad [c_1(x_1...x_n)]_e\!\downarrow \quad \dfrac{Q_{c_1} \xrightarrow{a_1(x_1...x_n)} Q_1}{[Q_{c_1}]_e \xrightarrow{a_1(v^1_{a1}...v^n_{a1})} [Q_1]_{e_1}}}{[Q]_e \xrightarrow{a_1(v^1_{a1}...v^n_{a1})} [Q_1]_{e_1}}$$

...

$$\frac{Q_{n-1} \xrightarrow{c_n(x_1...x_n)} Q_{c_n} \quad [c_n(x_1...x_n)]_{e_{n-1}}\!\downarrow \quad \dfrac{Q_{c_n} \xrightarrow{a_n(x_1...x_n)} Q_n}{[Q_{c_n}]_{e_{n-1}} \xrightarrow{a_n(v^1_{a1}...v^n_{a1})} [Q_n]_{e_n}}}{[Q_{n-1}]_{e_{n-1}} \xrightarrow{a_n(v^1_{a1}...v^n_{a1})} [Q_n]_{e_n}}$$

$\Rightarrow$ $\{\!|$Definition 5.4.1 and let $\tau'_e = a_1(v^1_{a1}...v^n_{a1})...a_n(v^1_{an}...v^n_{an})$ $|\!\}$

$[Q]_e \overset{\tau'_e}{\twoheadrightarrow} [Q_n]_{e_n}$

$\Rightarrow$ $\{\!|\, [P]_e \downarrow \tau_e \,|\!\}$

$\forall \tau_e \in \Sigma^*$, if $[P]_e \downarrow \tau_e$, then $\exists \tau'_e \in \Sigma^*$, such that $[Q]_e \downarrow \tau'_e$ and $\tau_e \simeq \tau'_e$

$\Rightarrow$ $\{\!|$Definition 5.4.7 $|\!\}$

$[P]_e \sqsubseteq [Q]_e$

(2) From Definition 5.4.3, we can get a trace $\tau', \tau \sqsubseteq_e \tau'$, by removing all the conditions from τ. Here let $\tau' = \hat{\tau} = a_1(x_1...x_n).a_2(x_1...x_n)....a_n(x_1...x_n)$. This will not restrict the result.

$Q \downarrow \tau'$

$\Rightarrow$ $\{\!|$Definition 5.4.1 $|\!\}$

$\exists Q_n \in \mathcal{P}$ such that: $Q \overset{\tau'}{\twoheadrightarrow} Q_n$

$\Rightarrow$ $\{\!|\, (R^{[]_e}_a)$ and $(R^{[]_e}_c)$ in Table 5.2 $|\!\}$

$$\frac{Q \xrightarrow{a_1(x_1,...,x_n)} Q_{a_1}}{[Q]_e \xrightarrow{a_1(v^1_{a1},...,v^n_{a1})} [Q_{a_1}]_{e_1}} \quad \frac{Q_{a_1} \xrightarrow{a_2(x_1,...,x_n)} Q_{a_2}}{[Q_{a_1}]_{e_1} \xrightarrow{a_2(v^1_{a2},...,v^n_{a2})} [Q_{a_2}]_{e_2}} \;...\; \frac{Q_{a(n-1)} \xrightarrow{a_n(x_1,...,x_n)} Q_n}{[Q_{a(n-1)}]_{e_{n-1}} \xrightarrow{a_n(v^1_{an},...,v^n_{an})} [Q_n]_{e_n}}$$

$\Rightarrow$ $\{\!|$Definition 5.4.1 and let $\tau'_e = a_1(v^1_{a1}...v^n_{a1})...\ a_n(v^1_{an}...v^n_{an})$ $|\!\}$

$[Q]_e \overset{\tau'_e}{\twoheadrightarrow} [Q_n]_{e_n}$

$\Rightarrow$ $\{\!|\, [P]_e \downarrow \tau_e \,|\!\}$

$\forall \tau_e \in \Sigma^*$, if $[P]_e \downarrow \tau_e$, then $\exists \tau'_e \in \Sigma^*$, such that $[Q]_e \downarrow \tau'_e$ and $\tau_e \simeq \tau'_e$

$\Rightarrow$ $\{\!|$Definition 5.4.7 $|\!\}$

$[P]_e \sqsubseteq [Q]_e$

(3) From Definition 5.4.3, there may exist τ', such that $\tau \sqsubseteq_e \tau'$, and $\tau = \tau' \sigma^e_{\approx}$, for a sorted substitution $\sigma^e_{\approx} = \{y_1 \mapsto x_1, ..., y_n \mapsto x_n\}$. Then:

$\tau' = c_1(y_1...y_n).a_1(y_1...y_n)...c_n(y_1...y_n).a_n(y_1...y_n)$

$Q{\downarrow}\tau'$

$\Rightarrow$ {|Definition 5.4.1 |}

$\exists Q_n \in \mathcal{P}$ such that: $Q \overset{\tau'}{\twoheadrightarrow} Q_n$

$\Rightarrow$ {|$(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2 |}

When $\tau' = c_1(y_1...y_n).a_1(y_1...y_n)...c_n(y_1...y_n).\ a_n(y_1...y_n)$, since the domain of y_i is the top level domain (see Definition 5.2.6) and $\tau = \tau'\sigma^e_{\approx}$, the variables in $c_i(y_1...y_n)$ and $a_i(y_1...y_n)$ can take the same value with $c_i(x_1...x_n)$ and $a_i(x_1...x_n)$ in s.

$$\frac{Q \xrightarrow{c_1(y_1...y_n)} Q_{c_1} \quad [c_1(y_1...y_n)]_e{\downarrow} \quad \dfrac{Q_{c_1} \xrightarrow{a_1(y_1...y_n)} Q_1}{[Q_{c_1}]_e \xrightarrow{a_1(v^1_{a1}...v^n_{a1})} [Q_1]_{e_1}}}{[Q]_e \xrightarrow{a_1(v^1_{a1}...v^n_{a1})} [Q_1]_{e_1}} \ ...$$

$$\frac{Q_{n-1} \xrightarrow{c_n(y_1...y_n)} Q_{c_n} \quad [c_n(y_1...y_n)]_{e_{n-1}}{\downarrow} \quad \dfrac{Q_{c_n} \xrightarrow{a_n(y_1...y_n)} Q_n}{[Q_{c_n}]_{e_{n-1}} \xrightarrow{a_n(v^1_{a1}...v^n_{a1})} [Q_n]_{e_n}}}{[Q_{n-1}]_{e_{n-1}} \xrightarrow{a_n(v^1_{a1}...v^n_{a1})} [Q_n]_{e_n}}$$

$\Rightarrow$ {|Definition 5.4.1 and let $\tau'_e = a_1(v^1_{a1}...v^n_{a1})...a_n(v^1_{an}...v^n_{an})$ |}

$[Q]_e \overset{\tau'_e}{\twoheadrightarrow} [Q_n]_{e_n}$

$\Rightarrow$ {|$[P]_e{\downarrow}\tau_e$ |}

$\forall \tau_e \in \Sigma^*$, if $[P]_e{\downarrow}\tau_e$, then $\exists \tau'_e \in \Sigma^*$, such that $[Q]_e{\downarrow}\tau'_e$ and $\tau_e \simeq \tau'_e$

$\Rightarrow$ {|Definition 5.4.7 |}

$[P]_e \sqsubseteq [Q]_e$

(4) From Definition 5.4.3, there may exists τ', such that $\tau \sqsubseteq_e \tau'$, and τ' has different actions from τ. So we randomly pick up an action $a_k(x_1...x_n)$ in τ and we suppose that it is the only changed action in τ', (this will not restrict the result).

Suppose that α is a variable or a constant and $c_k.a_k = c'_k(x_1...x_n)\ .(x_m == \alpha).a_k(x_1...x_m...x_n)$ in τ, then $\tau' = c_1(x_1...x_n).a_1(x_1...x_n)...c'_k(x_1...x_n).a_k(x_1...x_{m-1}, \alpha, x_{m+1}...x_n)...c_n(x_1...x_n).a_n(x_1...x_n)$

$Q{\downarrow}\tau'$

$\Rightarrow$ {|Definition 5.4.1 |}

$\exists Q_n \in \mathcal{P}$ such that: $Q \overset{\tau'}{\twoheadrightarrow} Q_n$

$\Rightarrow$ {|$(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2 |}

From (α), we know that if $a_k(x_1...x_m...x_n)$ is executed in environment s_{k-1}, it will be

$a_k(v_{ak}^1 ... v_{ak}^m ... v_{ak}^n)$, so a_k's x_m in environment s_{k-1} is v_{ak}^m. Since $[c_k]_{e_{k-1}}\downarrow$ and $c_k = c'_k.(x_m == \alpha)$, we have $[c'_k.(x_m == \alpha)]_{e_{k-1}}\downarrow$, which means that the value of α is v_{ak}^m in environment s_{k-1}.

$\tau = c_1(x_1 ... x_n).a_1(x_1 ... x_n)...c'_k(x_1 ... x_n).(x == \alpha).\ a_k(x_1 ... x_{m-1}, x_m, x_{m+1} ... x_n)...a_n(x_1 ... x_n)$, and $\tau' = c_1(x_1 ... x_n).a_1(x_1 ... x_n)...c'_k(x_1 ... x_n).a_k(x_1 ... x_{m-1}, \alpha, x_{m+1} ... x_n)...a_n(x_1 ... x_n)$.

$$\frac{Q \xrightarrow{c_1(x_1 ... x_n)} Q_{c_1} \quad [c_1(x_1 ... x_n)]_e\downarrow \quad \dfrac{Q_{c_1} \xrightarrow{a_1(x_1 ... x_n)} Q_1}{[Q_{c_1}]_e \xrightarrow{a_1(v_{a1}^1 ... v_{a1}^n)} [Q_1]_{e_1}}}{[Q]_e \xrightarrow{a_1(v_{a1}^1 ... v_{a1}^n)} [Q_1]_{e_1}} \quad ...$$

$$,\ \frac{\left(Q_{k-1} \xrightarrow{c'_k(x_1 ... x_n)} Q_{ck'}, \quad [c'_k(x_1 ... x_n)]_{e_{k-1}}\downarrow, \quad \dfrac{Q_{ck'} \xrightarrow{a_k(x_1 ... x_{m-1}, \alpha, x_{m+1} ... x_n)} Q_k}{[Q_{ck'}]_{e_{k-1}} \xrightarrow{a_k(v_{ak}^1 ... v_{ak}^m ... v_{ak}^n)} [Q_k]_{e_k}}\right)}{[Q_{k-1}]_{e_{k-1}} \xrightarrow{a_k(v_{ak}^1 ... v_{ak}^m ... v_{ak}^n)} [Q_k]_{e_k}} \quad ...$$

$$\frac{Q_{n-1} \xrightarrow{c_n(x_1 ... x_n)} Q_{c_n} \quad [c_n(x_1 ... x_n)]_{e_{n-1}}\downarrow \quad \dfrac{Q_{c_n} \xrightarrow{a_n(x_1 ... x_n)} Q_n}{[Q_{c_n}]_{e_{n-1}} \xrightarrow{a_n(v_{a1}^1 ... v_{a1}^n)} [Q_n]_{e_n}}}{[Q_{n-1}]_{e_{n-1}} \xrightarrow{a_n(v_{a1}^1 ... v_{a1}^n)} [Q_n]_{e_n}}$$

$\Rightarrow$ {|Definition 5.4.1 |}

when the value of α is v_{ak}^m in environment s_{k-1}. We have $\tau'_e = a_1(v_{a1}^1, ..., v_{a1}^n)...\ a_k(v_{ak}^1 ... v_{ak}^m ... v_{ak}^n)$ $...a_n(v_{an}^1 ... v_{an}^n)$. such that: $[Q]_e \overset{\tau'_e}{\twoheadrightarrow} [Q_n]_{e_{qn}}$

$\Rightarrow$ {|$[P]_e\downarrow\tau_e, \tau_e = \tau'_e$ |}

$\forall \tau_e \in \Sigma^*$, if $[P]_e\downarrow\tau_e$, then $\exists \tau'_e \in \Sigma^*$, such that $[Q]_e\downarrow\tau'_e$ and $\tau_e \simeq \tau'_e$

$\Rightarrow$ {|Definition 5.4.7 |}

$[P]_e \sqsubseteq [Q]_e$

$(*)$ Based on the transitivity of $\sqsubseteq_e$, we can always find τ'_e, $\tau_e \simeq \tau'_e$, such that: $[Q]_e \overset{\tau'_e}{\twoheadrightarrow} [Q_n]_{e_n}$

$\Rightarrow$ {|$[P]_e\downarrow\tau_e, \tau_e = \tau'_e$ |}

$\forall \tau_e \in \Sigma^*$, if $[P]_e\downarrow\tau_e$, then $\exists \tau'_e \in \Sigma^*$, such that $[Q]_e\downarrow\tau'_e$ and $\tau_e \simeq \tau'_e$

$\Rightarrow$ {|Definition 5.4.7 |}

$[P]_e \sqsubseteq [Q]_e$

□

Lemma 5.7.4 *For two processes P and Q, $e \in \mathcal{E}$, we have: $P \preceq Q \Longrightarrow [P]_e \sqsubseteq [Q]_e$*

Proof:

Since $\preceq \subseteq \sqsubseteq_e$, we have: $P \preceq Q \Longrightarrow P \sqsubseteq_e Q \Longrightarrow [P]_e \sqsubseteq [Q]_e$ □

The relation $\preceq$ has also an interesting property given hereafter.

Lemma 5.7.5 *In Σ^*, the relation $\preceq$ is congruent for the operator ".". i,e. for all τ, τ' and τ'' in Σ, we have:*

$\tau \preceq \tau' \Longrightarrow$

- $\tau.\tau'' \preceq \tau'.\tau''$
- $\tau''.\tau \preceq \tau''.\tau'$

Proof: Directly from Definition 5.4.2. □

Proposition 5.4.6: [Properties of $\preceq$]

The ordering $\preceq$ is congruent for the operators (" $+$ ", "." and " $*$ "), i,e: for any processes P, P' and Q, we have:

$P \preceq P' \Longrightarrow$

$$\begin{array}{rrcl rrcl}
(1) & P+Q & \preceq & P'+Q & (2) & Q+P & \preceq & Q+P' \\
(3) & P.Q & \preceq & P'.Q & (4) & Q.P & \preceq & Q.P' \\
(5) & P^*Q & \preceq & P'^*Q & (6) & Q^*P & \preceq & Q^*P'
\end{array}$$

Proof:

□

If $P \preceq P'$, then by Definition 5.4.4 ($\preceq$):

1. if $P{\downarrow}\tau$ then $\exists \tau' \in \Sigma^*$, such that:$P'{\downarrow}\tau'$ and $\tau \preceq \tau'$. (α)
2. if $P \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*$, such that:$P' \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. (β)

The proof is by structural induction when $P \preceq P'$.

