Fine-grained Information Flow for Concurrent Computation

Li, Ximeng

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Fine-grained Information Flow for Concurrent Computation

Ximeng Li

Kongens Lyngby 2015
PhD-2015-388

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
compute@compute.dtu.dk
www.compute.dtu.dk PhD-2015-388

Summary

It is essential to protect IT systems against security threats. An example would be the control of aircraft, which uses an internal network that passengers can access. It is important to ensure that malicious code on passenger equipment cannot endanger flight safety.

Information flow control is an important approach to the protection of systems against such threats. Notable examples include tainting analyses in languages such as Javascript, and program transformations on cryptographic algorithms to avoid information leakage through running time. A wide variety of techniques, including type systems and reference monitors, have been proposed in the context of programming languages and process calculi, to enforce such properties. The most widely used definitions of information flow security are noninterference-like properties.

For concurrent systems where processes communicate with each other to accomplish computational tasks, fine-grained security policies can be formulated by distinguishing between whether communication can happen, and what is communicated. As the first contribution of this PhD thesis, we formulate a noninterference-like property that takes all combinations of sensitivity levels for "whether" and "what" into consideration, emphasizing the importance of the *integrity* case where the former is more sensitive than the latter. This case captures the effect of Message Authentication Codes (MAC) and the consequence of Denial of Service (DoS) attacks. It is also proved that the property degenerates to a classical one when the two dimensions are intentionally blurred.

As the second contribution, we focus on the "what" dimension and further allow the flow policy to vary under different contents stored and communicated. This is the area of content-dependent (or conditional) information flow, which has recently been studied for sequential programs. We generalize the use and enforcement of content-dependent flow policies to *concurrent, communicating processes*. A security type system is developed, incorporating a Hoare logic component that provides approximations of the memory contents at different program points. Most proofs for the theoretical results on content-dependency are performed in the Coq proof assistant.

The third contribution of this thesis is the obtainment of *compositionality results* that support modular security analyses of computer systems.

A *multiplexer pattern* that separates sensitive and non-sensitive network traffic is used as a running example. *Whether* communications can happen is easily influenced by an attacker — attacking one of the incoming channels would suffice. In any case, the two data paths are still differentiable by the sensitivity levels of *what* is communicated. In case the destinations of messages are determined by their tagging, content-dependent policies are able to convey the correlation between the sensitivity level of a message and its tagging, and our Hoare-logic equipped type system allows a modular analysis of the overall system.

Resume

Det er nødvendigt at beskytte IT systemer mod udefra kommende sikkerhedstrusler. Et eksempel kunne være kontrolsystemet på et fly, som benytter et internt netværk, som også passagerne kan tilgå. Det er her vigtigt at sikre, at ondsindet kode på passagerernes udstyr ikke kan bringe flysikkerheden i fare.

Kontrol af informationsflowet er en vigtig tilgangsvinkel til at beskytte IT systemer mod sådanne trusler. Klassiske eksempler er brugen af "tainting" analyser i sprog som JavaScript og brugen af programtransformationer på kryptografiske algoritmer (for at undgå at variationer i tidsforbruget lækker fortrolige oplysninger). Der er et utal af teknikker, der kan sikre at programmerne overholder det ønskede informationsflow, som for eksempel brugen af type-systemer og monitorer. Den præcise definition af, hvad korrekt informationsflow egentligt er, udtrykkes normalt i form af en såkaldt "non-interference" betingelse.

For parallelle systemer, hvor processer kommunikerer med hinanden, kan man formulere meget finkornede sikkerhedspolitikker ved at skelne mellem hvorvidt der kommunikeres og hvad der (i givet fald) kommunikeres. Som det første bidrag i denne PhD afhandling formulerer vi en "non-interference" betingelse, som kan tage højde for alle fire kombinationer af sikkerhedsniveauer for "hvorvidt" og "hvad". Vi fokuser på integritet, fordi "hvorvidt" her er mere følsom end "hvad" (i modsætning til litteraturens fokus på fortrolighed). Det gør os i stand til at modellere "Message Authentication Codes (MAC)" og konsekvenserne af angreb på tilgængeligheden ("Denial of Service"). Vi viser, at en bevidst sammenblanding af "hvorvidt" og "hvad" giver anledning til en velkendt sikkerhedsegenskab.

Som det andet bidrag udvider vi vore sikkerhedspolitikker til også af afhænge af de data, der kommunikeres eller lagres. Dette område, data-afhængigt informationsflow, har hidtil kun været studeret for sekventielle programmer, og vi generaliserer det til data-afhængige politikker for parallelle og kommunikerende processer. På det tekniske plan udvikler vi et type-system, som inddrager elementer fra Hoare's aksiomatiske logik for at karakterisere data værdierne på de forskellige programpunkter. Hovedparten af korrekthedsbeviserne er blevet kontrolleret ved hjælp af Coq bevisassistenten.

Et tredje bidrag er resultater, der viser hvordan man kan opnå sikkerhed på modulær vis, ved at analysere systemet komponent for komponent.

Som gennemgående eksempel benytter vi en multiplexer, der skal holde sensitive og ikke-sensitive data adskilte. Den er følsom over for angreb på "hvorvidt" der kommunikeres, fx hvis der er angreb på en af de indgående kanaler. Vi er her i stand til at udtrykke de relevante sikkerhedspolitikker til at adskille de to kommunikationsveje gennem multiplexeren. Data-afhængige politikker er i stand til at beskrive situationer, hvor adresserne afhænger af rutningsdata. Vores typesystem udvidet med Hoare's aksiomatiske semantik tillader en modulær analyse af det samlede system.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a PhD in Computer Science.

This thesis contains a complete and consistent development of my PhD topic. The research has been performed under the supervision of Professor Flemming Nielson and Professor Hanne Riis Nielson in the period from October 2012 to September 2015.

Chapter 3 and Chapter 4 expound on part of the work published as [LNN15], and Chapter 5 elaborates on part of the work published as [LNNF15]. The joint work [NNL15b] explores a different kind of security property than that adopted in [LNNF15], but benefits the development of [LNNF15], especially in the design of the type system.

Lyngby, 30-September-2015

Ximeng Li

Acknowledgements

My gratitude goes first to my supervisor Flemming Nielson and co-supervisor Hanne Riis Nielson, who provided the opportunity of working in an interesting research topic. I enjoyed insightful discussions with them, and benefited from their excellent personal guidance in both research methodologies and presentation skills. Consequently, I attained a good focus in my work and started to value the simplicity and elegancy of results.

I would like to thank my evaluation committee — Luca Aceto, Heiko Mantel, and Christian W. Probst — for providing invaluable comments on a preliminary version of this thesis.

I am also glad to have those who shared offices or activities with me: Omar Almousa, Zaruhi Aslanyan, Lars Frydendal Bonnichsen, Laust Brock-Nannestad, Alessandro Bruni, Ann-Cathrin Dunker, Piotr Filipiuk, Marieta Georgieva Ivanova, Nicklas Bo Jensen, Erisa Karafili, Alberto Lluch Lafuente, Hugo-Andrés López, Andrea Margheri, Jan Midtgaard, Sebastian Alexander Mödersheim, Christian W. Probst, Carroline Ramli, Nataliya Skrypnyuk, Michal Terepeta, Roberto Vigo, Ender Yuksel, Kebin Zeng, and Fuyuan Zhang.

Xinyu Feng from the University of Science and Technology of China visited for two months, with whom I enjoyed a series of discussions on security-related bisimulations. Xi Wu came for a half-year external stay from East China Normal University, with whom I had a little quest into coordination languages.

My stay hosted by Andrei Sabelfeld at Chalmers University of Technology helped shape some of the ideas underlying this dissertation. I am grateful to people I met and discussed or chatted with: Andrei Sabelfeld, Willard Rafnsson, David Sands, Bart van Delft and Benjamin C. Pierce. I am also indebted to others in the same department for their social accommodation and company.

My thankfulness goes last but not least to my parents, for their support and love.

Contents

List of Tables and Figures

Chapter 1

Introduction

Information flow control aims to provide end-to-end guarantees against inadvertent information leakage/corruption in computer systems. In a system, an agent a may be granted the privilege to observe certain values/behaviors, and it is important to control the extent to which a can deduce other system-wide values/behaviors based on its observation. Similarly, a may obtain rights to modify certain values/behaviors, and it is vital to control the extent to which a can influence other system-wide values/behaviors via the modifications it is allowed to make. These are the typical issues of concern in information flow control. Compared with access control, which is focused on mediating accesses attempted at the *perimeter* of a system, the stress of information flow measures is more on restricting the propagation of the effects of such accesses *within* the system.

One of the earliest investigations into information flow problems is the security model by Bell and LaPadula [BL76], designed for operating systems. In this model, subjects (processes operating on behalf of users) and objects (files, segments, devices, etc.) are both assigned security levels. A basic principle is *no read up, no write down*, i.e., reading operations cannot be performed by a subject from a more highly protected object, and writing attempts cannot be made by a subject to a less protected object. These levels being confidentiality levels, the aim is to avoid information flow to less "secretive" subjects/objects.

Denning [Den76] addresses explicit flows arising out of assignments and implicit flows of conditionals, using lattice-based security models. The approach is applied to the certification of programs in Denning&Denning [DD77].

Information flow measures are important for safety-critical systems. A compelling example is about timing channels in RSA implementations: an attacker can deduce certain parameters (such as the key) based on its observation on timing. Language-based measures can be applied to eliminate such un-desirable

information flow [Aga00, SM03a, MS15]. The area has also gained practical impact in securing voting systems [CCM08], and in the end-to-end security enforcement in real-world programming languages like PHP [JKK06] and JavaScript [HBBS14]. In the latter direction, the FlowSafe project [AF09, Flo10] of Mozilla is a notable initiative taken in the industry.

The Common Criteria (CC, [Org12]) has "information flow control policy" as one of the Security Functional Requirements to be considered in the certification of systems. Multiple Independent Levels of Security (MILS [Rus08]) is an architecture that aims to achieve resource sharing by constructing a system in partitions, and controlled information flow between these partitions. It has a wide range of applications (e.g., [HHOAF05, AFOTH06, MPTB12, ATRS15]).

There are numerous approaches to the enforcement of information flow security — an important one is *typing*, which was initiated by Volpano et al. [VIS96]. In [VIS96], the principles of the analysis by Denning&Denning [DD77] are captured in a type system and a soundness result with respect to a noninterference property is proved. The typing approach has been studied extensively [ML97, SV98, Aga00, SS00, BC02, Smi06, CM06, BS10, ADZ+12, RHS12, MAC12, BPR13, LC15] in languages with different features.

On the other hand, the security guarantee is usually formulated as a noninterference-like property, requiring that the public parts of a system should stay invariant against variations in the confidential parts [Coh77, GM82]. For confidentiality, noninterference[1] requires that variations in confidential behaviors/contents, cannot cause variations in publicly observable behaviors/contents. This requirement is imposed by comparing different executions of the same specification/program, and mandating that these executions should have similar public behaviors/contents, despite the differences in previous secret behaviors/contents. This relational nature makes noninterference properties qualify as "hyperproperties" [CS10], in comparison to classical "properties" [AS87].

Due to the different semantic elements of state-machine-based formalisms, process calculi, and programming languages, and different application scenarios (programs, protocols, choreographies, etc.), noninterference properties can take different forms that embody similar spirits. Previous research in the formulation and "meta-properties" of noninterference properties is again extensive [SS00, FG01, SM02, Man03, FRS05, Cas07, MZ08, MS10, AM11, HS12, VDMZ13, RS14]. A key "meta-property" here is *compositionality*, that scales up the security guarantee for composites, no matter what enforcement methods are used for the security guarantee of individual components. The investigation into compositional security started a long time ago [McC87], and is still an important element in recent theoretical research (e.g., [RS14]).

[1] In a narrow sense, "noninterference" often points to the particular security condition of Goguen and Meseguer [GM82] but we have rather chosen to use the same term to refer to all possibilistic security conditions embodying similar spirits.

Today's computing systems are prevalently concurrent, distributed, and interactive. Scientific computing, banking, service booking, industrial control, traffic management — are all concerned with some of these traits, which arise along with the Internet and for the efficiency and effectiveness of an increasingly collaborative society. *Communication* between different system components has become a vital element of contemporary IT systems. Compared with the setting of sequential programs, concurrent and distributed systems admit new forms of information leakage and corruption, such as through racing behaviors, and create the need for different noninterference properties. Interactivity calls for investigation into open systems, and approaches to describing and constraining the computation environment.

Contributions In this thesis, we demonstrate how the information flow security of concurrent, communicating systems can be specified and verified in a finer-grained way than provided by previous approaches, in the following aspects:

1. The presence and content of communication can be protected separately. For integrity, in particular, communication channels with low presence integrity and high content integrity are related to the use of Message Authentication Codes (MAC) and Denial of Service (DoS) attacks.

2. Different policies can be specified and enforced under different contents of communication and memory, which is particularly meaningful in scenarios where the destinations of messages between different processes depend on their tagging.

Moreover, the fine-grained distinction (2) of contents can be largely integrated with the distinction (1) between "presence" and "content".

We define security properties and study their compositionality, which either serves as an analysis approach in itself, or facilitates soundness proofs for information flow type systems. The property proposed to support "presence" and "content" as independent dimensions is a process-algebraic property and its relation with classical process-algebraic noninterference properties is studied.

Structure In Chapter 2, we introduce some general technical background for this dissertation. In Chapter 3, we present the Quality Calculus [NNV12], in which the connection of two important means of stating noninterference results — through unwinding and in terms of self-bisimilarity — is discussed. In Chapter 4, we investigate the distinction between the "presence" and "content" of communication in the Quality Calculus. The study is carried out on *integrity*, as opposed to confidentiality, since this provides new insights vis-à-vis existing literature. In Chapter 5, we investigate the information flow security of concurrent programs, with the use of confidentiality policies that are dependent on

the content of memory and communication. The "presence" of communication is again treated as a separate dimension from "content". Comparisons are made between our proposed noninterference properties and existing ones. The conclusion is given in Chapter 6, followed with a further discussion of several elements of our development and opportunities for future work.

For the study of content-dependent security policies, we direct our attention from the Quality Calculus to a less advanced concurrent, imperative language. On the one hand, this is driven by the need of our partner Airbus in the European Artemis project SESAMO [SES] to ensure the information flow security of the software *implementation* hosted in the IT system onboard aircraft. A comparatively minor reason is: to our knowledge, most of the related work that addresses content-dependency has been performed in a language-based, rather than process-algebraic setting.

We developed (13 K lines of) proofs[2] for most of the theoretical results in Chapter 5 in the Coq proof assistant [CPA]. The Coq formalization of our language and type system, together with the statements of lemmas and theorems, are detailed in Appendix B. This presentation is generated using the "coqdoc" tool [Tea14].

[2] http://orbit.dtu.dk/files/123041334/phd388_Li_X_dif_com_coq.zip

Chapter 2

Setting the Scene

This dissertation builds on certain *common practices* in process-algebraic and langauge-based information flow control, and the area is introduced briefly in Section 2.1. A key element of our technical development will be bisimulation-based noninterference properties. We motivate the use of such properties in Section 2.2, with a slightly philosophical digression in Sub-section 2.2.1, on the manifestation of the noninterference-style thinking in a variety of areas. We then discuss the need for treating the system and the environment separately in Section 2.3. Two important elements of the environment are *strategies* and *schedulers*. After that, we compare the respective merits of different analysis techniques for information flow (Section 2.4), motivating our development of a type system for content-dependent flow policies later in Chapter 5.

2.1 Process-Algebraic and Language-Based Security

Information flow security has been studied extensively in process calculi and minimal languages derived from core calculi. This collection of calculi include the π-calculus [Pot02, HY07], CCS-like calculi [FG01, FR06, vBV11], CSP [Ros95, RH13], calculi designed for session-based communication [CCDR10], for generative communication and coordination [TNH06, Ald06], for mobility [CF01, BCF08], and so on.

Understanding information flow problems in process-calculi helps make systems security-enabled from the design phase. In addition, many operators in process calculi are well-understood, and generality is enjoyed by the information flow constraints induced by these operators.

Information flow security has also been studied extensively in programming languages (e.g., [ML97, BC02, Dam06, BS10, MC12]). So far, *downgrading* and *downgrading policies* [SM03b, CM04, CM06, MR07, LM08, SS09, MB09] have been studied more heavily for language-based security than for process-algebraic security. Downgrading is the operation that allows one to relax information flow constraints by treating a variable as having a less restrictive security class than the one originally assigned to it. Downgrading policies govern the kind of downgrading operations that are relatively safe. Hence the great amount of work in downgrading for language-based security reflects the concern in dealing with real code, thereby needing to reason about the *relative security* of code, when *strict noninterference* is broken.

A notion of timing-sensitive noninterference in language-based security has been reduced to the process-algebraic security condition PBNDC [FR06], by suitably encoding programs as processes in VSPA — a CCS-like process calculus [FRS05]. This demonstrates the existence of certain cross-language/formalism patterns in the essence of noninterference conditions, which are involved in the BNDC-like properties. In fact, this pattern will recur as the basis of the security properties studied in this thesis.

2.2 Noninterference

Before devising an analysis for information flow security, a key question to answer is:

What are secure systems?

This question is often answered by formulating a noninterference-like condition [Man11].

One important origin of noninterference is Cohen's seminal work [Coh77] inspired by information theory and cybernetics. Information flow is captured by *dependencies*, which are in turn characterized by the propagation of *varieties*. In more detail, computations are thought of as holistic "pipes" $\rhd^F$, where F corresponds to the program executed. When a certain variable x varies at one end of $\rhd^F$, thereby introducing some "perturbation", with all the other variables staying constant, the observation that another variable y also varies, at the other end of $\rhd^F$, unveils a dependency of y on x, or a flow from x to y. This immediately leads to two research tracks pursued separately ever since — a quantitative one [CHM05, Smi09], and a qualitative one [VIS96, SV98, BC02, Man03, MSS11] — in the area of information flow analysis. The *quantitative approach* studies how much of x is leaked into y, which is reflected by how heavily the variations of x cause variations of y, captured in information-theoretic terms.

The *qualitative approach*, on the other hand, requires outright that *no* variation of y should be observed under variations of x.