- $P + Q \preceq P' + Q$

  if $(P + Q)\downarrow\tau$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

  $P\downarrow\tau$ or $Q\downarrow\tau$

$\Rightarrow$ $\{\!|(\alpha)|\!\}$

  $\exists\tau' \in \Sigma^*$, such that: $P'\downarrow\tau'$ and $\tau \preceq \tau'$ or $Q\downarrow\tau$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

  $\exists\tau' \in \Sigma^*$ such that: $(P' + Q)\downarrow\tau'$ and $\tau \preceq \tau'$ or $\exists\tau \in \Sigma^*$, such that: $(P' + Q)\downarrow\tau$ and $\tau \preceq \tau$ (1)

  if $P + Q \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

  $P \overset{\tau}{\twoheadrightarrow} 1$ or $Q \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)|\!\}$

  $\exists\tau' \in \Sigma^*$, such that: $P' \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$ or $Q \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

  $\exists\tau' \in \Sigma^*$, such that: $(P' + Q)\tau'\twoheadrightarrow 1$ and $\tau \preceq \tau'$ or $\exists\tau \in \Sigma^*$,
  such that: $(P' + Q)\tau\twoheadrightarrow 1$ and $\tau \preceq \tau$ (2)

From (1), (2) and Definition 5.4.4 ($\preceq$), we can say $P + Q \preceq P' + Q$.

- $Q + P \preceq Q + P'$
  Same with above.

- $P.Q \preceq P'.Q$

  if $P.Q\downarrow\tau$

$\Rightarrow$ $\{\!|(R_{\dot{l}}),\ (R_{\dot{r}})$ from Table 5.2, Definition 5.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$P{\downarrow}\tau$ or $P \overset{\tau_1}{\twoheadrightarrow} 1, Q{\downarrow}\tau_2$

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$\exists\tau' \in \Sigma^*$, such that: $P'{\downarrow}\tau'$ and $\tau \preceq \tau'$ or $P \overset{\tau_1}{\twoheadrightarrow} 1, Q{\downarrow}\tau_2$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$\exists\tau' \in \Sigma^*$ such that: $P'{\downarrow}\tau'$ and $\tau \preceq \tau'$ or $\exists\tau'_1 \in \Sigma^*$ such that: $P' \overset{\tau'_1}{\twoheadrightarrow} 1$ and $\tau_1 \preceq \tau'_1, Q{\downarrow}\tau_2$

$\Rightarrow$ $\{\!|(R_{\dot{l}}),\ (R_{\dot{r}})$ from Table 5.2 $|\!\}$

$P'.Q{\downarrow}\tau'$ or $P'.Q{\downarrow}(\tau'_1.\tau_2)$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5$|\!\}$

$P'.Q{\downarrow}\tau', \tau \preceq \tau'$ or $P'.Q{\downarrow}(\tau'_1.\tau_2), \tau = \tau_1.\tau_2 \preceq \tau'_1.\tau_2 \quad (1)$

if $P.Q \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_{\dot{l}}),\ (R_{\dot{r}})$ from Table 5.2, Definition 5.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$P \overset{\tau_1}{\twoheadrightarrow} 1, Q \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$\exists\tau'_1 \in \Sigma^*$ such that: $P' \overset{\tau'_1}{\twoheadrightarrow} 1$ and $\tau_1 \preceq \tau'_1$, $Q \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_{\dot{l}}),\ (R_{\dot{r}})$ from Table 5.2 and $\tau = \tau_1.\tau_2\ |\!\}$

$\exists\tau_2 \in \Sigma^*$ such that: $P'.Q \overset{\tau'_1.\tau_2}{\twoheadrightarrow} 1$ and $\tau = \tau_1.\tau_2 \preceq \tau'_1.\tau_2 \quad (2)$

From (1), (2) and Definition 5.4.4 ($\preceq$), we can say $P.Q \preceq P'.Q$.

- $Q.P \sqsubseteq_T Q.P'$

if $Q.P \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2, Definition 5.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$Q \downarrow \tau$ or $Q \overset{\tau_1}{\twoheadrightarrow} 1$, $P \downarrow \tau_2$

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$Q \downarrow \tau$ or $Q \overset{\tau_1}{\twoheadrightarrow} 1$, $\exists \tau_2' \in \Sigma^*, P' \downarrow \tau_2', \tau_2 \preceq \tau_2'$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2 $|\!\}$

$Q.P' \downarrow \tau$ or $\exists \tau_1.\tau_2' \in \Sigma^*, Q.P' \downarrow (\tau_1.\tau_2')$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5 $|\!\}$

$Q.P' \downarrow \tau, \tau \preceq \tau$ or $\exists \tau_1.\tau_2' \in \Sigma^*, Q.P' \downarrow (\tau_1.\tau_2'), \tau = \tau_1.\tau_2 \preceq \tau_1.\tau_2'$ (1)
if $Q.P \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2, Definition 5.4.1 and $\tau = \tau_1.\tau_2|\!\}$

$Q \overset{\tau_1}{\twoheadrightarrow} 1$, $P \overset{\tau_2}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$Q \overset{\tau_1}{\twoheadrightarrow} 1$, $\exists \tau_2' \in \Sigma^*$, such that: $P' \overset{\tau_2'}{\twoheadrightarrow} 1$ and $\tau_2 \preceq \tau_2'$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2 and $\tau = \tau_1.\tau_2\ |\!\}$

$\exists \tau_1.\tau_2' \in \Sigma^*$, such that $Q.P' \overset{\tau_1.\tau_2'}{\twoheadrightarrow} 1$ and $\tau = \tau_1.\tau_2 \preceq \tau_1.\tau'$ (2)

From (1), (2) and Definition 5.4.4 ($\preceq$), we can say $Q.P \preceq Q.P'$.

- $P^*Q \preceq P'^*Q$

let τ be a trace and $P^*Q{\downarrow}\tau$, if $Q{\downarrow}\tau$, we have $P'^*Q{\downarrow}\tau$ and $\tau \preceq \tau$

otherwise if $Q{\downarrow}\tau$ is not true.

$\Rightarrow$ $\{\!|(R_{\dot{l}}), (R_{\dot{r}})$ from Table 5.2, Definition 5.4.1 and
let $\tau = \tau_1.\tau_2$, $\tau_1 = \tau_1^1 ... \tau_1^n$, $1 \leq i \leq n.$ $|\!\}$

$P \overset{\tau_1^i}{\twoheadrightarrow} 1\ P{\downarrow}\tau_2$ or $P \overset{\tau_1^i}{\twoheadrightarrow} 1\ Q{\downarrow}\tau_2$.

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$\exists \tau_2' \in \Sigma^*$ such that: $P \overset{\tau_1^i}{\twoheadrightarrow} 1\ P'{\downarrow}\tau_2'$, $\tau_2 \preceq \tau_2'$ or $P \overset{\tau_1^i}{\twoheadrightarrow} 1\ Q{\downarrow}\tau_2$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$

$\exists \tau_2' \in \Sigma^*, \tau_1'^i \in \Sigma^*$ such that: $P' \overset{\tau_1'^i}{\twoheadrightarrow} 1\ P'{\downarrow}\tau_2'$, $\tau_2 \preceq \tau_2'$, $\tau_1^i \preceq \tau_1'^i$
or $\exists \tau_1'^i \in \Sigma^*$ such that: $P' \overset{\tau_1'^i}{\twoheadrightarrow} 1\ Q{\downarrow}\tau_2\ \tau_1^i \preceq \tau_1'^i$

$\Rightarrow$ $\{\!|(R_{\dot{l}}), (R_{\dot{r}})$ from Table 5.2 and let $\tau_1' = \tau_1'^1 ... \tau_1'^n$ $|\!\}$

$P'^*Q{\downarrow}\tau_1'.\tau_2'$ or $P'^*Q{\downarrow}(\tau_1'.\tau_2)$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5$|\!\}$

$P'^*Q{\downarrow}\tau_1'.\tau_2', \tau = \tau_1.\tau_2 \preceq \tau_1'.\tau_2'$ or $P'^*Q{\downarrow}(\tau_1'.\tau_2), \tau = \tau_1.\tau_2 \preceq \tau_1'.\tau_2$ (1)

let τ be a trace and $P^*Q \overset{\tau}{\twoheadrightarrow} 1$, if $Q \overset{\tau}{\twoheadrightarrow} 1$, we have $P'^*Q \overset{\tau}{\twoheadrightarrow} 1$ and $\tau \preceq \tau$

otherwise if $Q \overset{\tau}{\twoheadrightarrow} 1$ is not true.

$\Rightarrow$ $\{\!|(R_{\dot{l}}), (R_{\dot{r}})$ from Table 5.2, Definition 5.4.1 and
Let $\tau = \tau_1.\tau_2$, $\tau_1 = \tau_1^1 ... \tau_1^n$, $1 \leq i \leq n.$ $|\!\}$

$P \overset{\tau_1^i}{\twoheadrightarrow} 1, Q \overset{\tau_2}{\twoheadrightarrow} 1.$

$\Rightarrow$ $\{\!|(\beta)\ |\!\}$
$\exists, \tau_1'^i \in \Sigma^*$, such that: $P' \overset{\tau_1'^i}{\twoheadrightarrow} 1\ Q \overset{\tau_2}{\twoheadrightarrow} 1\ \tau_1^i \preceq \tau_1'^i$

$\Rightarrow$ $\{\!|(R_{\dot{l}}), (R_{\dot{r}})$ from Table 5.2 and let $\tau_1' = \tau_1'^1 ... \tau_1'^n$ $|\!\}$

$$P'^{*}Q \overset{\tau_1'.\tau_2}{\twoheadrightarrow} 1$$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5$|\!\}$

$$P'^{*}Q \overset{\tau_1'.\tau_2}{\twoheadrightarrow} 1, \tau = \tau_1.\tau_2 \preceq \tau_1'.\tau_2 \qquad (2)$$

From (1), (2) and Definition 5.4.4 ($\preceq$), we can say $P^{*}Q \preceq P'^{*}Q$.

- $Q^{*}P \preceq Q^{*}P'$

let τ be a trace and $Q^{*}P\downarrow\tau$,

if $P\downarrow\tau$

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$\exists\tau' \in \Sigma^{*}$ such that: $P'\downarrow\tau', \tau \preceq \tau'$

$\Rightarrow$ $\{\!|(R_l^{;})$, $(R_r^{;})$ from Table 5.2 $|\!\}$

$Q^{*}P'\downarrow\tau', \tau \preceq \tau'$

else if $P\downarrow\tau$ is not true.

$\Rightarrow$ $\{\!|(R_l^{;})$, $(R_r^{;})$ from Table 5.2, Definition 5.4.1 and let $\tau = \tau_1.\tau_2$, $\tau_1 = \tau_1^1 ... \tau_1^n$, $1 \leq i \leq n$. $|\!\}$

$Q \overset{\tau_1^i}{\twoheadrightarrow} 1\ Q\downarrow\tau_2$ or $Q \overset{\tau_1^i}{\twoheadrightarrow} 1\ P\downarrow\tau_2$.

$\Rightarrow$ $\{\!|(\alpha)\ |\!\}$

$Q \overset{\tau_1^i}{\twoheadrightarrow} 1\ Q\downarrow\tau_2$ or $\exists\tau_2' \in \Sigma^{*}$, such that: $Q \overset{\tau_1^i}{\twoheadrightarrow} 1\ P'\downarrow\tau_2'$, $\tau_2 \preceq \tau_2'$

$\Rightarrow$ $\{\!|(R_l^{;})$, $(R_r^{;})$ from Table 5.2 $|\!\}$

$Q^{*}P'\downarrow\tau_1.\tau_2$ or $Q^{*}P'\downarrow(\tau_1.\tau_2')$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5$|\!\}$

$Q^*P' \downarrow \tau_1.\tau_2, \tau_1.\tau_2 \preceq \tau$ or $Q^*P' \downarrow \tau_1.\tau_2', \tau = \tau_1.\tau_2 \preceq \tau_1.\tau_2' \quad (1)$

let τ be a trace and $Q^*P \overset{\tau}{\twoheadrightarrow} 1$, if $P \overset{\tau}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|(\beta)\, |\!\}$

$\exists \tau' \in \Sigma^*$, such that: $P' \overset{\tau'}{\twoheadrightarrow} 1, \tau \preceq \tau'$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2 $|\!\}$

$Q^*P' \overset{\tau'}{\twoheadrightarrow} 1, \tau \preceq \tau'$

else if $P \overset{\tau}{\twoheadrightarrow} 1$ is not true.

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2, Definition 5.4.1 and let $\tau = \tau_1.\tau_2, \tau_1 = \tau_1^1 ... \tau_1^n, 1 \leq i \leq n.$ $|\!\}$

$Q \overset{\tau_1^i}{\twoheadrightarrow} 1$, $P \overset{\tau_2}{\twoheadrightarrow} 1.$

$\Rightarrow$ $\{\!|(\beta)\, |\!\}$

$\exists, \tau_2' \in \Sigma^*$ such that: $Q \overset{\tau_1^i}{\twoheadrightarrow} 1 \; P' \overset{\tau_2'}{\twoheadrightarrow} 1 \; \tau_2 \preceq \tau_2'$

$\Rightarrow$ $\{\!|(R_l^{;}), (R_r^{;})$ from Table 5.2 and $\tau_1 = \tau_1^1 ... \tau_1^n$ $|\!\}$

$Q^*P' \overset{\tau_1.\tau_2'}{\twoheadrightarrow} 1$

$\Rightarrow$ $\{\!|\tau = \tau_1.\tau_2$ and Lemma 5.7.5$|\!\}$

$Q^*P' \overset{\tau_1.\tau_2'}{\twoheadrightarrow} 1, \tau = \tau_1.\tau_2 \preceq \tau_1.\tau_2' \qquad (2)$

From (1), (2) and Definition 5.4.4 ($\preceq$), we can say that $Q^*P \preceq Q^*P'$.

Conclusion: $\preceq$ is a congruent relationship with respect to operation "+", "." and "*".

Theorem 5.4.17: Let P and Q be two processes and s be an environment, then we have:

$$[P]_e \sqcap [Q]_e \sim [P \sqcap_e Q]_e$$

.

Proof:

Based on the definition of gcf, we need to prove that the three following statements are true.

1. $[P \sqcap_e Q]_e \sqsubseteq [P]_e$,
2. $[P \sqcap_e Q]_e \sqsubseteq [Q]_e$ and
3. For all $[R']_{e'}$ such that $[R']_{e'} \sqsubseteq [P]_e$ and $[R']_{e'} \sqsubseteq [Q]_e$, we have $[R']_{e'} \sqsubseteq [P \sqcap_e Q]_e$.

(1) Definition 5.4.13

$\Rightarrow$

$P \sqcap_e Q \preceq P$

$\Rightarrow$ {|Lemma 5.7.4 |}

$[P \sqcap_e Q]_e \sqsubseteq [P]_e$

(2) Definition 5.4.13

$\Rightarrow$

$P \sqcap_e Q \sqsubseteq_e Q$

$\Rightarrow$ {|Lemma 5.7.3 |}

$[P \sqcap_e Q]_e \sqsubseteq [Q]_e$

(3) Let $\tau_{rs'} \in \Sigma^*$ and $[R']'_e {\downarrow} \tau_{rs'}$

$\Rightarrow$ {|Definition 5.4.1 |}

$\exists [R'_n]_{e'_n} \in \mathcal{P}$ such that: $[R']_{e'} \overset{\tau_{rs'}}{\twoheadrightarrow} [R'_n]_{e'_n}$

$\Rightarrow$ {|Based on Rules $(R_a^{[]_e})$ and $(R_c^{[]_e})$ in Table 5.2, when a process run in an environment, it generates closed traces (no variables) without conditions.