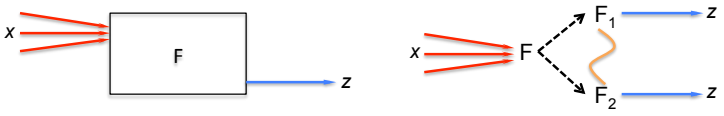


Figure 2.1: Intuition for Bisimulation-Based Noninterference

The left part of Figure 2.1 depicts the scenario where the variation in x at one end of $\triangleright^F$ *cannot* cause any variation in z at the other end of $\triangleright^F$. In this case we say that there is *no flow* from x to z, or that the value of x *does not interfere* with the value of z. A formal expression can be

$$\forall v_1, v_2, v : (\llbracket F \rrbracket [x \mapsto v_1][z \mapsto v])\, z = (\llbracket F \rrbracket [x \mapsto v_2][z \mapsto v])\, z.$$

Suppose an attacker can make observations of z at the intermediate stages of the computation F. Then the equality on z needs to be imposed at all these stages. Due to the difference in the value of x, F may well evolve into different programs F_1 and F_2: take for instance $F = \mathsf{if}\ x > 0\ \mathsf{then}\ F_1\ \mathsf{else}\ F_2$. We need to express:

1. the values of z are equal when F_1 and F_2 are reached, respectively, and
2. they will also be equal in the future derivatives of F_1 and F_2.

This further creates the need to build an association relation (illustrated by the tilde between F_1 and F_2 in the right part of Figure 2.1) on all pairs of derivatives of F where the comparison between the possible values of z needs to be performed. And we now have the motivation to use a small-step semantics and a bisimulation-based noninterference property (e.g., [SS00, FG01, Dam06, Smi06, PHN13, RS14]).

Similarly, concurrency and nondeterminacy render the most simple-minded two-run comparison flawed. Consider the following program with four processes running concurrently:

$$(x := 1 \,||\, c!2 \,||\, c!3 \,||\, c?z) \setminus \{c\}.$$

The first process assigns 1 to x, the second and the third output the values 2 and 3, respectively, over channel c, and the fourth inputs from channel c into z. It is represented by $\bullet \setminus \{c\}$ that communication over c must be a synchronization between an output and an input. In this program, there is no information leakage from x into z. However, it is not the case that comparing an arbitrary pair of executions of the program starting from different values in x, we end up with the same value in z. Because of the race between the two outputs, it is undetermined which of them actually synchronizes with the input. On the

other hand, the bisimulation-like requirement that *for all* possible executions, *there exists* an execution starting from a different value in x, leading to the same value in z, seems to match the intuition for the security of this program reasonably well.

For confidentiality, we distinguish between confidential channels/variables and public ones, and want to make sure that public content/behaviors should stay invariant under potential perturbation in confidential content/behaviors. For integrity, we distinguish between corrupt (low integrity) channels/variables and clean (high integrity) ones, and aims to guarantee that clean content/behaviors should stay unaffected under variations of corrupt content/behaviors.

It is customary to use a security lattice $\mathcal{L}$ whose elements denote the security classes of communication channels and variables. And such distinction can be represented by a partition of $\mathcal{L}$ into sets $\mathcal{U}$ and $\mathcal{D}$, where $\mathcal{D}$ is a downward-closed subset (e.g., [DP02], also formulated in Definition 2.1) of $\mathcal{L}$. In fact, requiring the downward-closedness of $\mathcal{D}$ is equivalent to requiring the upward-closedness of $\mathcal{U}$ (Lemma 2.2, whose proof is straightforward).

Definition 2.1 (Downward-closed and Upward-closed Subsets) Suppose $\mathcal{L}$ is a lattice.

1. A set $A \subseteq \mathcal{L}$ is a downward-closed subset (or down-set) of $\mathcal{L}$, if $\forall l_1, l_2 : l_2 \in A \wedge l_1 \sqsubseteq l_2 \;\Rightarrow\; l_1 \in A$.

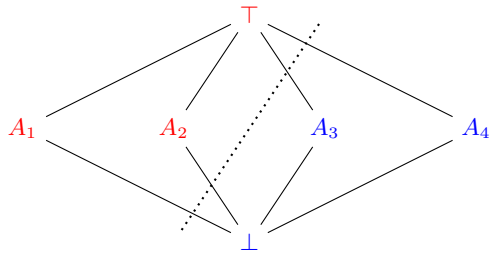
2. A set $B \subseteq \mathcal{L}$ is an upward-closed subset (or up-set) of $\mathcal{L}$, if $\forall l_1, l_2 : l_1 \in B \wedge l_1 \sqsubseteq l_2 \;\Rightarrow\; l_2 \in B$.

Lemma 2.2 *For a lattice $\mathcal{L}$ and its partition into $\mathcal{U}$ and $\mathcal{D}$, $\mathcal{D}$ is a down-set of $\mathcal{L}$ if and only if $\mathcal{U}$ is an up-set of $\mathcal{L}$.*

For confidentiality, elements (often called labels) of the set $\mathcal{D}$ usually denote low confidentiality, and elements of $\mathcal{U}$ often denote high confidentiality. For integrity, on the other hand, elements of the set $\mathcal{D}$ usually denote high integrity, and elements of $\mathcal{U}$ often denotes low integrity. Note that this is all in line with the intuition for $\mathcal{D}$ and $\mathcal{U}$ to be an up-set and a down-set, respectively, of $\mathcal{L}$. In this dissertation, we also use the term "clean" to describe entities with high integrity, and the term "corrupt" to describe entities with low integrity.

For the convenience of discussion, existing work [SV98, ZM01, BC02, Smi06, MZ08, BDG11, RS14] often uses a two-point lattice $(\{H, L\}, L \sqsubseteq H)$ where the levels H and L represent membership in $\mathcal{U}$ and $\mathcal{D}$, respectively; unless the topic of discussion is the practical meaningfulness of the elements of a specific type of $\mathcal{L}$ [ML97, CM06, MB09, SRMM12]. For confidentiality, when the two-point lattice is used, H (resp. L) thus denotes high confidentiality (resp. low confidentiality).

Remark 2.1 *Another way of distinguishing between high and low is by picking a reference element l from the lattice $\mathcal{L}$, and then regarding $l' \sqsubseteq l$ as l' being low, $l' \not\sqsubseteq l$ as l' being high (e.g., [VIS96, ZM05]). The security condition will then be stated by ranging over all such reference labels l. The approach of down-sets and up-sets presented above leads to more partitions than using a reference element does. Take the lattice below as an example, it is not difficult to verify that $\{A_3, A_4, \bot\}$ (resp. $\{A_1, A_2, \top\}$) qualifies as a down-set (resp. up-set) of it. However, it is impossible to pick $l_0 \in \{A_1, A_2, A_3, A_4, \top, \bot\}$ such that all l' satisfying $l' \sqsubseteq l_0$ constitute the set $\{A_3, A_4, \bot\}$ and all l' satisfying $l' \not\sqsubseteq l_0$ constitute the set $\{A_1, A_2, \top\}$.*



It can be seen that using down-sets and up-sets allows more freedom in regarding an element incomparable to the reference element as either high or low.

2.2.1 The Noninterference-Style Thinking at Large

The idea of noninterference embodies a kind of thinking that manifests itself in many different places.

Whereas noninterference mandates that certain elements should be resillient to potentially significant variations in certain other elements, the notion of continuity in mathematics requires that small variations of certain variables only create small variations of certain other variables. The recent work in continuity analysis [CGL10] for programs is an application of this mathematical idea of continuity in programming languages. In program analysis [NNH05], a live variable analysis provides approximative answers to the question of whether a variable may be used before being modified. The semantic characterization of a variable being *dead* is: variations of the variable cannot interfere with the future execution of the program [Nie85]. For man-made systems, fault tolerance [KK07] requires that normal system functionalities should stay unaffected under perturbation created by faulty components (e.g., [Web89]). For biological systems, "equi-finality" captures the phenomenon that certain organisms eventually develop into the same form despite the differences in their environment (e.g., [VB03]). The Turing test [Tur50] is concerned with the *non-distinguishability* of two different worlds, one involving a computer and the other a human being. The idea of having experimental groups and control groups in scientific

experiments helps identify whether a medicine, etc., has observable effects.

Going back to computer security, the notion of "robust declassification" [ZM01] in information flow control says that the decision and content of declassifications should not be influenced by changes in low integrity data. It has a more formal expression roughly saying: the fact that a system is non-interfering should stay unaffected under variations in the placement of attack commands in certain holes of the original code. Differential privacy [Dwo06] is concerned with the privacy guarantees provided by data extraction protocols for databases. In more detail, the probability for a value to be yielded by the protocol should not vary much, under all possible differences of the source database in only one element. "Semantic security" [GM84] requires that the information that an eavesdropper can compute about the cleartext should remain the same no matter if it has access to the ciphertext or not. A zero knowledge proof is a proof that reveals only the validity of the assertion proved and nothing else [GMR85]. This is formulated by saying that comparing two scenarios, one in which an agent is given the proof, the other in which it is given the (valid) assertion, the agent cannot feasibly compute more in the former scenario than in the latter. This kind of "simulation paradigm" also manifests itself in the security characterization used by Multiparty Computation (MPC, e.g., [Orl11]). In MPC, multiple parties run a distributed, decentralized protocol to compute a function of their inputs, that they intend to keep confidential against the others. An MPC protocol is deemed secure if every attack by a dishonest party can be "simulated" in an alternative, ideal world, where the protocol is replaced with an honest third party performing the computation in a centralized manner.

2.2.2 Traced-based and Knowledge-based Security Properties

The noninterference properties studied in this dissertation are bisimulation-based. In the literature, trace-based properties [GM82, Man03, RHS12] and knowledge-based properties [BDG11, AM11, AC12, vDHS15] are two other important classes of security properties.

Mantel [Man03] proposed a rather comprehensive framework of trace-based properties. There are a series of "basic security predicates" (BSP) in the framework that can be composed to form the security requirement suitable in each particular application scenario. A BSP is typically a closure property stating that inserting/removing certain confidential events in a trace results in another trace having the same public view; thus the attacker cannot deduce the non-occurrence/occurrence of these events by accessing the public view of traces. Several classical trace-based properties can also be obtained by composing BSPs.

Knowledge-based properties directly characterize the change in the knowledge possessed by the attacker as it observes the execution of a program. This

knowledge is typically in the form of a set of possible initial memories or possible input environments, and it grows when the set shrinks. "Exclusion knowledge", on the other hand, gives the set of initial memories or input environments that can be excluded given the observation; this leads to a more natural description of knowledge increase [BvDS15]. Knowledge-based properties are fairly intuitive and suitable for interpreting dynamic policies (e.g., [BS10, AC12, BvDS15]).

Trace-based properties emphasize the results of perturbations (which are the observations), whereas knowledge-based properties emphasize the causes of observations (which are the perturbations). The connection between certain properties belonging to the two respective classes has been established, e.g., by Balliu [Bal13]. On the other hand, in line with the well-known fact that bisimulation equivalence is finer than trace equivalence, bisimulation-based properties are often stronger than (and can be used to prove) their trace-based counterparts (e.g., [FG01]), although results on the coincidence of trace-based and bisimulation-based properties do exist [MS01].

2.3 Systems and their Environment

The computational systems that we consider will be specified in a process-algebraic language, or a simple programming language. The advantage of using a language (over abstract models such as state machines) is that the specification/realization of systems is eased. Using a simple programming language also provides insights to the analysis of real world code.

In a language that is expressive enough, the environment can be modeled by an ordinary process just like any other process executing in the system itself. However, the environment is often given a separate model, for example as a function having access to certain observables, for the following reasons:

1. It may be cumbersome to "program" the environment, but relatively easy to specify it in an abstract manner. Many concrete details of the environment are also unknown and need to be under-specified: an example of unknown detail is the implementation of certain APIs.

2. The environment might be of a different "nature" than that of the system; examples are where the system is benign whereas the environment is malign (e.g., an attacker with a certain profile), where the system is possibilistic, whereas the environment is probabilistic (e.g, a probabilistic scheduler), etc.

3. To get a realistic view of computation, it is desirable for the boundary between the system and the environment to be distinguished from the boundary between different processes within the system.

In this section we will discuss an important feature of the environment in information flow control, that is its *totality*. Then we provide an overview of two incarnations of the environment, *strategies* and *schedulers*, with somewhat similar forms, but different purposes. All these three elements of the environment are closely related to the development of this thesis.

2.3.1 Totality of Environment

An environment is total (e.g., [Man03]) with respect to a set of actions if at each state of computation (a process, a memory or a combination of both, depending on the language or formalism), the environment allows each action in the set to be performed. In the literature, the terms "input-enabled" (e.g., [EGP08]) and "input-complete" (e.g., [TLHL11]) often carry the same kind of meaning as "total" does.

In a communication-oriented setting, there are two aspects of totality:

- given channel c, whether communication is allowed over c, and
- the possible contents that can be received over c.

Our development will not hinge on any particular totality assumptions. Finer-grained specification of the environment is made possible through the use of strategies: one is able to specify the possible channels that the process can communicate over, and data that can be received, after a particular computation history.

2.3.2 Strategies

Strategies [WJ90, OCC06, MC12, RHS12] are one kind of abstraction of the environment process. They are introduced to provide more flexibility in the specification of environments that deviate from a total one.

In general, environment strategies are functions that make decisions on how to interact with the system, based on certain observations. We use strategies for process-algebraic systems, and the observations of our strategies are thus about the histories of computation composed of communication actions. On the other hand, the interaction decisions are expressed by a set of communication binders, that can synchronize with the active communication binders of the system, for the next transition step.

The most powerful such strategy can attempt to interact with the system in all possible ways, including blockage of communication, or feeding arbitrarily corrupted data to the system, based on any observation it has made. An

environment with this strategy is the implicit assumption of classical process-algebraic security properties like SBNDC [FG01]. A less powerful strategy is obtained by constraining this ability. This is sensible since the attackers often act under certain resource constraints in practice.

2.3.3 Schedulers

Schedulers are a similar abstraction of the environment with a particular purpose: managing the switches between different processes in the course of execution. The interaction of a scheduler with the system proceeds in a fairly fixed pattern: the scheduler decides on which process(es)[1] to execute next, the selected process(es) execute for a specified time slice, before the next such round begins.

Process-algebraic security is more in line with modeling schedulers as ordinary processes. On the other hand, dealing with schedulers as a separate abstraction is an important subject in securing concurrent programs (e.g., [SS00, Smi06, RS09, MS10]). For concurrent programs, the basic way of specifying an operational semantics is by selecting a process to execute for one step fully non-deterministically. However, a refinement of this mechanism that reduces non-determinacy can break security. Unfortunately, such reduction in non-determinacy is usually caused by the employment of a particular scheduler.

We illustrate this point with the following program taken from [BC02]. Shared variables (h and l) are used rather than communication, to simplify the scenario for the problem to be more easily perceived. Here h is confidential while l is public. It is not difficult to see that for each scheduled execution reaching its first assignment to l, there exists a schedule with different evaluation of $h = 1$ (likely to be different than the first schedule), such that the same assignment is also the first to l reached. This is the intuition for the security of the system under a fully (cooperatively) non-deterministic scheduler [SV98].

$$(\mathsf{if}\ h = 1\ \mathsf{then\ skip};\mathsf{skip\ else\ skip};\ l := 1) \mid l := 2$$

We introduce certain constraints on the scheduling, and assume a round robin scheduler with time slice 3 is used. Assume that the evaluation of each conditional, as well as the execution of each assignment, takes one time unit. Now any execution with 1 as the initial value of h will eventually assign 1 to l; whereas any execution with a different initial value than 1 in h will end up with 2 in l. Information in h has leaked into l.

There are two classes of security properties when the scheduler is considered — for a specific scheduler or for a class of schedulers. The most general kind

[1] Suppose two processes can be selected at the same time for their synchronous communication.

of properties in the latter class are called scheduler-independent security. They assert that a system is secure under the control of virtually all schedulers[2].

2.4 Analysis Techniques

The verification of information flow security can be performed in several alternative means: dynamic monitoring [Fen74], multi-execution [DP10, RS13], self-composition [TA05, BDR11], model checking [CFK+14], bisimulation checking [FG01], exploitation of compositionality [RS14], type systems [VIS96], and so on.

Secure multi-execution runs a program once for each security level. Statements producing observable outputs are only executed in the run at their own security level, while inputs at high security levels are modified to an *undefined* value. The enforcement is *precise* in that if a program is termination-sensitively noninterferent, then eventually we have all the outputs as produced by the normal (single) execution. However, performance issues become pressing when the computational task is heavy and the security lattice is huge.

Dynamic monitoring enforces termination-insensitive noninterference. It may incur slight performance issues, and is not suitable for safety critical systems in that the execution needs to be stopped when information leaks are dynamically detected. In some cases, halted executions themselves can result in termination leaks.

Model checking also allows for precise and permissive enforcement of information flow policies, but the approach suffers from state-space explosion when dealing with software, especially systems with fine-grained concurrency, and/or big data-domains that call for delicate abstraction techniques to be employed. Bisimulation-checking has similar advantages and disadvantages to those of model checking.

Self-composition reduces information flow properties to standard properties (as opposed to hyperproperties) that can be expressed in a classical program logic or temporal logic. Its value is mostly in making it possible for security verification to borrow techniques and insights from safety verification.

Finally, information flow type systems are a good choice of technique for the static verification of large-scale software applications. The analysis is performed statically and insecure code will not be executed, which makes the approach suitable for safety critical systems. The analysis does not explore the state-space, which makes it scale to larger systems compared with model checking. Dealing with complex security lattices poses no great difficulty, as long as it is simple

[2]Certain constraints, like the scheduler is not able to observe confidential memory, are often unavoidable though.

to decide the partial order between two elements of the lattice. However, information flow type systems are often overly coarse and they fire too many "false alarms". When a false alarm is fired, it may be difficult to locate the cause of the type-checking failure (e.g., the Jif system [Mye99]). Downgrading operations are frequently needed, partially compensating for the imprecision [NNL15b]. However, the use of downgrading makes it more difficult to articulate the security guarantee, despite the extensive study of downgrading policies [SS09]. Nevertheless, we believe that the aforementioned merits of information flow type systems make them irreplaceable by other techniques, and we work on more fine-grained typing to counter the drawbacks in permissiveness.

Chapter 3

Unwinding Security as Self-Bisimilarity

In process-algebraic security, an important class of security properties is the BNDC-like (BNDC for "Bisimulation-based Non-Deducibility on Compositions") properties studied by Focardi et al (e.g., [FG01, FR06]). Many of the BNDC-like properties follow a pattern called "unwinding". That is, they require that for all derivatives P' of the initial process P, two different processes reachable from P' via some particular actions should be related in some way (typically they should be equivalent with respect to their public/clean behaviors).

In language-based security, the formulation of security properties frequently follows a different pattern. That is, a particular kind of bisimulation is defined such that bisimilarity is not reflexive, and a process P is deemed secure if P is bisimilar to itself (e.g., [SS00, BC02, Dam06]).