Let $\tau_{rs'} = a_1(v_{a1}^1 ... v_{a1}^n) ... a_n(v_{an}^1 ... v_{an}^n)$, $(v_{ai}^i ... v_{ai}^n) \in (s_i(x_1) ... s_i(x_n)$ for $1 \leq i \leq n$ |}

$[R']_{e'} \overset{a_1(v_{a1}^1 ... v_{a1}^n) ... a_n(v_{an}^1 ... v_{an}^n)}{\twoheadrightarrow} [R'_n]_{e'_n}$

$\Rightarrow$ {|Definition 5.4.7 ($\sqsubseteq$) and $[R']_{e'} \sqsubseteq [P]_e$ |}

$\exists \tau_{ps} \in \Sigma^*$, such that $[P]_e {\downarrow} \tau_{ps}$ and $\tau_{rs'} \simeq \tau_{ps}$

$\Rightarrow$ {|Definition 5.4.1 |}

$\exists P_n \in \mathcal{P}$ and $s_n \in \mathcal{E}$ such that $[P]_e \overset{\tau_{ps}}{\twoheadrightarrow} [P_n]_{e_n}$

$\Rightarrow$ {|Let $\tau_{ps} = a_1(v_{a1}^1 ... v_{a1}^n) ... a_k(v_{a_k}^1 ... v_{a_k}^n)$ This will not make the proof lose its generality. |}

$[P]_e \overset{a_1(v_{a1}^1 ... v_{a1}^n) ... a_k(v_{a_k}^1 ... v_{a_k}^n)}{\twoheadrightarrow} [P_n]_{e_n}$

$\Rightarrow$ {|$(R_a^{[]_e})$, $(R_c^{[]_e})$ in Table 5.2 and let $s_i = eff(a_i, s_{i-1})$ |}

For each step $\left([P_i]_{e_i} \overset{a_i(v_{ai}^1, ..., v_{ai}^n)}{\longrightarrow} [P_{i+1}]_{e(i+1)}\right)$ of $[P]_e$, there is a corresponding step for process P.

Let $\tau_p = c_{p1}(x_p^1 ... x_p^n).a_1(x_p^1 ... x_p^n)...c_{pn}(x_p^1 ... x_p^n).a_n(x_p^1 ... x_p^n)$, where each $c_{pi}(x_p^1 ... x_p^n) \in \mathcal{B}$, we have:

$$\dfrac{P \overset{c_{p1}(x_p^1 \dots x_p^n)}{\longrightarrow} P_{c_1} \quad [c_{p1}(x_p^1 ... x_p^n)]_e\downarrow \quad \dfrac{P_{c_1} \overset{a_1(x_p^1 \dots x_p^n)}{\longrightarrow} P_1}{[P_{c_1}]_e \overset{a_1(v_{a1}^1 \dots v_{a1}^n)}{\longrightarrow} [P_1]_{e_1}}}{[P]_e \overset{a_1(v_{a1}^1 \dots v_{a1}^n)}{\longrightarrow} [P_1]_{e_1}} \quad ...$$

$$\dfrac{P_{n-1} \overset{c_{pn}(x_p^1 \dots x_p^n)}{\longrightarrow} P_{c_n} \quad [c_{pn}(x_p^1 ... x_p^n)]_{e_{n-1}}\downarrow \dfrac{P_{c_n} \overset{a_n(x_p^1 \dots x_p^n)}{\longrightarrow} P_n}{[P_{c_n}]_{e_{n-1}} \overset{a_n(v_{a1}^1 \dots v_{a1}^n)}{\longrightarrow} [P_n]_{e_n}}}{[P_{n-1}]_{e_{n-1}} \overset{a_n(v_{a1}^1 \dots v_{a1}^n)}{\longrightarrow} [P_n]_{e_n}} \qquad \gamma$$

$\Rightarrow$ $\{\!|$Definition 5.4.1 $|\!\}$

$P \overset{\tau_p}{\twoheadrightarrow} P_n$ $\qquad (\alpha)$

Same as above process, we can prove that: $Q \overset{\tau_q}{\twoheadrightarrow} Q_n$, $\tau_q = c_{q1}(x_q^1 ... x_q^n).a_1(x_q^1 ... x_q^n)...c_{qn}(x_q^1 ... x_q^n)$ $.a_n(x_q^1 ... x_q^n)$, where each $c_{qi}(x_q^1 ... x_q^n) \in \mathcal{B}$ $\qquad (\beta)$

$(*)$ Here, we want to build a process R_x, such that $R_x \preceq P$ and $R_x \sqsubseteq_e Q$. Let $R_x = c_{p1}(x_p^1, ..., x_p^n)$. $c_{q1}(x_q^1, ..., x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_1(x_p^1, ..., x_p^n)...c_{pn}(x_p^1, ..., x_p^n)$. $c_{qn}(x_q^1, ..., x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_n(x_p^1, ..., x_p^n).0$

Let $\tau = c_{p1}(x_p^1, ..., x_p^n).c_{q1}(x_q^1, ..., x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_1(x_p^1, ..., x_p^n)$ $...c_{pn}(x_p^1, ..., x_p^n).c_{qn}(x_q^1, ..., x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_n(x_p^1, ..., x_p^n)$
we have: $R_x \overset{\tau}{\twoheadrightarrow} 0$

$\Rightarrow$ $\{\!|(\alpha)$ and Definition 5.4.2's rule 2 $|\!\}$

$\exists \tau_p \in \Sigma^*$ such that: $P \overset{\tau_p}{\twoheadrightarrow} P_{c_{k+1}}$ and $\tau \preceq \tau_p$

$\Rightarrow$ $\{\!|\forall \tau'$, such that $R_x \downarrow \tau'$, τ' must be a prefix of τ. So we can always find a prefix of τ_p, which is τ_p', such that: $\tau' \preceq \tau_p'$ $|\!\}$

if $R_x \downarrow \tau'$ then $\exists \tau_p' \in \Sigma^*$ such that $P \downarrow \tau_p'$ and $\tau' \preceq \tau_p'$,

$\Rightarrow$ $\{\!|$There is no trace τ such that $R_x \overset{\tau}{\twoheadrightarrow} 1$ $|\!\}$

if $R_x \downarrow \tau'$ then $\exists \tau_p' \in \Sigma^*$ such that $P \downarrow \tau_p'$ and $\tau' \preceq \tau_p'$,
if $R_x \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*, \tau \preceq \tau'$ such that $P \overset{\tau'}{\twoheadrightarrow} 1$.

$\Rightarrow$ $\{\!|$Definition 5.4.5 $|\!\}$

$R_x \preceq P$

Based on Definition 5.4.3 and by repeating the above process, we have $R_x \sqsubseteq_e Q$

$\Rightarrow$ $\{\!|$Definition 5.4.13 and $R_x \preceq P$ $|\!\}$

$R_x \preceq P \sqcap_e Q$

$\Rightarrow$ $\{\!|$Lemma 5.7.4 $|\!\}$

$[R_x]_e \sqsubseteq [P \sqcap_e Q]_e \qquad (\eta)$

$R_x \overset{\tau}{\twoheadrightarrow} 0$

$\Rightarrow$ $\{\!|(R_a^{[]_e}), (R_c^{[]_e})$ in Table 5.2 Since the trace τ has the same actions as the trace τ_p and the sequence of these actions are also the same. So, if in the environment e, each action's variables can have same values as in (γ), and since they have the same values, they have the same effects on environments, so we still have $eff(a, s_{i-1}) = s_i$. Because the environments are same as in (γ), c_{pi} and c_{qi} could also be true.$|\!\}$

For $\tau = c_{p1}(x_p^1...x_p^n).c_{q1}(x_q^1...x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_1(x_p^1...x_p^n)...c_{pn}(x_p^1...x_p^n).$
$c_{qn}(x_q^1...x_q^n).(x_p^1 == x_q^1 \wedge ... \wedge x_p^n == x_q^n).a_n(x_p^1...x_p^n)$

$$\frac{\left(R_x \overset{c_{p1}(x_p^1...x_p^n)}{\longrightarrow} R_{c_{p1}},\ [c_{p1}(x_p^1...x_p^n)]_e\downarrow,\ R_{c_{p1}} \overset{c_{q1}(x_q^1...x_q^n)}{\longrightarrow} R_{c_{q1}},\ [c_{q1}(x_q^1...x_q^n)]_e\downarrow,\ R_{c_{q1}} \overset{(x_p^1==x_q^1\wedge...\wedge x_p^n==x_q^n)}{\longrightarrow} R_{c_1},\ \frac{\llbracket(v_{a_1}^1==v_{a_1}^1\wedge...\wedge v_{a_1}^n==v_{a_1}^n)\rrbracket_{\mathcal{B}}=true,\ v_{a_1}^i\in s(x_p^i),\ v_{a_1}^i\in s(x_q^i)}{[(x_p^1==x_q^1\wedge...\wedge x_p^n==x_q^n)]_e\downarrow}\ \ \frac{R_{c_1} \overset{a_1(x_p^1,...,x_p^n)}{\longrightarrow} R_{x1}}{[R_{c_1}]_e \overset{a_1(v_{a1}^1,...,v_{a1}^n)}{\longrightarrow} [R_{x1}]_{e_1}} \right)}{[R_x]_e \overset{a_1(v_{a1}^1,...,v_{a1}^n)}{\longrightarrow} [R_{x1}]_{e_1}} \ ......$$

$$\frac{\left(R_{x(n-1)} \overset{c_{pn}(x_p^1...x_p^n)}{\longrightarrow} R_{c_{pn}},\ [c_{pn}(x_p^1...x_p^n)]_{e_{n-1}}\downarrow,\ R_{c_{pn}} \overset{c_{qn}(x_q^1...x_q^n)}{\longrightarrow} R_{c_{qn}},\ [c_{qn}(x_q^1...x_q^n)]_{e_{n-1}}\downarrow,\ R_{c_{qn}} \overset{(x_p^1==x_q^1\wedge...\wedge x_p^n==x_q^n)}{\longrightarrow} R_{c_n},\ \frac{\llbracket v_{a_n}^1==v_{a_n}^1 ... v_{a_n}^n==v_{a_n}^n\rrbracket_{\mathcal{B}}=true,\ v_{a_n}^i\in s_{n-1}(x_p^i),\ v_{a_n}^i\in s_{n-1}(x_q^i)}{[(x_p^1==x_q^1\wedge...\wedge x_p^n==x_q^n)]_{e_{n-1}}\downarrow}\ \ \frac{R_{c_n} \overset{a_n(x_p^1,...,x_p^n)}{\longrightarrow} 0}{[R_{c_n}]_{e_{n-1}} \overset{a_n(v_{a1}^1,...,v_{a1}^n)}{\longrightarrow} [0]_{e_n}} \right)}{[R_{x(n-1)}]_{e_{n-1}} \overset{a_n(v_{a1}^1,...,v_{a1}^n)}{\longrightarrow} [0]_{e_n}}$$

$\Rightarrow$ $\{\!|$Definition 5.4.1 and let $\tau_{xs} = a_1(v_{a1}^1...v_{a1}^n)...a_n(v_{an}^1...v_{an}^n)$ $|\!\}$

$[R_x]_e \overset{\tau_{xs}=a_1(v_{a1}^1...v_{a1}^n)...a_n(v_{an}^1...v_{an}^n)}{\twoheadrightarrow} [0]'_e$

$\Rightarrow$ $\{\!|(\eta)$ ($[R_x]_e \sqsubseteq_e [P \sqcap_e Q]_e$) and Definition 5.4.7 $|\!\}$

$\exists\tau_{\sqcap s} \in \Sigma^*$ such that:$[P \sqcap_e Q]_e \downarrow\!\!\!\!\downarrow \tau_{\sqcap s}$ and $\tau_{xs} \simeq \tau_{\sqcap s}$.

$\Rightarrow$ $\{\!|\tau_{rs'} = a_1(v_{a1}^1...v_{a1}^n)...a_n(v_{a_n}^1...v_{a_n}^n)$ $|\!\}$

$\exists\tau_{\sqcap s} \in \Sigma^*$ such that: $[P \sqcap_e Q]_e \downarrow\!\!\!\!\downarrow \tau_{\sqcap s}$ and $\tau_{rs'} \simeq \tau_{xs} \simeq \tau_{\sqcap s}$,

$\Rightarrow$ $\{\!|$We choose $\tau_{rs'}$ arbitrarily $|\!\}$

$\forall\tau_{rs'} \in \Sigma^*$, if $[R']'_e \downarrow\!\!\!\!\downarrow \tau_{rs'}$ then $\exists\tau_{\sqcap s} \in \Sigma^*$ such that: $[P \sqcap Q]_e \downarrow\!\!\!\!\downarrow \tau_{\sqcap s}$ and $\tau_{rs'} \simeq \tau_{\sqcap s}$.

$\Rightarrow$ $\{\!|$Definition 5.4.7 $|\!\}$

$[R']'_e \sqsubseteq [P \sqcap Q]_e$

- Conclusion Theorem 5.4.17 is true.

□

Lemma 5.7.6 *Let P and Q be two processes, $e \in \mathcal{E}$, τ, τ_p and $\tau_q \in \Sigma^*$, then if $\mathcal{V}(P) \subseteq N_v(e)$, $P{\downarrow}\tau_p, Q{\downarrow}\tau_q$, $\tau \preceq \tau_p$ and $\tau \sqsubseteq_e \tau_q$, then $\exists\tau_{pq} \in \Sigma^*$ such that $(P \sqcap_e Q){\downarrow}\tau_{pq}$ and $\tau \preceq \tau_{pq}$.*

Proof:

Let R be a process and $R = \tau.0$

$\Rightarrow$ $\{\!|P{\downarrow}\tau_p, \tau \preceq \tau_p\,|\!\}$

$\forall\tau \in \Sigma^*$, if $R{\downarrow}\tau$ then $\exists\tau' \in \Sigma^*$, such that:$P'{\downarrow}\tau'$ and $\tau \preceq \tau'$.

$\Rightarrow$ $\{\!|$There is no trace τ, such that $R \overset{\tau}{\twoheadrightarrow} 1\,|\!\}$

$\forall\tau$, if $R \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$P \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$.

$\Rightarrow$ $\{\!|$Definition 5.4.4 $|\!\}$

$R \preceq P$

$\Rightarrow$ $\{\!|$We can prove that $R \sqsubseteq_e Q$ by repeating the above steps. $|\!\}$

$R \preceq P, R \sqsubseteq_e Q$

$\Rightarrow$ $\{\!|$Definition 5.4.15 $|\!\}$

$R \preceq P \sqcap_e Q$

$\Rightarrow$ $\{\!|$Definition 5.4.5 and $R{\downarrow}\tau$ $|\!\}$

$\exists\tau_{pq} \in \Sigma^*, (P \sqcap_e Q){\downarrow}\tau_{pq}$ and $\tau \preceq \tau_{pq}$

□

5.7.3 Proof for Proposition 5.5.6

Proposition 5.5.6: $P \approx o(P) + \sum_{\alpha\in\delta(P)} \alpha\ \partial_\alpha(P)$

Proof:

The proof is by structural induction on the process of $CBPA^*_{0,1}\left(0 \mid 1 \mid a \mid c \mid P{+}Q \mid P.Q \mid P^*Q \mid \lambda xP\right)$. Since $\approx$ is a congruence relationship for $BPA^*_{0,1}$ and $CBPA^*_{0,1}$ add only conditions, then we need only to upgrade the proof of Proposition 4.5.6 in page 63 to conditions induction.