The connection between language-based security and process-algebraic security has been established [FRS05, Cas07]. In more detail, under an encoding of While programs in the process calculus VSPA, a notion of *timing-sensitive security* for such programs is shown to coincide with PBNDC (Persistent BNDC [FR06]). At about the same time, Crafa and Rossi [CR05] showed that the PBNDC property coincides with self-bisimilarity in the π-calculus.

In this chapter, we present a similar result to that of Crafa and Rossi in our setting. That is, by suitably relating the formulation of a process-algebraic bisimulation to the unwinding pattern, a process is unwinding secure, if and only if it is self-bisimilar. Our purpose is to pave the way for our further development in Chapter 4, where a security property capturing the meaning of security levels in two different dimensions — presence and content — will be presented. We intend to build our property on the unwinding framework, but find its direct

form more difficult to use (for our very purpose) than that of self-bisimilarity.

A benefit of our presentation might be that the close relationship between the two patterns is exposed with leanest machinery. In [FRS05] and [Cas07], program variables are encoded as processes that support reading/writing operations, and related complication is caused. In [CR05], the expression of the noninterference properties meshes with typing.

In this chapter, the reader is introduced to the Quality Calculus [NNV12], a recent variant of the π-calculus made to help build robust systems in the presence of unreliable communication links. The Quality Calculus will be used to develop the theory of this chapter as well as that of Chapter 4.

We will work with *integrity*, and assume the existence of an integrity environment $\mathcal{I} : \mathbf{Ch} \cup \{\bot\} \to \{H, L, \bot\}$ that maps channels to their integrity levels. Naturally, H means high integrity and L means low integrity. We stipulate that $\mathcal{I}(\bot) = \bot$, and $\forall c \in \mathbf{Ch} : \mathcal{I}(c) \neq \bot$, i.e., the function $\mathcal{I}(-)$ is strict and bottom-reflecting.

3.1 The Quality Calculus

3.1.1 Syntax

The Quality Calculus [NNV12] has its roots in the π-calculus and CCS, but allows one to specify criteria on which communications have to succeed for the computation to continue. This can be expressed by the construct $\&_q(b_1, ..., b_n)$ with predicate q and communication binders b_1,...,b_n. Using the construct $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$, one can then specify different continuations of the computation, depending on whether each of these communications has succeeded. The predicate q can refer to any specific binder among b_1, ..., b_n, by its index (1,...,n), and can denote any boolean combination of their evaluation status. Since some previous inputs might be unsuccessful, the optional data types from languages like Standard ML are used: expressions are evaluated to $\mathsf{some}(c)$ with some constant c if they are not missing data, or to none otherwise. For a quick example, consider the process $\&_{1\vee2}(c_1?x_1, c_2?x_2).\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : P_1\ \mathsf{else}\ P_2$. The computation can proceed to the case construct as long as either the input over c_1, or the input over c_2, is successful. In case the former succeeds, receiving datum d_1, x_1 will be bound to $\mathsf{some}(d_1)$, and subsequently P_1 will be executed, making use of y_1, which will be bound to d_1. If the input over c_1 does not succeed until the case construct is reached, then x_1 will be bound to none, which will lead to the execution of P_2 instead of P_1.

The complete syntax is given in Table 3.1. Terms and expressions are separate syntactical categories that capture the distinction between data and optional data. Correspondingly a term variable y can be bound to data and an

Table 3.1: The Syntax of the Quality Calculus

(Terms)	t	::=	$c \mid y \mid f(t_1, \ldots, t_n)$
(Expressions)	e	::=	$x \mid \mathsf{some}(t) \mid \mathsf{none}$
(Binders)	b	::=	$t?x \mid t!t'\{x\} \mid \&_q(b_1, ..., b_n)$
(Processes)	P	::=	$(\nu c)P \mid P_1 \mid P_2 \mid b.P \mid \mathsf{A}(\vec{e}) \mid 0 \mid$
			$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2 \mid$
			$\mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$

expression variable x can be bound to $\mathsf{some}(-)$ or none, containing information about whether there is data at all.

We use c to refer to a constant. The set of constants that can serve as the communication media are channels (**Ch**) and those that can be communicated are data (**Dt**). We allow channels to be communicated but refrain from requiring that all channels can. On the other hand, it is certainly not true that all data can serve as communication media. Hence $\mathbf{Ch} \cap \mathbf{Dt} \neq \emptyset$ but in general there is no inclusion relationship between the two sets. The members of **Dt** are sometimes denoted by d (potentially with subscripts and primes).

Atomic output binders are of the form $t!t'\{x\}$, where the variable x is used as an indicator of whether the output has succeeded, the output content is also bound to x in case it has. We abbreviate $t!t'\{x\}$ to $t!t'$ when such indication provided by x is not needed. With a procedure call $A(\vec{e})$, where $\vec{e}$ is the vector/list of actual arguments, the procedure A needs to be defined as a process, with $A(\vec{x}) \triangleq P$, where $\vec{x}$ is the vector/list of formal parameters. The other features not mentioned so far are mostly standard. For operator precedence, we assume that "." (prefixing with binder) binds more tightly than "|" (parallel composition). Although the Quality Calculus does not have a non-deterministic choice operator, an encoding of internal nondeterminstic choice (in the style of Hoare's CSP) can be done using composite binders $\&_q(b_1, ..., b_n)$ and case constructs, as presented in [NNV12].

Vector Notation For a vector $\vec{v}$, we write $|\vec{v}|$ for its total number of components, and v_j for the j-th one.

3.1.2 Semantics

We make use of a structural congruence that identifies processes that are structurally different but behaviorally equivalent. This structural congruence is the smallest congruence relation satisfying the rules in Table 3.3, where $P \equiv_\alpha P'$ represents that P and P' are α-equivalent. Note that putting two constructs

that are related in a congruence relation in the same context yields related constructs.

In Table 3.3, $fc(P)$ gives the set of free constants of the process P, and is characterized by the defining equations in Figure 3.2 as the least fixpoint. We

$$
\begin{aligned}
fc((\nu c)P) &= fc(P) \setminus \{c\} \\
fc(P_1|P_2) &= fc(P_1) \cup fc(P_2) \\
fc(b.P) &= fc(b) \cup fc(P) \\
fc(\mathsf{A}(\vec{e})) &= fc(P) \cup fc(\vec{e}) \quad \text{where } A(\vec{e}) \triangleq P \\
fc(0) &= \emptyset \\
fc(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2) &= fc(e) \cup fc(P_1) \cup fc(P_2) \\
fc(\mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2) &= fc(t) \cup fc(P_1) \cup fc(P_2)
\end{aligned}
$$

Figure 3.2: The Defining Equations for $fc(P)$

can complete the part of the equation system for binders, expressions and terms in the natural way.

Table 3.3: The Structural Congruence

$P_1\|P_2 \equiv P_2\|P_1$	$(\nu c_1)(\nu c_2)P \equiv$
$P_1\|(P_2\|P_3) \equiv (P_1\|P_2)\|P_3$	$\quad (\nu c_2)(\nu c_1)P \quad (\text{if } c_1 \neq c_2)$
$P \mid 0 \equiv P$	$(\nu c)(P_1\|P_2) \equiv ((\nu c)P_1)\|P_2$
$(\nu c)P \equiv P \quad (\text{if } c \notin fc(P))$	$\quad (\text{if } c \notin fc(P_2))$
$P \equiv_\alpha P' \Rightarrow P \equiv P'$	

The transition relation for processes is given in Table 3.4 and the transition relation for binders is given in Table 3.5. We follow [NNV12] to use without specifying in detail an evaluation relation $\triangleright$ for terms and expressions.

The transitions for processes take the form $P \xrightarrow{\alpha} P'$, representing that the process P performs the action[1] α and becomes P'. In Table 3.4, the rules for communication come first. The rule for output simply says that if the binder b can perform an output $c!c'$, then we check whether all sub-binders necessary for b to be passed have already performed their own communications, using the binder evaluation relation $b ::_v \theta$ defined in Table 3.5. If b can be passed, then we embark on the continuation P with the substitution θ; otherwise we stay with the derivative b' of b that records all the communications that have happened so far. The input rule is analogous. The only operation that produces a τ is internal synchronization, as expressed by the third rule.

[1]Note that the α here has nothing to do with $\equiv_\alpha$.