When $P = c$:

$o(c) + \sum_{\alpha\in\delta(c)} \alpha\ \partial_\alpha(c)$

= $\{\!|o(c) = 0$ and $\delta(c) = \{c\}|\!\}$
$0 + \sum_{\alpha \in \{c\}} \alpha\ \partial_\alpha(c)$
=
$0 + c.\partial_c(c)$
= $\{\!|\partial_c(c) = 1|\!\}$
$0 + c.1$
= $\{\!|P.1 \approx P$ and $0 + P \approx P|\!\}$
c

□

5.7.4 Proof for Proposition 5.5.13

Lemma 5.7.7 $\forall \tau, \tau', \tau_1$ *and* $\tau_2 \in \Sigma^*$, $\sigma \in \Gamma$, *we have:*

- $\tau.\tau' \sqsubseteq_e \tau_1.\tau_2$ and $\tau \preceq \tau_1\sigma \Longrightarrow \tau' \sqsubseteq_e \tau_2\sigma$.
- $\tau.\tau' \sqsubseteq_e \tau_1.\tau_2$ and $\tau' \preceq \tau_2\sigma \Longrightarrow \tau \sqsubseteq_e \tau_1\sigma$.
- $\tau \preceq \tau'$ and $\tau_1 \sqsubseteq_e \tau_2 \Longrightarrow \tau.\tau_1 \sqsubseteq_e \tau'.\tau_2$

Proof: Directly from Definition 5.4.3 and Definition 5.4.2. □

Lemma 5.7.8 *Let* $a_1, a_2 \in \mathcal{A}$ *and* $e \in \mathcal{E}$, *such that* $a_1 \nabla_e a_2 = \{(\alpha, \sigma)\}$. *Then:*

- $\alpha \approx a_1 \sqcap_e a_2$
- $\alpha \preceq a_1$
- $\alpha \preceq a_2\sigma$
- $\alpha \sqsubseteq_e a_2$

Proof:

Directly from definition of 5.2.9.

□

Proposition 5.5.13:

For $P, Q \in \mathcal{P}, e \in \mathcal{E}, a_1, a_2 \in \mathcal{A}, c_2 \in \mathcal{B}$, if $\mathcal{V}(c_1.P) \subseteq N_v(e)$, we have:

$$a)\begin{cases} \underline{a_1.P \sqcap_e a_2.Q} \approx \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\big(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\big) \\ \qquad\qquad \text{where } (a, \sigma_\delta) \in a_1 \nabla_e a_2 \\ \underline{\Gamma_{a_1.P, a_2.Q}} = T(\underline{\Gamma}, (\underline{a_1.P \sqcap_e a_2.Q})\Downarrow) \end{cases}$$

$$b)\begin{cases} \underline{a_1.P \sqcap_e c_2.Q} \approx \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(a_1P, Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big) \\ \underline{\Gamma_{a_1.P \sqcap_e c_2.Q}} = T(\underline{\Gamma}, (\underline{a_1.P \sqcap_e c_2.Q})\Downarrow) \end{cases}$$

For $P, Q \in \mathcal{P}, e \in \mathcal{E}, a_2 \in \mathcal{A}, c_1, c_2 \in \mathcal{B}$, if $\mathcal{V}(c_1.P) \subseteq N_v(e)$, we have:

$$c)\begin{cases} \underline{c_1.P \sqcap_e a_2.Q} \approx \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(P, a_2.Q)}}} c_1\sigma_\partial.\big(\underline{P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial}\big) \\ \underline{\Gamma_{c_1.P \sqcap_e a_2.Q}} = T(\underline{\Gamma}, (\underline{c_1.P \sqcap_e a_2.Q})\Downarrow) \end{cases}$$

$$d)\begin{cases} \underline{c_1.P \sqcap_e c_2.Q} \approx \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(P, Q)}}} (c_1.c_2)\sigma_\partial.\big(\underline{P\sigma_\partial \sqcap_e Q\sigma_\partial}\big) \\ \underline{\Gamma_{c_1.P \sqcap_e c_2.Q}} = T(\underline{\Gamma}, (\underline{c_1.P \sqcap_e c_2.Q})\Downarrow) \end{cases}$$

Proof:

In $a)$, for each variable $\underline{\Gamma_{(P,Q)}}$, we let $\underline{\Gamma_{(P,Q)}} = T(\underline{\Gamma}, (\underline{P \sqcap_e Q})\Downarrow)$.
Suppose $\underline{R} = \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\big(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\big)$, where $(a, \sigma_\delta) \in a_1 \nabla_e a_2$
based on the definition of gcf_e, we need to prove that the three following statements are true.

1. $R \preceq a_1.P$,
2. $R \sqsubseteq_e a_2.Q$ and
3. For all R' such that $R' \preceq a_1.P$ and $R' \sqsubseteq_e a_2.Q$, we have $R' \preceq R$.

(1) To prove that $\underline{R} \preceq a_1.P$, we need to prove the following.

$\forall \tau \in \Sigma^*$, if $\underline{R}{\downarrow}\tau$ then $\exists \tau' \in \Sigma^*$ such that:$a_1.P{\downarrow}\tau'$ and $\tau \preceq \tau'$. (1_a)
$\forall \tau \in \Sigma^*$, if $\underline{R} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*$ such that:$a_1.P \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. (1_b)

For (1_a), let τ be a trace and $\underline{R}{\downarrow}\tau$

$$\Rightarrow \quad \{\!|\underline{R} = \sum_{\sigma_\partial \,\in\, \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\big(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\big) \text{ where } (a, \sigma_\delta) \in a_1 \nabla_e a_2 \,|\!\}$$

$$\left(\sum_{\sigma_\partial \in \Gamma_{(P\sigma_\delta, Q\sigma_\delta)}} a\sigma_\partial.(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta)\right)\downarrow\tau$$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2. $|\!\}$

$\exists\sigma_\partial \in \Gamma_{(P\sigma_\delta, Q\sigma_\delta)}$, such that:$[a\sigma_\partial.(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta)]\downarrow\tau$

(α_1) When $\tau = a\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau' = a_1$ $(a, \sigma_\delta) \in a_1 \nabla_e a_2$ and Lemma 5.7.8 $|\!\}$

$a_1.P\downarrow\tau'$ and $a \preceq \tau'$

$\Rightarrow$ $\{\!|\sigma_\partial \in \Gamma_{(P\sigma_\delta, Q\sigma_\delta)}, \mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect a $|\!\}$

$a_1.P\downarrow\tau'$ and $\tau = a\sigma_\partial \preceq \tau'$

(α_2) When $\tau \neq a\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau = a\sigma_\partial.\tau_1$, $\left(\sum_{\sigma_\partial \in \Gamma_{(P\sigma_\delta, Q\sigma_\delta)}} a\sigma_\partial.(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta)\right)\downarrow\tau$
and $(R_l^;), (R_r^;)$ from Table 5.2 $|\!\}$

$(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta)\downarrow\tau_1$

$\Rightarrow$ $\{\!|(P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta) \preceq P\sigma_\partial\sigma_\delta$ and Definition 5.4.4 $|\!\}$

$\exists\tau_p \in \Sigma^*$, such that: $P\sigma_\partial\sigma_\delta\downarrow\tau_p$ and $\tau_1 \preceq \tau_p$

$\Rightarrow$ $\{\!|\mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial\sigma_\delta$ will not affect P $|\!\}$

$P\downarrow\tau_p$ and $\tau_1 \preceq \tau_p$

$\Rightarrow$ $\{\!|a\sigma_\partial \preceq a_1$ and Proposition 5.4.6 $|\!\}$

$\tau = c_2\sigma_\partial.\tau_1 \preceq a_1.\tau_p$

$\Rightarrow$ $\{\!|a_1.P\downarrow a_1.\tau_p$ $|\!\}$

$\exists\tau' = a_1.\tau_p$, such that: $a_1.P\downarrow\tau'$ and $\tau \preceq \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall\tau \in \Sigma^*$, if $R\downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_1.P\downarrow\tau'$ and $\tau \preceq \tau'$. (1_a)

We can prove (1_b) by repeating the above steps.

(1_a) and $(1_b) \Rightarrow (1)$ is true:

(2) To prove $R \sqsubseteq_e a_2.Q$, we need to prove the following:

$\forall \tau \in \Sigma^*$, if $\underline{R}{\downarrow}\tau$ then $\exists \tau' \in \Sigma^*, \mid a_2.Q{\downarrow}\tau'$ and $\tau \sqsubseteq_e \tau'$. $\quad (2_a)$

$\forall \tau \in \Sigma^*$, if $\underline{R} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*, \mid a_2.Q \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \sqsubseteq_e \tau'$. $\quad (2_b)$

For (2_a), let τ be a trace and $\underline{R}{\downarrow}\tau$

$\Rightarrow \quad \{\!|\, \underline{R} = \sum_{\sigma_\partial \in \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\left(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\right)$ where $(a, \sigma_\delta) \in a_1 \nabla_e a_2 \,|\!\}$

$\left(\sum_{\sigma_\partial \in \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\left(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\right) \right){\downarrow}\tau$

$\Rightarrow \quad \{\!|\, (R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

$\exists \sigma_\partial \in \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}$, such that: $[a\sigma_\partial.\left(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\right)]{\downarrow}\tau$

(α_1) When $\tau = a\sigma_\partial$

$\Rightarrow \quad \{\!|\,$Let $\tau' = a_2$ and Lemma 5.7.8 $|\!\}$

$a_2.Q{\downarrow}\tau'$ and $a \sqsubseteq_e \tau'$

$\Rightarrow \quad \{\!|\, \sigma_\partial$ substitutes only free variables to non free ones$|\!\}$

$a_2.Q{\downarrow}\tau'$ and $a\sigma_\partial \sqsubseteq_e a \sqsubseteq_e \tau'$

(α_2) When $\tau \neq a\sigma_\partial$

$\Rightarrow \quad \{\!|\,$Let $\tau = a\sigma_\partial.\tau_1$ and $\left(\sum_{\sigma_\partial \in \underline{\Gamma_{(P\sigma_\delta, Q\sigma_\delta)}}} a\sigma_\partial.\left(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\right) \right){\downarrow}\tau$,

$(R_l^;), (R_r^;)$ from Table 5.2 $|\!\}$

$\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}{\downarrow}\tau_1$

$\Rightarrow \quad \{\!|\, \left(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\right) \sqsubseteq_e Q\sigma_\partial\sigma_\delta$ and Definition 5.4.5 $|\!\}$

$\exists \tau_q \in \Sigma^*$, such that: $Q{\downarrow}\tau_q, Q\sigma_\partial\sigma_\delta{\downarrow}\tau_q\sigma_\partial\sigma_\delta$ and $\tau_1 \sqsubseteq_e \tau_q\sigma_\partial\sigma_\delta$

$\Rightarrow \quad \{\!|\, \sigma_\partial$ is a sorted substitution $|\!\}$

and $\tau_1 \sqsubseteq_e \tau_q\sigma_\partial\sigma_\delta \sqsubseteq_e \tau_q\sigma_\delta$

$\Rightarrow \quad \{\!|\, (a, \sigma_\delta) \in a_1 \nabla_e a_2$ and Lemma 5.7.8 $|\!\}$

$a \preceq a_2\sigma_\delta$

$\Rightarrow \quad \{\!|\, \tau_1 \sqsubseteq_e \tau_q\sigma_\delta$ and Lemma 5.7.7 $|\!\}$

$a.\tau_1 \sqsubseteq_e a_2\sigma_\delta.\tau_q\sigma_\delta$

$\Rightarrow$ $\{\!|\sigma_\delta$ is a sorted substitution $|\!\}$

$a.\tau_1 \sqsubseteq_e a_2\tau_q$

$\Rightarrow$ $\{\!|$Let $\tau' = a_2.\tau_q$ and Definition 5.4.5 $|\!\}$

$\exists\tau' \in \Sigma^*$, such that: $a_2.Q{\downarrow}\tau'$ and $\tau = a.\tau_1 \sqsubseteq_e a_2.\tau_q = \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall\tau \in \Sigma^*$, if $R{\downarrow}\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_2.Q{\downarrow}\tau'$
and $\tau \sqsubseteq_e \tau'$. (2_a)

We can prove (2_b) by repeating the above steps.

(2_a) and $(2_b) \Rightarrow$ (2) is true:

(3) For all R' such that $R' \preceq a_1.P$ and $R' \sqsubseteq_e a_2.Q$, we have $R' \preceq R$. To prove $R' \preceq R$, we need to prove the following:

if $R'{\downarrow}\tau$ then $\exists\tau' \in \Sigma^*$, such that: $R{\downarrow}\tau'$ and $\tau \preceq \tau'$. (3_a)
if $R' \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$R \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. (3_b)

For (3_a), let τ be a trace and $R'{\downarrow}\tau$

$\Rightarrow$ $\{\!|R' \preceq a_1.P$, $R' \sqsubseteq_e a_2.Q$ and Definition 5.4.5 $|\!\}$

$\exists\tau_p \in \Sigma^*$, such that: $a_1.P{\downarrow}a_1.\tau_p$ $P{\downarrow}\tau_p$ and $\tau \preceq a_1.\tau_p$
$\exists\tau_q \in \Sigma^*$, such that: $a_2.Q{\downarrow}a_2.\tau_q$ $Q{\downarrow}\tau_q$and $\tau \sqsubseteq_e a_2.\tau_q$

$\Rightarrow$ $\{\!|$let $\tau = \tau'.\tau''$, $\tau' \preceq a_1$ $\tau'' \preceq \tau_p$ and $\tau' \sqsubseteq_e a_2, \tau'' \sqsubseteq_e \tau_q$ $|\!\}$

$a_1.P{\downarrow}a_1.\tau_p$ and $\tau'.\tau'' \preceq a_1.\tau_p$, $\tau' \preceq a_1$
$a_2.Q{\downarrow}a_2.\tau_q$ and $\tau'.\tau'' \sqsubseteq_e a_2.\tau_q$, $\tau' \sqsubseteq_e a_2$

$\Rightarrow$ $\{\!|(a, \sigma_\delta) \in a_1 \nabla_e a_2, a \approx a_1 \sqcap_e a_2$,
and Lemma 5.7.6 $|\!\}$

$a_1.P{\downarrow}a_1.\tau_p$ and $\tau'.\tau'' \preceq a_1.\tau_p$, $\tau' \preceq a$
$a_2.Q{\downarrow}a_2.\tau_q$ and $\tau'.\tau'' \sqsubseteq_e a_2.\tau_q$, $\tau' \preceq a$

$\Rightarrow$

$\tau' \preceq a$

$\Rightarrow$ $\{\!|\mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect a $|\!\}$

$\tau' \preceq a.\sigma_\partial$

$\Rightarrow$ $\{\!|(a, \sigma_\delta) \in a_1 \nabla_e a_2$, Lemma 5.7.8 and Lemma 5.7.5 $|\!\}$

$\tau' \preceq a.\sigma_\partial$, $a \preceq a_2\sigma_\delta$

$\Rightarrow$

$\tau' \preceq a.\sigma_\partial,\ a\sigma_\partial \preceq a_2\sigma_\partial\sigma_\delta$

$\Rightarrow$ $\{\!|$Lemma 5.7.5 $|\!\}$

$\tau' \preceq a_2\sigma_\partial\sigma_\delta$

$\Rightarrow$ $\{\!|\tau = \tau'.\tau'' \sqsubseteq_e a_2.\tau_q,\ \tau'' \sqsubseteq_e \tau_q$, and Lemma 5.7.7 $|\!\}$

$\tau'' \sqsubseteq_e \tau_q\sigma_\partial\sigma_\delta$

$\Rightarrow$ $\{\!|\tau'' \preceq \tau_p$
and $\mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial\sigma_\delta$ will not affect τ_p $|\!\}$

$\tau'' \sqsubseteq_e \tau_q\sigma_\partial\sigma_\delta,\ \tau'' \preceq \sigma_\partial\sigma_\delta$

$\Rightarrow$ $\{\!|P\sigma_\partial\sigma_\delta{\downarrow}\tau_p\sigma_\partial\sigma_\delta, Q\sigma_\partial\sigma_\delta{\downarrow}\tau_q\sigma_\partial\sigma_\delta$ and Lemma 5.7.6 $|\!\}$

$\exists\tau_{pq} \in \Sigma^*$, such that: $\big(\underline{P\sigma_\partial\sigma_\delta \sqcap_e Q\sigma_\partial\sigma_\delta}\big){\downarrow}\tau_{pq},\ \tau'' \preceq \tau_{pq}$

$\Rightarrow$ $\{\!|\tau' \preceq a.\sigma_\partial$ and Lemma 5.7.5 $|\!\}$

$\tau'.\tau'' \preceq a.\sigma_\partial.\tau_{pq}$

$\Rightarrow$

if $\underline{R'}{\downarrow}\tau, \tau = \tau'\tau''$ then $\exists a.\sigma_\partial.\tau_{pq} \in \Sigma^*$, such that:$\underline{R}{\downarrow}a.\sigma_\partial.\tau_{pq}$ and $\tau = \tau'\tau'' \preceq a.\sigma_\partial.\tau_{pq}$. $\quad (3_a)$

We can prove (3_b) by repeating the above steps.