Table 3.4: The Transition Relation for Processes

$$\frac{b \xrightarrow{c!c'} b'}{b.P \xrightarrow{c!c'} P'} \text{ where } P' = \begin{cases} P\theta & b' ::_{\mathrm{tt}} \theta \\ b'.P & b' ::_{\mathrm{ff}} \theta \end{cases} \qquad \frac{b \xrightarrow{c?c'} b'}{b.P \xrightarrow{c?c'} P'} \text{ where } P' = \begin{cases} P\theta & b' ::_{\mathrm{tt}} \theta \\ b'.P & b' ::_{\mathrm{ff}} \theta \end{cases}$$

$$\frac{P_1 \xrightarrow{c!c'} P_1' \quad P_2 \xrightarrow{c?c'} P_2'}{P_1 | P_2 \xrightarrow{\tau} P_1' | P_2'}$$

$$\frac{e \triangleright \mathsf{some}(c) \quad P_1[c/y] \xrightarrow{\alpha} P'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2 \xrightarrow{\alpha} P'} \qquad \frac{e \triangleright \mathsf{none} \quad P_2 \xrightarrow{\alpha} P'}{\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2 \xrightarrow{\alpha} P'}$$

$$\frac{t \triangleright \mathbf{tt} \quad P_1 \xrightarrow{\alpha} P'}{\mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\alpha} P'} \qquad \frac{t \triangleright \mathbf{ff} \quad P_2 \xrightarrow{\alpha} P'}{\mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\alpha} P'}$$

$$\frac{\vec{e} \triangleright \vec{w} \quad P[\vec{w}/\vec{x}] \xrightarrow{\alpha} P'}{\mathsf{A}(\vec{e}) \xrightarrow{\alpha} P'} \text{ if } A(\vec{x}) \triangleq P \qquad \frac{P \xrightarrow{\alpha} P'}{(\nu c)P \xrightarrow{\alpha} (\nu c)P'} \text{ if } c \notin Ch(\alpha)$$

$$\frac{P_1 \xrightarrow{\alpha} P_2}{P_1 | P \xrightarrow{\alpha} P_2 | P} \qquad \frac{P_1 \equiv P_2 \quad P_2 \xrightarrow{\alpha} P_3 \quad P_3 \equiv P_4}{P_1 \xrightarrow{\alpha} P_4}$$

Table 3.5: The Transition Relation for Binders

$$\frac{t \triangleright c}{t?x \xrightarrow{c?c'} [c : \mathsf{some}(c')/x]} \qquad \frac{t \triangleright c \quad t' \triangleright c'}{t!t'\{x\} \xrightarrow{c!c'} [c : \mathsf{some}(c')/x]}$$

$$\frac{b_j \xrightarrow{\alpha} b_j'}{\&_q(\ldots, b_j, \ldots) \xrightarrow{\alpha} \&_q(\ldots, b_j', \ldots)}$$

$$\frac{t!t'\{x\} ::_{\mathrm{ff}} [\mathsf{none}/x] \quad t?x ::_{\mathrm{ff}} [\mathsf{none}/x] \quad [c : \mathsf{some}(c')/x] ::_{\mathrm{tt}} [\mathsf{some}(c')/x] \quad b_1 ::_{v_1} \theta_1 \ \ldots \ b_n ::_{v_n} \theta_n}{\&_q(b_1, ..., b_n) ::_v \theta_n \ldots \theta_1} \; v = [\![q]\!](v_1, \ldots, v_n)$$

The rules for the case construct base the decision as to which branch to take on whether the evaluation of e results in $\mathsf{some}(-)$ or none. If the first branch is taken, the substitution $[c/y]$ is applied to it, to signal that y has been bound to the constant c. The rule for procedure calls execute the procedure body for one step, after substituting the evaluation result of the actual arguments for the formal ones. Note that by $\vec{e} \triangleright \vec{w}$ we mean $|\vec{e}| = |\vec{w}|$ and $\forall i : e_i \triangleright w_i$. The rule for $(\nu c)P$ says that $(\nu c)P$ can perform an action that P can perform, provided that the channel c does not appear in α.

A property of this semantics is that all actions are communication actions, among which (only) internal communications are represented by τ's.

We now turn to the transition and evaluation of binders specified in Table 3.5. The transitions for binders take the form $b \xrightarrow{\alpha} b'$, representing that the binder b performs the action α and becomes b'. Since a composite binder is not always passed after performing a communication action, the intermediate form $[c : \mathsf{some}(c')/x]$ (essentially an extension to the binder syntax) is used to record the data involved in the communications that have already happened. In the case of input, the received content c' is picked non-deterministically and any constant is possible. The notation $b' ::_v \theta$ defined in the same table represents the evaluation of binder b' into substitution θ and boolean v, where θ records the data bound to variables and v indicates whether the binder can already be passed. The notation $[\![q]\!](v_1, \ldots, v_n)$ specifies how to combine the evaluation statuses of the sub-binders of a composite binder with predicate q. Two simple examples are given below.

$$[\![\forall]\!](v_1, \ldots, v_n) = [\![1 \wedge ... \wedge n]\!](v_1, \ldots, v_n) = v_1 \wedge ... \wedge v_n$$
$$[\![\exists]\!](v_1, \ldots, v_n) = [\![1 \vee ... \vee n]\!](v_1, \ldots, v_n) = v_1 \vee ... \vee v_n$$

The semantics of the Quality Calculus is illustrated by the following example.

Example 3.1 *Consider the procedure call M, where*

$$\begin{aligned} M \triangleq \&_{1\vee 2}(c_1?x_1, c_2?x_2).\ & \\ & \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : c_1'!y_1.M \\ & \mathsf{else\ case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2) : c_2''!y_2.M \\ & \mathsf{else}\ 0 \end{aligned}$$

We have the following transition sequence.

$$\begin{aligned} M \xrightarrow{c_2?d}\ & \mathsf{case\ none\ of\ some}(y_1) : c_1'!y_1.M \\ & \mathsf{else\ case\ some}(d)\ \mathsf{of\ some}(y_2) : c_2''!y_2.M\ \mathsf{else}\ 0 \\ \xrightarrow{c_2''!d}\ & M \end{aligned}$$

We elaborate on the derivation of the first step above. We have the binder transition $\&_{1\vee 2}(c_1?x_1, c_2?x_2) \xrightarrow{c_2?d} \&_{1\vee 2}(c_1?x_1, [c_2 : \mathsf{some}(d)/x_2])$, and the evaluation $\&_{1\vee 2}(c_1?x_1, [c_2 : \mathsf{some}(d)/x_2]) ::_{\mathbf{tt}} [\mathsf{some}(d)/x_2]$. Thus the binder $\&_{1\vee 2}(c_1?x_1, c_2?x_2)$ can be passed, with the substitution $[\mathsf{some}(d)/x_2]$ applied to the process that remains. □

We introduce several pieces of notation related to the semantics to pave way for further development. The channel used by an action α is represented by $ch(\alpha)$, as formulated in the definition below.

Definition 3.1 ($ch(\alpha)$) $ch(\alpha) = \begin{cases} c & \text{if } \exists \rho, c' : \alpha = c\rho c' \\ \bot & \text{otherwise} \end{cases}$

The weak transition $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ represents $(\stackrel{\tau}{\longrightarrow})^* \circ \stackrel{\alpha}{\longrightarrow} \circ (\stackrel{\tau}{\longrightarrow})^*$ when $\alpha \neq \tau$, and $(\stackrel{\tau}{\longrightarrow})^*$ (rather than $(\stackrel{\tau}{\longrightarrow})^* \circ \stackrel{\tau}{\longrightarrow} \circ (\stackrel{\tau}{\longrightarrow})^*$) when $\alpha = \tau$. We write $P \rightarrow^* P'$ to represent that there is a transition sequence starting from P and reaching P' in zero or more steps. The zero-step transition $P \rightarrow^0 P'$ is understood in the more general sense of $P \equiv P'$, rather than $P = P'$.

3.2 Security through Unwinding

An important approach to the non-interference-style characterization of process-algebraic security is unwinding [BFPR03b, BPR04].

In more detail, a process is said to be secure if it belongs to an unwinding class parameterized by a relation $\sim$ on processes and a transition sequence $\dashrightarrow$:

$$\mathcal{W}(\sim, \dashrightarrow) = \{P \mid \forall P', P'', \alpha \; s.t. \; P \rightarrow^* P' \wedge P' \stackrel{\alpha}{\longrightarrow} P'' \wedge \mathcal{I}(ch(\alpha)) = L : \exists P''' : P' \dashrightarrow P''' \wedge P'' \sim P'''\}.$$

For integrity, $\mathcal{W}(\sim, \dashrightarrow)$ contains processes P that satisfy the following condition. For each derivative P' of P, and P'' reachable from P' by performing a low integrity action, there exists P''' reachable from P' over $\dashrightarrow$, such that P'' is related to P''' in $\sim$.

There are two classes of choices for $\sim$, one being a (non-standard) bisimulation relation, the other being a trace-equivalence. We focus on the former class, where all the notable security properties (SBNDC, PBNDC, CP_BNDC) are expressed by instantiating $\sim$ with "weak bisimilarity on non-corrupt actions" (denoted by $\approx^H$) [BFPR03b, BPR04]. The relation $\approx^H$ is the union of all "weak bisimulations on non-corrupt actions", defined below.

Definition 3.2 (Weak Bisimulation on Non-Corrupt Actions) A symmetric relation R is a weak bisimulation on non-corrupt actions if for all processes P_1, P_2, $P_1 \; R \; P_2$ implies

$$\forall \alpha, P_1' \; s.t. \; P_1 \stackrel{\alpha}{\longrightarrow} P_1' \wedge \mathcal{I}(ch(\alpha)) \neq L : \exists P_2' : P_2 \stackrel{\widehat{\alpha}}{\Longrightarrow} P_2' \wedge P_1' \; R \; P_2'.$$

We restrict ourselves to $\mathcal{W}(\approx^H, \dashrightarrow)$ in the subsequent discussion. About $\dashrightarrow$, we make the following assumptions, which are met by all the major security properties SBNDC, PBNDC and CP_BNDC.

Assumption 3.1 *For all processes P and P', and communication actions α,*

1. $P \stackrel{\tau}{\longrightarrow} P' \Rightarrow P \dashrightarrow P'$,
2. $P \stackrel{\alpha}{\longrightarrow} P' \wedge \mathcal{I}(ch(\alpha)) = H \;\; \Rightarrow \;\; P \dashrightarrow P'$, *and*
3. $\dashrightarrow$ *is transitive.*

3.3 Security through Self-Bisimilarity

The unwinding scheme requires that a low integrity action should be "simulated" by a sequence $\dashrightarrow$ of actions, directly in the formulation of $\mathcal{W}(-,-)$. The simulation requirement of a high integrity action, on the other hand, is embedded inside $\sim$.