(3_a) and $(3_b) \Rightarrow (3)$ is true.
$(1), (2)$ and $(3) \Rightarrow (a)$ is true

In b), for each variable $\underline{\Gamma_{(P,Q)}}$, we let $\underline{\Gamma_{(P,Q)}} = T(\underline{\Gamma}, (\underline{P \sqcap_e Q}){\Downarrow})$. Suppose
$\underline{R} = \sum_{\sigma_\partial \in \underline{\Gamma_{(a_1P,Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)$ based on the definition of gcf_e, we need to prove that the following three statements.

1. $\underline{R} \preceq a_1.P$,
2. $\underline{R} \sqsubseteq_e c_2.Q$ and
3. For all $\underline{R'}$ such that $\underline{R'} \preceq a_1.P$ and $\underline{R'} \sqsubseteq_e c_2.Q$, we have $\underline{R'} \preceq \underline{R}$.

(1) To prove $\underline{R} \preceq a_1.P$, we need to prove the following:

$\forall\tau \in \Sigma^*$, if $\underline{R}{\downarrow}\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_1.P{\downarrow}\tau'$ and $\tau \preceq \tau'$. $\quad (1_a)$
$\forall\tau \in \Sigma^*$, if $\underline{R} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$a_1.P \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. $\quad (1_b)$

For (1_a), let τ be a trace and $\underline{R}{\downarrow}\tau$

$\Rightarrow$ $\{\!|\underline{R} = \sum_{\sigma_\partial \in \underline{\Gamma_{(a_1P,Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)$ $|\!\}$

$\left(\sum_{\sigma_\partial \in \Gamma_{(a_1 P, Q)}} c_2\sigma_\partial.\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big) \right) \downarrow \tau$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

$\exists \sigma_\partial \in \Gamma_{(a_1 P, Q)}$, such that:$[c_2\sigma_\partial.\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big)]\downarrow\tau$

(α_1) When $\tau = c_2\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau' = \epsilon$ and Lemma 5.7.8 $|\!\}$

$a_1.P\downarrow\tau'$ and $\tau \preceq \tau'$

(α_2) When $\tau \neq c_2\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau = c_2\sigma_\partial.\tau_1$, $[c_2\sigma_\partial.\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big)]\downarrow\tau$ and $(R_l^{;})$, $(R_r^{;})$ from Table 5.2 $|\!\}$

$\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big)\downarrow\tau_1$

$\Rightarrow$ $\{\!|\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big) \preceq (a_1.P)\sigma_\partial$ and Definition 5.4.4 $|\!\}$

$\exists \tau_p \in \Sigma^*$, such that: $(a_1.P)\sigma_\partial\downarrow(a_1.\tau_p)\sigma_\partial$ and $\tau_1 \preceq (a_1.\tau_p)\sigma_\partial$

$\Rightarrow$ $\{\!|\mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect $a_1.\tau_p$ $|\!\}$

$\tau_1 \preceq a_1.\tau_p$

$\Rightarrow$ $\{\!|$Rule 2 of Definition 5.4.2 $|\!\}$

$\tau = c_2\sigma_\partial.\tau_1 \preceq a_1.\tau_p$

$\Rightarrow$ $\{\!|$Let $\tau' = a_1.\tau_p$ and Definition 5.4.5 $|\!\}$

$\exists \tau' \in \Sigma^*$ such that: $a_1.P\downarrow\tau'$ and $\tau = c_2\sigma_\partial.\tau_1 \preceq \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall \tau \in \Sigma^*$, if $\underline{R}\downarrow\tau$ then $\exists \tau' \in \Sigma^*$, such that: $a_1.P\downarrow\tau'$ and $\tau \preceq \tau'$. (1_a)

We can prove (1_b) by repeating the above steps.

(1_a) and $(1_b) \Rightarrow (1)$ is true:

(2) To prove $\underline{R} \sqsubseteq_e c_2.Q$, we need to prove the following:

$\forall \tau \in \Sigma^*$, if $\underline{R}\downarrow\tau$ then $\exists \tau' \in \Sigma^*$, such that:$c_2.Q\downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$. (2_a)
$\forall \tau \in \Sigma^*$, if $\underline{R} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*$, such that:$c_2.Q \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \sqsubseteq_e \tau'$. (2_b)

For (2_a), let τ be a trace and $\underline{R}\downarrow\tau$

$\Rightarrow$ $\{\!|\underline{R} = \sum_{\sigma_\partial \in \underline{\Gamma_{(a_1 P, Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)\ |\!\}$

$$\Big(\sum_{\sigma_\partial \in \underline{\Gamma_{(a_1 P, Q)}}} c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)\Big)\downarrow\tau$$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

$\exists\sigma_\partial \in \underline{\Gamma_{(a_1 P, Q)}}$, such that:$[c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)]\downarrow\tau$

(α_1) When $\tau = c_2\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau' = c_2$ and σ_∂ is a sorted substitution $|\!\}$

$c_2.Q\downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$

(α_2) When $\tau \neq c_2\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau = c_2\sigma_\partial.\tau_1$, $[c_2\sigma_\partial.\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)]\downarrow\tau$ and $(R_l^\cdot), (R_r^\cdot)$ from Table 5.2 $|\!\}$

$\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big)\downarrow\tau_1$

$\Rightarrow$ $\{\!|\big(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\big) \sqsubseteq_e Q\sigma_\partial$ and Definition 5.4.5 $|\!\}$

$\exists\tau_q \in \Sigma^*$, such that: $Q\sigma_\partial\downarrow\tau_q\sigma_\partial, Q\downarrow\tau_q$ and $\tau_1 \sqsubseteq_e \tau_q\sigma_\partial$

$\Rightarrow$ $\{\!|\sigma_\partial \in \Gamma^e_{a_1.P,Q} \Longrightarrow \tau_1 \preceq \tau_q\sigma_\partial\ |\!\}$

$\exists\tau_q \in \Sigma^*$, such that: $Q\sigma_\partial\downarrow\tau_q\sigma_\partial$ and $\tau_1 \preceq \tau_q\sigma_\partial$

$\Rightarrow$ $\{\!|$Lemma 5.7.5 $|\!\}$

$c_2\sigma_\partial.\tau_1 \preceq c_2\sigma_\partial.\tau_q\sigma_\partial$

$\Rightarrow$ $\{\!|\sigma_\partial$ is a sorted substitution $|\!\}$

$c_2\sigma_\partial.\tau_1 \sqsubseteq_e c_2.\tau_q$

$\Rightarrow$ $\{\!|$Let $\tau' = c_2.\tau_q$ and Definition 5.4.5 $|\!\}$

$\exists\tau' \in \Sigma^*$, such that: $c_2.Q\downarrow\tau'$ and $\tau = c_2\sigma_\partial.\tau_1 \sqsubseteq_e \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall\tau \in \Sigma^*$, if $\underline{R}\downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$c_2.Q\downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$. (2_a)

We can prove (2_b) by repeating the above steps.

(2_a) and $(2_b) \Rightarrow (2)$ is true.

(3) For all $\underline{R'}$ such that $\underline{R'} \preceq a_1.P$ and $\underline{R'} \sqsubseteq_e c_2.Q$, we have $\underline{R'} \preceq \underline{R}$. To prove $\underline{R'} \preceq \underline{R}$,

we need to prove the following:

$\forall \tau \in \Sigma^*$, if $\underline{R'} \downarrow \tau$ then $\exists \tau' \in \Sigma^*$, such that: $\underline{R} \downarrow \tau'$ and $\tau \preceq \tau'$. (3_a)

$\forall \tau \in \Sigma^*$, if $\underline{R'} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists \tau' \in \Sigma^*$, such that: $\underline{R} \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. (3_b)

For (3_a), let τ be a trace and $\underline{R'} \downarrow \tau$

$\Rightarrow$ $\{\!|\underline{R'} \preceq a_1.P$, $\underline{R'} \sqsubseteq_e c_2.Q$ and Definition 5.4.5 $|\!\}$

$\exists \tau_p \in \Sigma^*$, such that: $a_1.P \downarrow a_1.\tau_p$ $P \downarrow \tau_p$ and $\tau \preceq a_1.\tau_p$
$\exists \tau_q \in \Sigma^*$, such that: $c_2.Q \downarrow c_2.\tau_q$ $Q \downarrow \tau_q$and $\tau \sqsubseteq_e c_2.\tau_q$

$\Rightarrow$ $\{\!|$let $\tau = \tau'.\tau''$, $\tau' \preceq \epsilon$ $\tau'' \preceq a_1.\tau_p$ and $\tau' \sqsubseteq_e c_2, \tau'' \sqsubseteq_e \tau_q$ $|\!\}$

$a_1.P \downarrow a_1.\tau_p$ and $\tau'.\tau'' \preceq a_1.\tau_p$, $\tau' \preceq \epsilon$, $\tau'' \preceq a_1.\tau_p$
$c_2.Q \downarrow c_2.\tau_q$ and $\tau'.\tau'' \sqsubseteq_e c_2.\tau_q$, $\tau' \sqsubseteq_e c_2$, $\tau'' \sqsubseteq_e \tau_q$

$\Rightarrow$ $\{\!|a_1.P \downarrow a_1.\tau_p, Q \downarrow \tau_q$ and let τ_{pq} be a trace, $(a_1.P \sqcap_e Q) \downarrow \tau_{pq}$ and Lemma 5.7.6 $|\!\}$

$\tau'' \preceq \tau_{pq}$

$\Rightarrow$ $\{\!|\sigma_\partial \in \Gamma^e_{(a_1.P,Q)}$ and $Q \downarrow \tau_q$ $|\!\}$

$\tau'' \preceq \tau_{pq} \preceq \tau_q \sigma_\partial$

$\Rightarrow$ $\{\!|$Transitivity of $\preceq$ $|\!\}$

$\tau'' \preceq \tau_q \sigma_\partial$ (β)

$\Rightarrow$ $\{\!|\preceq \subseteq \sqsubseteq_e$ $|\!\}$

$\tau'' \sqsubseteq_e \tau_q \sigma_\partial$

$\Rightarrow$ $\{\!|\tau'' \preceq a_1.\tau_p$ and $\mathcal{V}(a_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect $a_1.\tau_p$ $|\!\}$

$\tau'' \sqsubseteq_e \tau_q \sigma_\partial$, $\tau'' \preceq (a_1.\tau_p)\sigma_\partial$

$\Rightarrow$ $\{\!|(a_1.P)\sigma_\partial \downarrow (a_1.\tau_p)\sigma_\partial, Q\sigma_\partial \downarrow \tau_q \sigma_\partial$,
let τ'_{pq}, be a trace $\left(\underline{(a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial}\right) \downarrow \tau'_{pq}$ and Lemma 5.7.6 $|\!\}$

$\tau'' \preceq \tau'_{pq}$ (γ)

$(\beta) : \tau'' \preceq \tau_q \sigma_\partial$

$\Rightarrow$ $\{\!|\tau = \tau'.\tau'' \sqsubseteq_e c_2.\tau_q$ and Lemma 5.7.7 $|\!\}$

$\tau' \sqsubseteq_e c_2 \sigma_\partial$

$\Rightarrow$ $\{\!|c_2\sigma_\partial$ should contain only non free variables, otherwise we can consider it as ϵ $|\!\}$

$\tau' \preceq c_2 \sigma_\partial$

$\Rightarrow$ $\{\!|(\gamma) : \tau'' \preceq \tau'_{pq}$ and Lemma 5.7.5 $|\!\}$

$\tau'.\tau'' \preceq c_2\sigma_\partial.\tau'_{pq}$

$\Rightarrow$ $\{\!|$Let $\tau'_r = c_2\sigma_\partial.\tau'_{pq}$, $\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big){\downarrow}\tau'_{pq}$ and $R = \sum_{\sigma_\partial \,\in\, \Gamma_{(a_1P,Q)}} c_2\sigma_\partial.\big((a_1.P)\sigma_\partial \sqcap_e Q\sigma_\partial\big)$ $|\!\}$

$\tau = \tau'.\tau'' \preceq \tau'_r$ and $R{\downarrow}\tau'_r$

if $R'{\downarrow}\tau$ then $\exists\tau'_r$, such that: $R{\downarrow}\tau'_r$ and $\tau \preceq \tau'_r$. $\quad (3_a)$

For (3_b), we can prove it by repeating the above steps.

(3_a) and $(3_b) \Rightarrow (3)$ is true.
$(1), (2)$ and $(3) \Rightarrow (b)$ is true

In c), for each variable $\Gamma_{(P,Q)}$, we let $\Gamma_{(P,Q)} = T(\Gamma, (P \sqcap_e Q){\Downarrow})$.
Suppose $R = \sum_{\sigma_\partial \,\in\, \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)$
based on the definition of gcf_e, we need to prove that the three following statements are true.

1. $R \preceq c_1.P$.
2. $R \sqsubseteq_e a_2.Q$ and
3. For all R' such that $R' \preceq c_1.P$ and $R' \sqsubseteq_e a_2.Q$, we have $R' \preceq R$.

(1) To prove $R \preceq c_1.P$, we need to prove the following:

$\forall\tau \in \Sigma^*$, if $R{\downarrow}\tau$ then $\exists\tau' \in \Sigma^*$, such that:$c_1.P{\downarrow}\tau'$ and $\tau \preceq \tau'$. $\quad (1_a)$
$\forall\tau \in \Sigma^*$, if $R \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$c_1.P \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. $\quad (1_b)$

For (1_a), let τ be a trace and $R{\downarrow}\tau$

$\Rightarrow$ $\{\!|R = \sum_{\sigma_\partial \,\in\, \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)$ $|\!\}$

$$\Big(\sum_{\sigma_\partial \,\in\, \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)\Big){\downarrow}\tau$$

$\Rightarrow$ $\{\!|(R_l^+), (R_r^+)$ from Table 5.2 $|\!\}$

$\exists\sigma_\partial \in \Gamma_{(P,a_2.Q)}$, such that:$[c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)]{\downarrow}\tau$

(α_1) When $\tau = c_1\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau' = c_1$, $\sigma_\partial \in \Gamma_{(P,a_2.Q)}$, $\mathcal{V}(c_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect c_1 $|\!\}$

$c_1.P{\downarrow}\tau'$ and $\tau \preceq \tau'$

(α_2) When $\tau \neq c_1\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau = c_1\sigma_\partial.\tau_1$, $[c_1\sigma_\partial.(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial)]\downarrow\tau$ and $(R_l^\cdot)$, $(R_r^\cdot)$ from Table 5.2 $|\!\}$

$(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial)\downarrow\tau_1$

$\Rightarrow$ $\{\!|(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial) \preceq P\sigma_\partial$ and Definition 5.4.4 $|\!\}$

$\exists\tau_p \in \Sigma^*$, such that: $P\sigma_\partial\downarrow\tau_p\sigma_\partial$ and $\tau_1 \preceq \tau_p\sigma_\partial$

$\Rightarrow$ $\{\!|\mathcal{V}(c_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect τ_p $|\!\}$

$\tau_1 \preceq \tau_p$

$\Rightarrow$ $\{\!|$Rule 2 of Definition 5.4.2 $|\!\}$

$\tau = c_1\sigma_\partial.\tau_1 \preceq c_1.\tau_p$

$\Rightarrow$ $\{\!|$Let $\tau' = c_1.\tau_p$ and Definition 5.4.5 $|\!\}$

$\exists\tau' \in \Sigma^*$, such that: $c_1.P\downarrow\tau'$ and $\tau = c_1\sigma_\partial.\tau_1 \prec \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall\tau \in \Sigma^*$, if $R\downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_1.P\downarrow\tau'$ and $\tau \preceq \tau'$. (1_a)

We can prove (1_b) by repeating the above steps.

(1_a) and $(1_b) \Rightarrow$ (1) is true:

(2) To prove $R \sqsubseteq_e a_2.Q$, we need to prove the following:

$\forall\tau \in \Sigma^*$, if $R\downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_2.Q\downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$. (2_a)

$\forall\tau \in \Sigma^*$, if $R \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$a_2.Q \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \sqsubseteq_e \tau'$. (2_b)

For (2_a), let τ be a trace and $R\downarrow\tau$.

$\Rightarrow$ $\{\!|R = \sum_{\sigma_\partial \in \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial)$ $|\!\}$

$$\Big(\sum_{\sigma_\partial \in \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial)\Big)\downarrow\tau$$

$\Rightarrow$ $\{\!|(R_l^+)$, (R_r^+) from Table 5.2 $|\!\}$

$\exists\sigma_\partial \in \Gamma_{(P,a_2.Q)}$, such that:$[c_1\sigma_\partial.(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial)]\downarrow\tau$

(α_1) When $\tau = c_1\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau' = \epsilon$ $|\!\}$

$a_2.Q\downarrow\tau'$ and $\tau \preceq \tau'$

(α_2) When $\tau \neq c_1\sigma_\partial$

$\Rightarrow$ $\{\!|$Let $\tau = c_1\sigma_\partial.\tau_1$, $[c_1\sigma_\partial.(\underline{P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial})]\Downarrow\tau$ and $(R_l^\cdot)$, $(R_r^\cdot)$ from Table 5.2 $|\!\}$

$(\underline{P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial})\Downarrow\tau_1$

$\Rightarrow$ $\{\!|(\underline{P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial}) \sqsubseteq_e (a_2.Q)\sigma_\partial$ and Definition 5.4.5 $|\!\}$

$\exists\tau_q \in \Sigma^*$, such that: $(a_2.Q)\sigma_\partial\Downarrow(a_2.\tau_q)\sigma_\partial, Q\Downarrow\tau_q$ and $\tau_1 \sqsubseteq_e (a_2.\tau_q)\sigma_\partial$

$\Rightarrow$ $\{\!|\sigma_\partial \in \underline{\Gamma_{(P,a_2.Q)}} \Longrightarrow \tau_1 \preceq a_2.\tau_q$ $|\!\}$

$\exists\tau_q \in \Sigma^*$, such that: $(a_2.Q)\sigma_\partial\Downarrow(a_2.\tau_q)\sigma_\partial$ and $\tau_1 \preceq a_2.\tau_q$

$\Rightarrow$ $\{\!|$Lemma 5.7.5 $|\!\}$

$c_1\sigma_\partial.\tau_1 \preceq a_2.\tau_q$

$\Rightarrow$ $\{\!|$Let $\tau' = a_2.\tau_q$ and Definition 5.4.5 $|\!\}$

$\exists\tau' \in \Sigma^*$, such that: $a_2.Q\Downarrow\tau'$ and $\tau = c_1\sigma_\partial.\tau_1 \sqsubseteq_e \tau'$

$(\alpha_1), (\alpha_2)$

$\Rightarrow$

$\forall\tau \in \Sigma^*$, if $\underline{R}\Downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$a_2.Q\Downarrow\tau'$ and $\tau \sqsubseteq_e \tau'$. (2_a)

We can prove (2_b) by repeating the above steps.

(2_a) and (2_b) $\Rightarrow$ (2) is true.

(3) For all $\underline{R'}$ such that $\underline{R'} \preceq c_1.P$ and $\underline{R'} \sqsubseteq_e a_2.Q$, we have $\underline{R'} \preceq \underline{R}$. To prove $\underline{R'} \preceq \underline{R}$, we need to prove the following:

$\forall\tau \in \Sigma^*$, if $\underline{R'}\Downarrow\tau$ then $\exists\tau' \in \Sigma^*$, such that:$\underline{R}\Downarrow\tau'$ and $\tau \preceq \tau'$. (3_a)
$\forall\tau \in \Sigma^*$, if $\underline{R'} \overset{\tau}{\twoheadrightarrow} 1$ then $\exists\tau' \in \Sigma^*$, such that:$\underline{R} \overset{\tau'}{\twoheadrightarrow} 1$ and $\tau \preceq \tau'$. (3_b)

For (3_a), let τ be a trace and $\underline{R'}\Downarrow\tau$

$\Rightarrow$ $\{\!|\underline{R'} \preceq c_1.P$, $\underline{R'} \sqsubseteq_e a_2.Q$ and Definition 5.4.5 $|\!\}$

$\exists\tau_p \in \Sigma^*$, such that: $c_1.P\Downarrow c_1.\tau_p$ $P\Downarrow\tau_p$ and $\tau \preceq c_1.\tau_p$
$\exists\tau_q \in \Sigma^*$, such that: $a_2.Q\Downarrow a_2.\tau_q$ $Q\Downarrow\tau_q$and $\tau \sqsubseteq_e a_2.\tau_q$

$\Rightarrow$ $\{\!|$let $\tau = \tau'.\tau''$, $\tau' \preceq c_1$ $\tau'' \preceq \tau_p$ and $\tau' \sqsubseteq_e \epsilon, \tau'' \sqsubseteq_e a_2.\tau_q$ $|\!\}$

$c_1.P\Downarrow c_1.\tau_p$ and $\tau'.\tau'' \preceq c_1.\tau_p$, $\tau' \preceq c_1$, $\tau'' \preceq \tau_p$
$a_2.Q\Downarrow a_2.\tau_q$ and $\tau'.\tau'' \sqsubseteq_e a_2.\tau_q$, $\tau' \sqsubseteq_e \epsilon$, $\tau'' \sqsubseteq_e a_2.\tau_q$

$\Rightarrow$ $\{\!|P\Downarrow\tau_p, a_2.Q\Downarrow(a_2.\tau_q)$, let τ_{pq} be a trace such that $(P \sqcap_e a_2.Q)\Downarrow\tau_{pq}$ and Lemma 5.7.6 $|\!\}$

$\tau'' \preceq \tau_{pq}$

$\Rightarrow$ $\{\!|\sigma_\partial \in \Gamma_{(P,a_2.Q)}$ and $(a_2.Q)\downarrow(a_2.\tau_q)$ $|\!\}$

$\tau'' \preceq \tau_{pq} \preceq (a_2.\tau_q)\sigma_\partial$

$\Rightarrow$ $\{\!|$Transitivity of $\preceq$ $|\!\}$

$\tau'' \preceq (a_2.\tau_q)\sigma_\partial$ (β)

$\Rightarrow$ $\{\!|\preceq \subseteq \sqsubseteq_e$ $|\!\}$

$\tau'' \sqsubseteq_e (a_2.\tau_q)\sigma_\partial$

$\Rightarrow$ $\{\!|\tau'' \preceq \tau_p$ and $\mathcal{V}(c_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect τ_p $|\!\}$

$\tau'' \preceq (a_2.\tau_q)\sigma_\partial, \tau'' \preceq \tau_p\sigma_\partial$

$\Rightarrow$ $\{\!|P\sigma_\partial\downarrow\tau_p\sigma_\partial, (a_2.Q)\sigma_\partial\downarrow(a_2.\tau_q)\sigma_\partial$, let τ'_{pq} be a trace such that $\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)\downarrow\tau'_{pq}$ and Lemma 5.7.6 $|\!\}$

$\tau'' \preceq \tau'_{pq}$ (γ)

$\tau' \preceq c_1$

$\Rightarrow$ $\{\!|\mathcal{V}(c_1.P) \subseteq N_v(e) \Rightarrow \sigma_\partial$ will not affect c_1 $|\!\}$

$\tau' \preceq c_1\sigma_\partial$

$\Rightarrow$ $\{\!|(\gamma) : \tau'' \preceq \tau'_{pq}$ and Lemma 5.7.5 $|\!\}$

$\tau'.\tau'' \preceq c_1\sigma_\partial.\tau'_{pq}$

$\Rightarrow$ $\{\!|$Let $\tau'_r = c_1\sigma_\partial.\tau'_{pq}$, $\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)\downarrow\tau'_{pq}$ and $\underline{R} = \sum_{\sigma_\partial \in \Gamma_{(P,a_2.Q)}} c_1\sigma_\partial.\big(P\sigma_\partial \sqcap_e (a_2.Q)\sigma_\partial\big)$ $|\!\}$

$\tau = \tau'.\tau'' \preceq \tau'_r$ and $\underline{R}\downarrow\tau'_r$

if $\underline{R'}\downarrow\tau$ then $\exists\tau'_r$, such that $\underline{R}\downarrow\tau'_r$ and $\tau \preceq \tau'_r$. (3_a)

For (3_b), we can prove it by repeating the above steps.

(3_a) and $(3_b) \Rightarrow (3)$ is true.
$(1), (2)$ and $(3) \Rightarrow (c)$ is true

(d) The proof is similar to (b) and (c).

$\square$

5.8 Conclusion

This chapter presents a formal and automatic approach that enforces security policies on untrusted programs. We targeted security policies written in the $\mathcal{V}LTL$ logic and programs specified on a C-Like programming language. Security policies and programs are transformed to processes in $\mathcal{E}BPA^{*}_{0,1}$, a rich process algebra that handle tests, variables and environments. Then, we explain why our main problem can be turned into computing the intersection between processes. After that, we give an algorithm that computes the expected intersection by resolving linear systems.

Chapter 6

FASER (Formal and Automatic Security Enforcement by Rewriting by algebra)

6.1 Prototype for $BPA^*_{0,1}$

We have developed a prototype that implements the algorithm that enforce policy on processes in $BPA^*_{0,1}$. As shown by Figure 6.1, in the input page, users can write down a program P and a security policy Q according to our $BPA^*_{0,1}$ syntax. Some examples are also shown in this page. After clicking the "go" button, the gcf of P and Q will be computed according to our algorithm and this will be displayed in the result page as shown by Figure 6.2. The linear system we generated during the algorithm will also be displayed in this result page. The prototype is programmed using Java and JSP. And we choose Tomcat 7.0 for web-server, since it is also a web application.

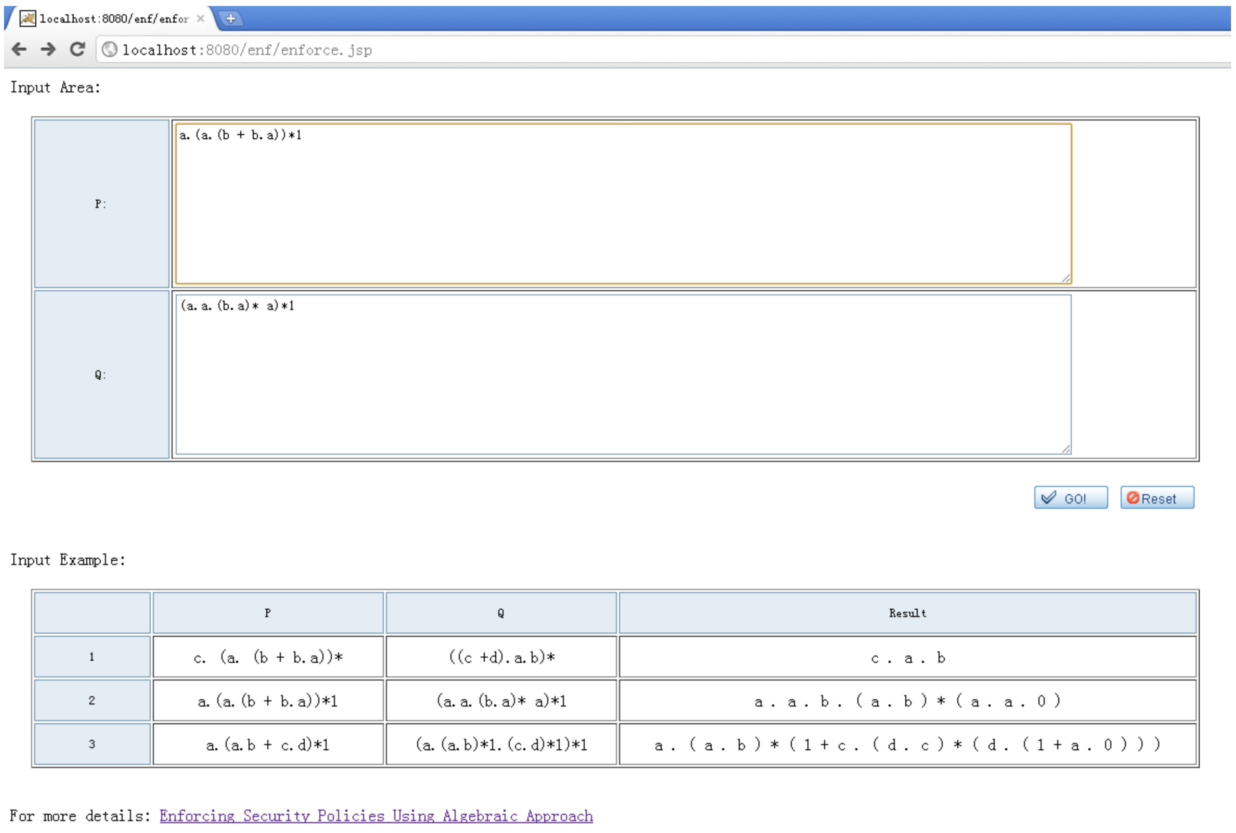
6.2 Prototype for $\mathcal{E}BPA^*_{0,1}$ with the $\mathcal{V}LTL$ logic

We developed, using JAVA and JSP, a prototype that implement our approach for $\mathcal{E}BPA^*_{0,1}$ with the $\mathcal{V}LTL$ logic. It is a web application and available online at [1]. We call this prototype FASER (Formal and Automatic Security Enforcement by Rewriting).