In this section, we formulate a kind of bisimulation that combines the simulation requirements for actions of low and high integrity. We then characterize the security of processes P as the bisimilarity of P to itself. The term "self-bisimilarity" may sound unfortunate since standard bisimilarity is an equivalence relation, and "self-bisimilarity" suggests triviality. Note that this is not the case with the (partial) bisimulations often used in language-based security [SS00, BC02, FRS05, Dam06].

We introduce the parameterized transition sequence $\stackrel{\alpha}{\Longrightarrow}_{\dashrightarrow}$ defined as:

$$\stackrel{\alpha}{\Longrightarrow}_{\dashrightarrow} = \begin{cases} \dashrightarrow & \text{if } \mathcal{I}(ch(\alpha)) = L \\ \stackrel{\widehat{\alpha}}{\Longrightarrow} & \text{otherwise} \end{cases}$$

We then define a notion of $\dashrightarrow$-bisimulation that is parameterized over the transition sequence $\dashrightarrow$.

Definition 3.3 ($\dashrightarrow$-Bisimulation) A symmetric relation R qualifies as a $\dashrightarrow$-bisimulation if $P_1\ R\ P_2$ implies

$$\forall \alpha, P_1'\ s.t.\ P_1 \stackrel{\alpha}{\longrightarrow} P_1' : \exists P_2' : P_2 \stackrel{\alpha}{\Longrightarrow}_{\dashrightarrow} P_2' \wedge P_1'\ R\ P_2'$$

We define $\dashrightarrow$-bisimilarity $\sim_{\dashrightarrow}$ as the union of all $\dashrightarrow$-bisimulations, and $\dashrightarrow$-security of a process P as the self-relatedness of P in $\sim_{\dashrightarrow}$.

Definition 3.4 ($\dashrightarrow$-Security) A process P is $\dashrightarrow$-secure if and only if $P \sim_{\dashrightarrow} P$.

3.4 Unwinding Security as Self-Bisimilarity

In this section we establish the connection between the two patterns in defining security with a concise proof.

Lemma 3.5 *The relation $\approx^H$ is an equivalence relation.*

Proof. The transitivity of $\approx^H$ can be shown as follows. Suppose $P_1 \approx^H P_2$ and $P_2 \approx^H P_3$. Then there exist weak bisimulations on non-corrupt actions, R' and R'', such that $(P_1, P_2) \in R'$ and $(P_2, P_3) \in R''$. It is not difficult to show that

the relation $\{(P_a, P_c) \mid (P_a, P_b) \in R' \wedge (P_b, P_c) \in R''\}$ is a weak bisimulation on non-corrupt actions that relates P_1 and P_3. The proofs for the reflexivity and symmetry of $\approx^H$ are also straightforward. □

Lemma 3.6 *For all processes P_1 and P_2, if $P_1 \approx^H P_2$, and $P_1 \dashrightarrow P_1'$, then there exists process P_2' such that $P_2 \dashrightarrow P_2'$, and $P_1' \approx^H P_2'$.*

The proof of this lemma is straightforward by induction on the length of $P_1 \dashrightarrow P_1'$, and use of Assumption 3.1 and Definition 3.2.

Lemma 3.7 (Unwinding Security Implies Self-Bisimilarity) *For all processes P, if $P \in \mathcal{W}(\approx^H, \dashrightarrow)$, then $P \sim_{\dashrightarrow} P$.*

Proof. Construct the relation

$$R_\star = \{(P_1, P_2) \mid P \to^* P_1 \wedge P \to^* P_2 \wedge P_1 \approx^H P_2\}$$

Lemma 3.5 gives the symmetry of $R_\star$, and the validity of $P\ R_\star\ P$. We now show that $R_\star$ is a $\dashrightarrow$-bisimulation.

Suppose there is transition $P_1 \xrightarrow{\alpha} P_1'$. We make a case distinction based on $\mathcal{I}(ch(\alpha))$.

1. $\mathcal{I}(ch(\alpha)) = L$. Since $P \to^* P_1$, and $P \in \mathcal{W}(\approx^H, \dashrightarrow)$, there exists some P_1'' such that $P_1 \dashrightarrow P_1''$ and $P_1' \approx^H P_1''$. By Lemma 3.6, there exists some process P_2' such that $P_2 \dashrightarrow P_2'$ and $P_1'' \approx^H P_2'$. By Lemma 3.5, we have $P_1' \approx^H P_2'$. Hence we have $P_2 \overset{\alpha}{\Longrightarrow}_{\dashrightarrow} P_2'$ and $(P_1', P_2') \in R_\star$.

2. $\mathcal{I}(ch(\alpha)) \neq L$. By the definition of $\approx^H$, there exists P_2' such that $P_2 \overset{\widehat{\alpha}}{\Longrightarrow} P_2'$ and $P_1' \approx^H P_2'$. Hence we have $P_2 \overset{\alpha}{\Longrightarrow}_{\dashrightarrow} P_2'$ and $(P_1', P_2') \in R_\star$.

Therefore $R_\star$ qualifies as a $\dashrightarrow$-bisimulation, which completes the proof. □

Lemma 3.8 $\sim_{\dashrightarrow} \subseteq \approx^H$

Proof. The proof is trivial by noting that each $\dashrightarrow$-bisimulation R qualifies as a weak bisimulation on non-corrupt actions. □

Lemma 3.9 (Self-Bisimilarity implies Unwinding Security) *For all processes P, if $P \sim_{\dashrightarrow} P$, then $P \in \mathcal{W}(\approx^H, \dashrightarrow)$.*

PROOF. Assume that $P \sim_{\dashrightarrow} P$. It is not difficult to show that for all P_1 such that $P \rightarrow^* P_1$, there exists P_2 such that $P \rightarrow^* P_2$ and $P_1 \sim_{\dashrightarrow} P_2$. Note that there is no constraint on either transition sequence.

For all α, P_1' such that $P_1 \xrightarrow{\alpha} P_1'$ and $\mathcal{I}(ch(\alpha)) = L$, there exists P_2' such that $P_2 \stackrel{\alpha}{\Longrightarrow}_{\dashrightarrow} P_2'$ (which boils down to $P_2 \dashrightarrow P_2'$) and $P_1' \sim_{\dashrightarrow} P_2'$. By Lemma 3.8, we have $P_1 \approx^H P_2$ and $P_1' \approx^H P_2'$. By the symmetry of $\approx^H$ (Lemma 3.5), we have $P_2 \approx^H P_1$. By Lemma 3.6, there exists P_1'' such that $P_1 \dashrightarrow P_1''$ and $P_2' \approx^H P_1''$. By the transitivity of $\approx^H$ we have $P_1' \approx^H P_1''$. □

THEOREM 3.10 (UNWINDING SECURITY AS SELF-BISIMILARITY) *For all processes P, $P \in \mathcal{W}(\approx^H, \dashrightarrow)$ if and only if $P \sim_{\dashrightarrow} P$.*

Theorem 3.10 essentially states that the unwinding class $P \in \mathcal{W}(\approx^H, \dashrightarrow)$ characterizes exactly the reflexive fragment of the partial equivalence relation $\sim_{\dashrightarrow}$.

REMARK 3.1 *A different perspective on the secure semantics induced by the unwinding-condition is provided — essentially that of the bisimulation-game for $\sim_{\dashrightarrow}$. In more detail, a game is played* continually *between an attacker and a defender: at any point of execution, the attacker tries to exert malicious influence by choosing to perform/suspend a low integrity action; whereas the defender attempts to nullify the effects by being able to perform the same high integrity action.*

SBNDC as Self-Bisimilarity

We specialize the previous result and obtain a notion of process-algebraic security that coincides with SBNDC (Strong Bisimulation-Based non Deducibility on Compositions [FG01]), but possesses the form of "self-bisimilarity". This provides a basis for the development in Chapter 4.

The following definitions are taken from [LNN15], but actually define $\stackrel{\alpha}{\Longrightarrow}_{\equiv}$ and $\equiv$-bisimulation (recall that $\equiv$ is the structural congruence) in the framework of this chapter.

DEFINITION 3.11 $(\stackrel{\alpha}{\Longrightarrow}_{\triangleright})$ $\stackrel{\alpha}{\Longrightarrow}_{\triangleright} = \begin{cases} \equiv & \text{if } \mathcal{I}(ch(\alpha)) = L \\ \stackrel{\widehat{\alpha}}{\Longrightarrow} & \text{otherwise} \end{cases}$

DEFINITION 3.12 ($\triangleright$-BISIMULATION) A binary relation R on processes qualifies as a $\triangleright$-bisimulation if it is a symmetric relation, and $P_1 \; R \; P_2$ implies

$$\forall P_1', \alpha \; s.t. \; P_1 \xrightarrow{\alpha} P_1' : \exists P_2' : P_2 \stackrel{\alpha}{\Longrightarrow}_{\triangleright} P_2' \wedge P_1' \; R \; P_2'.$$

In $\triangleright$-bisimulation, $\triangleright$ is not a parameter, but merely part of the name that reflects the fact that a low integrity communication is simulated by *inaction*, resulting in a *triangular* structure[2]. Note the strong resemblance between the requirement imposed for low and high integrity actions in $\triangleright$-bisimulation and the unwinding conditions lr_H and $osc_{L,\emptyset,H}$ in [Man01].