As shown in Figure 6.3, the prototype allows users to input programs in C-Like language and security policies using the $\mathcal{V}LTL$ logic. After clicking the "OK" button, the enforcement result and the intermediate calculus are shown like in Figure 6.4.

Also, this prototype allows users to generate a latex file containing the enforcement, including detailed steps, by clicking on *Save Result in Latex File* button as shown in Figure 6.5.

Programs can also be specified using the $\mathcal{E}BPA^*_{0,1}$ algebra. as shown in Figure 6.6.



Input Area:

P:	a.(a.(b + b.a))*1
Q:	(a.a.(b.a)* a)*1

Input Example:

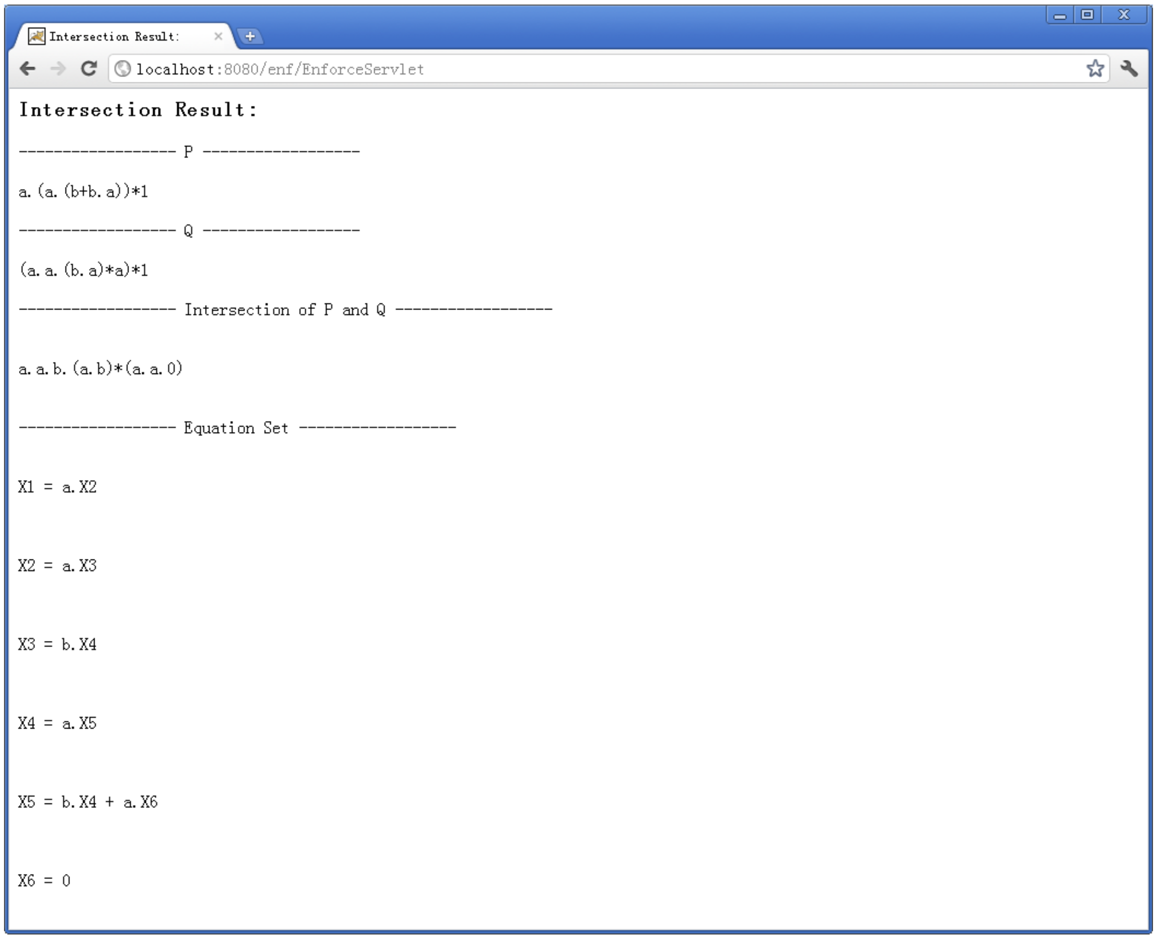
	P	Q	Result
1	c. (a. (b + b.a))*	((c +d).a.b)*	c . a . b
2	a.(a.(b + b.a))*1	(a.a.(b.a)* a)*1	a . a . b . (a . b) * (a . a . 0)
3	a.(a.b + c.d)*1	(a.(a.b)*1.(c.d)*1)*1	a . (a . b) * (1 + c . (d . c) * (d . (1 + a . 0)))

For more details: Enforcing Security Policies Using Algebraic Approach

Figure 6.1: Prototype Input Page.

6.3 Conclusion

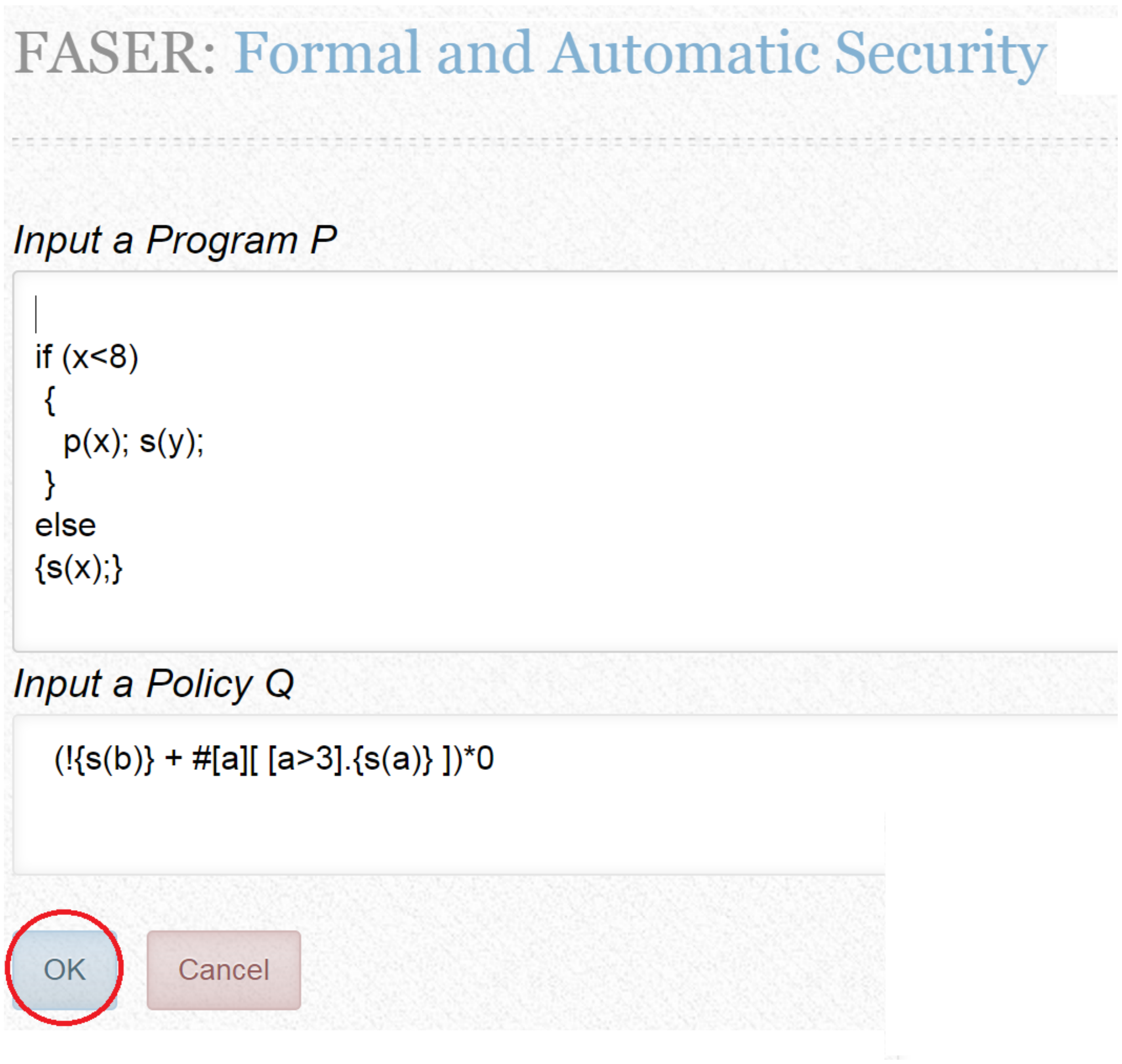
This chapter gives a short overview about FASER, the prototype that implement our approach. It is developed in Java and has a web interface. It allows end users to input their process either in $BPA^*_{0,1}$, $\mathcal{E}BPA^*_{0,1}$ or C-Like language, their security policies in either $BPA^*_{0,1}$, $\mathcal{E}BPA^*_{0,1}$ or the $\mathcal{V}LTL$ logic and shows the results together with the details of the intermediary steps in a text format or in a latex format.


Intersection Result:

localhost:8080/enf/EnforceServlet

```
Intersection Result:

------------------ P ------------------

a.(a.(b+b.a))*1

------------------ Q ------------------

(a.a.(b.a)*a)*1

------------------ Intersection of P and Q ------------------

a.a.b.(a.b)*(a.a.0)

------------------ Equation Set ------------------

X1 = a.X2

X2 = a.X3

X3 = b.X4

X4 = a.X5

X5 = b.X4 + a.X6

X6 = 0
```

Figure 6.2: Prototype Result Page.


FASER: Formal and Automatic Security

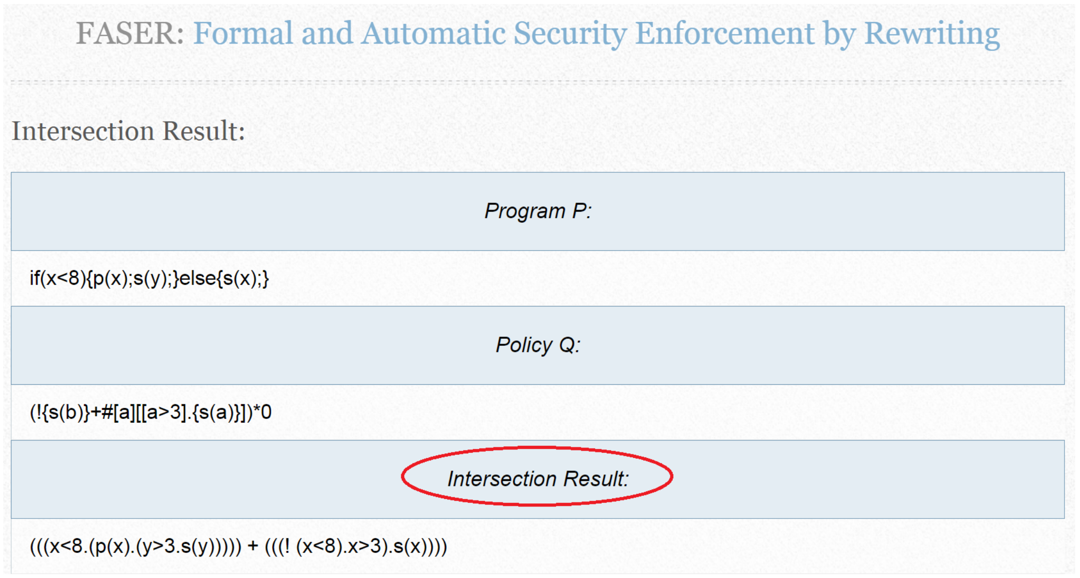
Input a Program P

```
if (x<8)
 {
   p(x); s(y);
 }
else
{s(x);}
```

Input a Policy Q

```
(!{s(b)} + #[a][ [a>3].{s(a)} ])*0
```

Figure 6.3: Prototype Input Page.



FASER: Formal and Automatic Security Enforcement by Rewriting

Intersection Result:

Program P:

```
if(x<8){p(x);s(y);}else{s(x);}
```

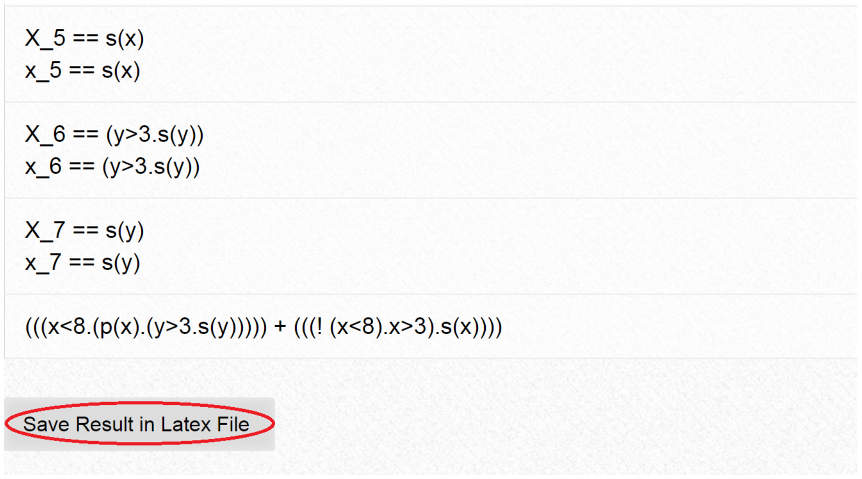
Policy Q:

```
(!{s(b)}+#[a][[a>3].{s(a)}])*0
```

Intersection Result:

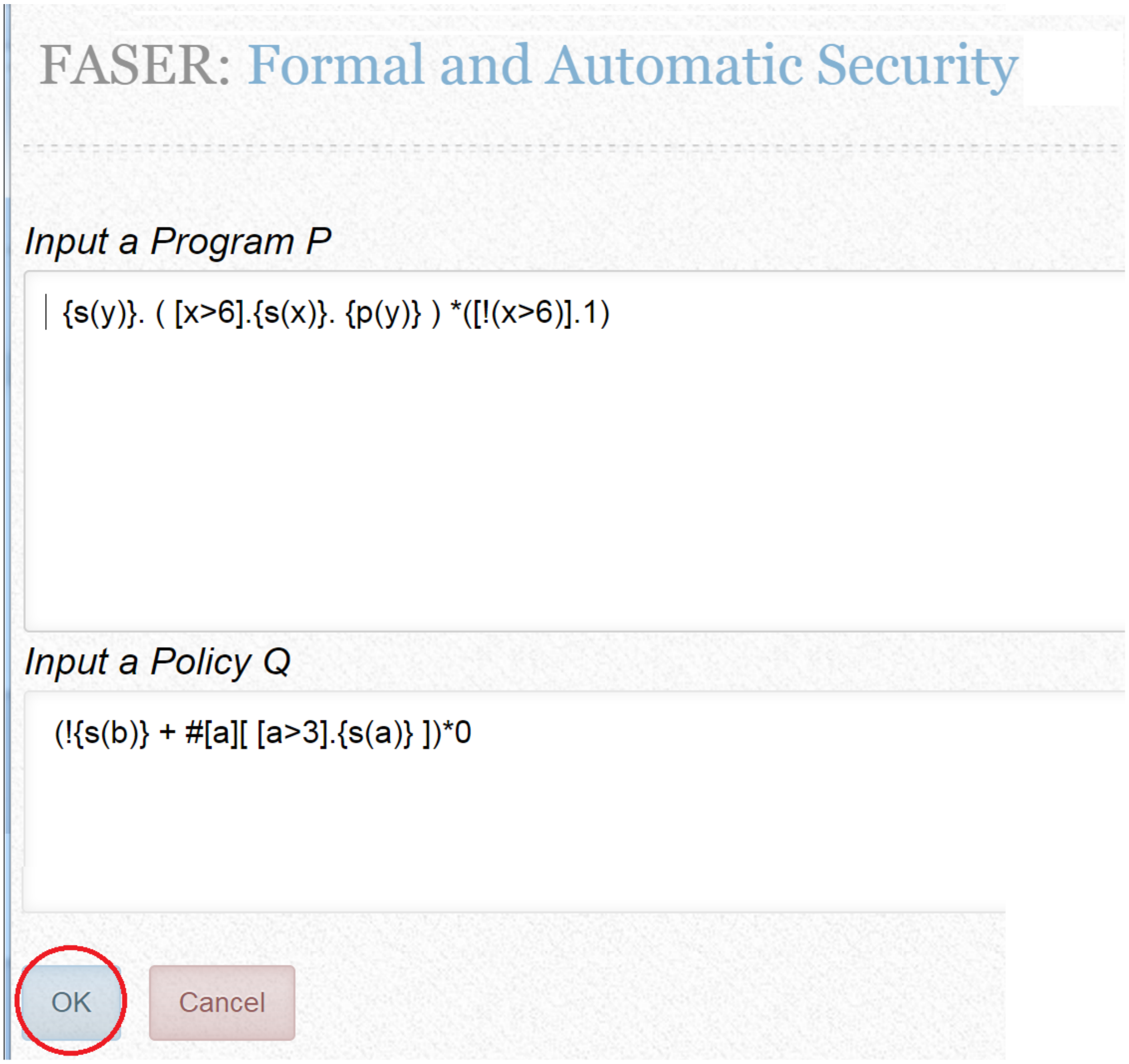
```
(((x<8.(p(x).(y>3.s(y))))) + (((! (x<8).x>3).s(x))))
```

Figure 6.4: Prototype Result Page.



```
X_5 == s(x)
x_5 == s(x)

X_6 == (y>3.s(y))
x_6 == (y>3.s(y))

X_7 == s(y)
x_7 == s(y)

(((x<8.(p(x).(y>3.s(y))))) + (((! (x<8).x>3).s(x))))
```

Figure 6.5: Prototype Latex File.



```
{s(y)}. ( [x>6].{s(x)}. {p(y)} ) *([!(x>6)].1)
```

```
(!{s(b)} + #[a][ [a>3].{s(a)} ])*0
```

Figure 6.6: Prototype Input Page with algebra.

Chapter 7

Conclusion and Perspectives

Nowadays, millions of computer are used to improve the quality of our lives, but the vast majority of them are threatened by security breaches. Financial losses are not only the possible security attacks consequences, human losses are also possible even for non-military staff (for example: vehicle, aviation systems). Therefore, we proposed a formal and automatic approach to enforce security policies on sequential programs based on rewriting technique. More specifically, given a program P and a security policy Q, we generated an intersection program P' that respects the security policy and behaves like P except that it stops any execution path whenever the enforced security policy is about to be violated. Aspect Orient Programming (AOP) also is a very important feature for our approach, it allows the end users to significantly reduce the maintenance cost and increase the modularity of systems.

Since our approach proposes an enforcement approach for security policies, it is important to understand which security policies can be enforced and by which mechanism. Therefore, based on some well-established fundamental results, we discuss the enforcement techniques and the classes of enforceable security policies in the state of art.

The goal of this thesis is to extend the work presented in [48] so that we can address more interesting security policies and interesting real programming languages. To this end, we started by understanding the technique presented in [48] and improving its foundation. For that reason we did, as mentioned in the chapter 5, the proof of all the stated properties, something that does not exist in the original work. Then, we update the definition of the trace-based equivalence so that it becomes a congruence relation, an important feature that allows us to analysis system in a modular way. Also, we extended $BPA^*_{0,1}$ algebra to $CBPA^*_{0,1}$ such that we can specify programs and policies with conditions. With this more expressive algebra, we update our approach accordingly.

To make our approach more expressive so that it can handle a real programming language, we extended our algebra to $\mathcal{E}BPA^*_{0,1}$, which takes variables, environment and conditions into consideration. We also adapt our algorithm to address these new problems.

Finally, we introduce FASER, the prototype that implement our approach. It is developed in Java and is a web application.

As future work, we are interested by the following directions:

1. **Equational Unification:** In the current work, when we compose two actions a and b to compute their most general unifier (mgu), we do not consider any theory. For example, $x = x + 2$ and $x = x + 1 + 1$ do not have a common mgu if we do not consider a theory containing an equation like $1 + 1 = 2$. In the future works, we want to have more flexibility when computing mgu by considering some theories.

2. **Program Edition:** Another interesting feature that we are willing to investigate is to have the possibility to modify the program behaviour when some actions are found or missing. Within the current version of the approach, we can only stop the program or limit the execution of some actions by some inserted conditions. But, it is not possible to substitute some part of the analysed program by another. More precisely, within the current version, the result of $a.P \sqcap b.Q$ is $mgu(a, b).(P \sqcap Q)$ which involve that $a.P \sqcap a.Q$ is $a.(P \sqcap Q)$. But we cannot specify that $a.P \sqcap a.Q$ will be $c.(P \sqcap Q)$ which is very useful behaviour to enforce some web application security policies, such that SQL injection. To handle this kind of edition we can consider each action of a security policy as a tuple $(b, Q', Q")$ and when we make the intersection $a.P \sqcap (b, Q', Q").Q$, the result is $Q'.(P \sqcap Q)$ if $mgu(a, b)$ exist and $Q".(P \sqcap Q)$ otherwise. By adding program edition into our approach, it enhancing our ability to solve more challenging security issues.

3. **PSL:** PSL (Property Specification Language) is a standard language accepted by industry to specify temporal properties of systems. In this part, we will try to extract an appropriate sub-language of PSL to specify properties that will be enforced by our techniques.

4. **Prototype Improvement:** We will apply the approach to more useful languages such as PHP to enforce web security policies. We will also address the amelioration of the prototype to bring it to professional level.

Bibliography

[1] Faser official website. Oct, 2014. `http://web_security.fsg.ulaval.ca:8080/enf/enforce-cbpa-program.jsp`.

[2] A. B. Romanowska A. Mućka and J. D. H. Smith. Many-sorted and single-sorted algebras. *Algebra universalis*, 69, Issue 2, pp 171-190, 2013.

[3] B. Alpern and Schneider F. Defining liveness. *Inf Proc Lett*, 1985.

[4] B. Alpern and Schneider F. Recognizing saftey and liveness. *Distributed Computing*, pages 2:117–126, 1987.

[5] V. Antimirov. Partial derivatives of regular expressions and finite automata constructions. pages 155:291–319, 1995.

[6] D. N. ARDEN. Delayed logic and finite state machines. pages 1–35., U. of Michigan Press, 1960.

[7] M. A. Armstrong. Basic topology. Springer, ISBN 0-387-90839-0, 1983.

[8] J.C.M. Baeten. A brief history of process algebra. volume 335 (2–3), page 131–146, 2005.

[9] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. *Report IW 206 , Mathematisch Centrum*, 1982.

[10] J. A. Bergstra and J. W. Klop. The algebra of recursively defined processes and the algebra of regular processes. *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 82–94, 1984.

[11] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[12] J. Somesh L. Yuan C. Edmund, G. Orna and V. Helmut. Counterexample-guided abstraction refinement. volume 1855: 154, 2000.

[13] A. Chudnov and D. A. Naumann. Information flow monitor inlining. pages 200–214, July 17-19, 2010.

[14] E. M. Clarke and B. H. Schlingloff. Model checking. *Handbook of Automated Reasoning*, pages 1635–1790, 2001.

[15] M. R. Clarkson and Schneider F. Hyperproperties. *Journal of Computer Security - 7th International Workshop on Issues in the Theory of Security (WITS'07)*, 18(Issue 6), September, 2010.

[16] P. Cousot. Program analysis: the abstract interpretation perspective. *ACM Comput. Surv*, page 165, 1996.

[17] M. Nouh V. Lima M. Debbabi L. Wang M. Pourzandi D. Mouheb, C. Talhi. Aspect-oriented modeling for representing and integrating security concerns in uml. pages 197–213, 2010.

[18] P. Deutsch and C. Grant. A flexible measurement tool for software systems. *Information Processing 71, Proceedings of the IFIP Congress*, 1, 1971.

[19] S. Dolan. Fun with semirings: a functional pearl on the abuse of linear algebra. *ACM SIGPLAN Notices - ICFP '13*, 48 Issue 9 Pages 101-110, September 2013.

[20] S. Eilenberg. Automata, languages and machines. volume Vol A. Academic Press, 1974.

[21] U. Erlingsson and F. Schneider. Irm enforcement of java stack inspection. pages 246–255, Oakland, California, 2000.

[22] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, pages 87–95. ACM Press, 2000.

[23] K. Crary G. Morrisett, D. Walker and N. Glew. From system f to typed assembly language. volume 21(3): 527-568, 1999.

[24] M. Mejri G. Sui and H. Ben Sta. Faser (formal and automatic security enforcement by rewriting): An algebraic approach. *Computational Intelligence for Security and Defence Applications (CISDA)*, 2012.

[25] P. Gastin and D. Oddoux. Fast ltl to büchi automata translation. pages 53–65, July 2001.

[26] J. A. Goguen and J. Meseguer. Security policies and security models. pages 11–20, Apr, 1982.

[27] D. Grossman and J. G. Morrisett. Scalable certification for typed assembly language. *TIC '00: Third International Workshop on Types in Compilation*, pages 117–146, 2001.

[28] R. Khoury H. Chabot and N. Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers & Security*, 30(4): 194-207, 2011.

[29] M. Mejri H. Ould-Slimane and K. Adi. Using edit automata for rewriting-based security enforcement. *DBSec*, pages 175–190, 2009.

[30] M. Mejri H. Ould-Slimane and K. Adi. Enforcing more than safety properties by program rewriting: Algebraic foundations. Edinburgh, UK, 2010, 2010.

[31] L. Bauer J. Ligatti and D. Walker. Edit automata: enforceable mechanisms for run-time security policies. *Int. J. Inf. Secur*, (2-16):2:117–126, 2005.

[32] L. Bauer et D. Walker J. Ligatti. Enforcing non-safety security policies with program monitors. volume 3679 of Lecture Notes in Computer Science, pages 355-373, 2005.

[33] L. Bauer et D. Walker J. Ligatti. Edit automata: Enforcement mechanisms for run-time security policies. volume 4(1-2):2-16, 26 Oct 2004.

[34] G. Morrisett K. Hamlen and F. Schneider. Computability classes for enforcement mechanisms. pages 175–205. ACM, January, 2006.

[35] R. Khoury. Enforcing security policies with monitors. Ph.D. thesis, Université Laval, 2011.

[36] R. Khoury. Symbolic analysis of assembly traces: Lessons learned and perspectives. Montreal, Qc, Canada, March 2015.

[37] R. Khoury and N. Tawbi. Corrective enforcement of security policies. *Formal Aspects in Security and Trust*, pages 176–190, 2010.

[38] Mendhekar. A Maeda. C Lopes. C Loingtier. J. M Kiczales. G, Lamping. J and Irwin. J. Aspect-oriented programming. volume LNCS 1241: 220–242. doi:10.1007/BFb0053381. ISBN 3-540-63089-9. CiteSeerX: 10.1.1.115.8660, 1997.

[39] J. C. King. Symbolic execution and program testing. volume 19, no. 7, page 385–394, 1976.

[40] S. C. Kleene. Representation of events in nerve nets and finite automata. pages 3–41, Princeton University Press, 1956.

[41] J. Ligatti L. Bauer and D. Walker. More enforceable security policies. Copenhagen, Denmark, July 2002.

[42] Turing. A. M. On computable numbers, with an application to the entscheidungsproblem. volume 42. series 2. 230–265, 1936.

[43] M. Mejri M. Langar and K. Adi. Formal enforcement of security policies on concurrent systems. *J. Symb. Comput*, 46(9): 997-1016, 2011.

[44] M. Koleini K. K. Micinski M. N. Rabe M. R. Clarkson, B. Finkbeiner and C. Sánchez. Temporal logics for hyperproperties. pages 265–284, April, 2014.

[45] A. Martelli and U. Montanari. An efficient unification algorithm. volume 4 Issue 2, pages 258–282, NY, USA, April 1982. ACM New York.

[46] J. McLean. A general theory of composition for a class of "possibilistic" properties. pages 53–67., 1996.

[47] K. L. McMillan. Symbolic model checking. ISBN 0-7923-9380-5, 1992.

[48] M. Mejri and H. Fujita. Enforcing security policies using algebraic approach. *SoMeT*, pages 84–98, 2008.

[49] J. Meseguer and J. A. Goguen. Order-sorted unifications. volume 8, Issue 4, Pages 383–413, October 1989.

[50] G. Necula. Proof-carring code. *Proc. ofPOPL'97*, pages 106–119, 1997.

[51] Online News. Anonymous launches massive cyber assault on israel. April, 2013. http://rt.com/news/opisrael-anonymous-final-warning-448.

[52] Online News. Premera blue cross breached, medical information exposed. Mar, 2015. http://www.reuters.com/article/2015/03/17/us-cyberattack-premera-idUSKBN0MD2FF20150317.

[53] A. Pardo. Many-sorted algebras. *Grupo de Métodos Formales Instituto de Computación Facultad de Ingenierá.*

[54] A . Pnueli. The temporal logic of programs. *FOCS, Proc. 18th IEEE Symp*, pages 46–57, 1977.

[55] M. Y. Vardi R. Gerth, D. Peled and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. pages 3–18, June 1995.

[56] K. Saul. An efficient unification algorithm. volume 16: 83-94, 1963.

[57] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1), 2000. 30–50.

[58] R. Shane. Millions of internet users hit by massive sony playstation data theft. Technical report, London: Telegraph, 2011-04-29.

[59] C. Small. Misfit: A tool for constructing safe extensible c++ systems. *the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 175–184, 1997.

[60] G. Sui and M. Mejri. Faser (formal and automatic security enforcement by rewriting) by bpa algebra with test. *International Journal of Grid and Utility Computing (IJGUC)*, 2013.

[61] J. Bacon T. Pasquier and B. Shand. Flowr. aspect oriented programming for information flow control in ruby. pages 37–48, New York, NY, USA, ISBN: 978-1-4503-2772-5, 2014.

[62] T. Takai and H. Furusawa. Monodic tree kleene algebra. In Renate A. Schmidt, editor, *RelMiCS*, volume 4136 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2006.

[63] C. Talhi. Memory constrained security enforcement. Ph.D. thesis, Université Laval, Avril 2007.

[64] R. Toledo and E. Tanter. Secure and modular access control with aspects. pages 157–170, AOSD '13,, NY, USA, 2013. ACM New York.

[65] M. Y. Vardi. Simple on-the-fly automatic verification of linear temporal logic. pages 238–266, Aug. 1996.

[66] D. Walker. A type system for expressive security policies. *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, 2000.

[67] wikipedia. 2014 celebrity photo leaks. Oct, 2014. `http://en.wikipedia.org/wiki/2014_celebrity_photo_leaks#cite_note-Sunderland-4`.