We will denote by $\sim_{\triangleright}$ the union of all $\triangleright$-bisimulations, i.e., $\triangleright$-bisimilarity. It is obvious that $\sim_{\triangleright}$ is identical to $\sim_{\equiv}$ and that it is a $\triangleright$-bisimulation itself.

The relation $\sim_{\triangleright}$ has the following simple property, which will be used in our further development in the next chapter.

Lemma 3.13 *For all processes P_1, P_1', and P_2, if $P_1 \sim_{\triangleright} P_2$, and $P_1 \equiv P_1'$, then $P_1' \sim_{\triangleright} P_2$.*

Proof. Construct the relation

$$R_\star = \{(P_\mathrm{a}', P_\mathrm{b}') \mid P_\mathrm{a} \sim_{\triangleright} P_\mathrm{b} \wedge P_\mathrm{a} \equiv P_\mathrm{a}' \wedge P_\mathrm{b} \equiv P_\mathrm{b}'\}.$$

Using the definition of $\sim_{\triangleright}$, it is trivial to show that $R_\star$ is a $\triangleright$-bisimulation such that $P_1' \; R_\star \; P_2$. □

We next give the definition of SBNDC as a specialization of the unwinding class, and state the equivalence of SBNDC and self-relatedness in $\sim_{\triangleright}$ as a corollary of Theorem 3.10.

Definition 3.14 (SBNDC) A process $P \in \mathit{SBNDC}$ if $P \in \mathcal{W}(\approx^H, \equiv)$.

Corollary 3.15 *For all processes P, $P \in \mathit{SBNDC}$ if and only if $P \sim_{\triangleright} P$.*

Remark 3.2 *In [LNN15], we suspected that the side result on identifying SBNDC and self-bisimilarity may be the first such coincidence result derived for a process algebra. This conjecture is falsified due to similar results in [CR05].*

[2]The notion of $\triangleright$-bisimulation is close, although not identical, to the notion of "Weak Bisimulation up to H with zero τ" that is used in an alternative formulation [BFPR03a] of SBNDC. Here the "H" means high confidentiality.

Chapter 4

Presence and Content, Independently

Communication is important in many process calculi (e.g., [SW01, Hoa85]). Communication channels can often be compared to variables: writing to a channel resembles assigning to a variable, and reading from a channel resembles evaluating a variable. Unlike the usual treatment of assignments in programming languages, communications are oftentimes *blocking* operations in process calculi. There is a need for *synchronization*, and failing to synchronize can block further computation. Thus it becomes more relevant to consider *whether* a communication can happen at all as a separate aspect in addition to *what* the communication content is. And the analogy of communication to assignments is retained in the dimension of *content*.

For a concrete example, consider the communicating process $c?x.c'!d$, versus the simple program $x := z; z' := d$. The content of the variable z cannot be leaked via the content of the variable z'. Similarly, the content received from the channel c cannot be leaked via the output content over channel c'. However, when $c?x$ could potentially be blocked, the *presence* of the input over c can be leaked via the mere *presence* of the output over c'. Hence, if c' is public, then c cannot be confidential. This is the requirement of the SBNDC condition discussed in Section 3.4, and the viewpoint taken in [FG01], [Kob05], [Cas07], [CCDR10], [vBV11], and [RHS12]. Allowing the distinction between the presence and content of communications, we immediately obtain a finer-grained analysis. Supposing both the "presence" level and the "content" level of c' are public, then the content level of c can still be confidential — only the presence level needs to be public, for the process to be secure.

The fine-grained distinction between "presence" and "content" is, however, not new: this approach has been pursued in previous developments on secur-

ing interactive programs (e.g., [SM02, RHS12, RS14]). Nevertheless, being concerned with confidentiality, existing work imposes the constraint that *"presence" can be no more confidential than "content"*: observing the content of a communication implies the knowledge that the communication is happening (present).

By the usually perceived duality (e.g., [MPP13]) between confidentiality and integrity, separating "presence" and "content" applies for integrity as well. Still consider the process $c_1?x.c_2!d$. If both the "presence" and the "content" of communication over c_2 are of high integrity, then only the "presence" of communication over c_1 needs to be of high integrity as well, the input content can still be of low integrity, for the process to be secure. However, the aforementioned constraint would preclude the use of channels with low presence integrity and high content integrity. Nevertheless, this combination is practically meaningful. When message authentication codes (MAC) are used, a MAC-checker can detect tampered (low integrity) content and choose to accept only high integrity content. As a result, the content, once received by an end user, can be used by her with confidence that no harm will arise. The worry, though, is that the communication allowing her to receive that content may not be present. The suspension of this communication may be caused, for example, by message rejection in the MAC-checker due to content corruption. It is therefore sensible to regard the user channel as having low presence integrity and high content integrity.

In this chapter, we formulate a novel bisimulation-based noninterference property for integrity, where the *presence* and *content* of communication events are dealt with separately, and *all* combinations of integrity levels for these two dimensions are allowed. The property is shown to degenerate to SBNDC (e.g., [FG01]) when the distinction between presence and content is intensionally blurred, and a compositionality result is obtained to facilitate a structural approach to information flow analysis in a concurrent setting.

In Section 4.1, we further motivate the distinction between presence integrity and content integrity with a few concrete example processes in the Quality Calculus. In Section 4.2, we introduce a semantics for the Quality Calculus that is parameterized with the strategy of the environment, which will help formulate a concise and intuitive security condition. In Section 4.3, we introduce δ-security — a noninterference notion that articulates the integrity guarantee of presence labels and content labels. Two theoretical properties of δ-security — its connection with PBNDC and its compositionality — are then discussed in Section 4.4. In Section 4.5, we present further examples that illustrate our compositionality result, and provide further insight on the difference between integrity and confidentiality when "presence" and "content" are considered as separate dimensions. Finally, related work is discussed in Section 4.6.

Syntactical Convention In the examples to be presented in this chapter, we sometimes write $c?x[y].P$ and $\&_q(..., c?x[y], ...).P$ in case the input has to happen for the binder corresponding to $c?x$ to be passed. This represents $c?x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P\ \mathsf{else}\ P'$ and $\&_q(..., c?x[y], ...).\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P\ \mathsf{else}\ P'$, respectively, where in the latter situation the case can also be nested in other branching constructs. This is only to prevent parts of our specifications from becoming unnecessarily bloated.

4.1 Presence Integrity and Content Integrity

Let us give a few examples (in Fig.4.1) to frame our mind in terms of *presence* integrity and *content* integrity, and further motivate the noninterference property to be proposed. Channels with two subscripts (L or H, representing their integrity classification) will often be used. The first level describes the presence dimension and the second describes the content dimension. For each subscript, an L (resp. H) will denote low (resp. high) integrity. Recall from Section 3.1 that "." binds more tightly than "|".

1. $c_{HL}?x[y].c'_{HL}!y$
2. $c_{LH}?x[y].c'_{LH}!y$
3. $c_{LH}?x_1[y_1].c_{LL}?x_2[y_2].c'_{LH}!f(y_1, y_2)$,
   where $\forall d_2, d'_2\ s.t.\ d_2 \neq d'_2 : \exists d_1 : f(d_1, d_2) \neq f(d_1, d'_2)$
4. $c_{LH}?x[y].c'_{LH}!y \mid c'_{LH}!d$
5. $c_{HH}?x[y].c'_{HH}!y \mid c'_{HH}!d$
6. M where
   $M \triangleq \&_{1\vee 2}(c_{LH}?x_1, c_{LL}?x_2).$
   $\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : c'_{LH}!y_1.M$
   $\mathsf{else\ case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2) : c'_{LL}!y_2.M$
   $\mathsf{else}\ 0$
7. A where
   $A \triangleq \&_{1\vee 2}(c_{LL}?x_1, c'_{LL}?x_2).$
   $\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : c_{HL}!y_1.A$
   $\mathsf{else\ case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2) : c_{HL}!y_2.A$
   $\mathsf{else}\ 0$

Figure 4.1: Example Processes for Presence and Content Integrity

Processes 1 and 2 are intuitively secure. In process 1, given the low content integrity and high presence integrity of c_{HL}, the corruption of the input content by an attacker can be "passed on" to the output content, while the input cannot be blocked by the attacker, consequently blocking the output. Hence the low content integrity, high presence integrity of c'_{HL} can be justified in accordance

with the integrity classes of c_{HL}. In process 2, any influence on the presence of the input can in turn influence the presence of the output, but cannot by itself corrupt the output content, which also demonstrates the consistency of the integrity classes of c_{LH} and c'_{LH}. On the other hand, process 3 is insecure: the classification of c'_{LH} does not meet the intuition that both the presence of the final output, and its content, can be badly influenced.

One might think that the presence integrity can either be high for all channels, or low for all channels, hence at most one of the classes "high" and "low" is needed. This is not true, as illustrated by the insecurity of process 4, and the security of process 5. In process 4, the content integrity of the output channel cannot be H, since the presence of the input leads to more choices for the output content over c'_{LH}, some of which ($\neq d$) may not be possible with the input still blocked.

Given the occasional existence of insecure dependency of high integrity content on low integrity presence such as that in process 4, it becomes interesting to see when certain source channels have the presence level L, which sink channels can still have the content level H without being affected. Process 6 is a call to the procedure M whose definition starts from the subsequent line. This process is in fact a simple-minded "multiplexer" that directs incoming data from c_{LH} to c'_{LH}, and from c_{LL} to c'_{LL}. Note that if one of the four channels has low presence integrity, then all channels have low presence integrity, since the influence by the presence of communication over one of the channels on the control flow is global. However, c'_{LH} preserves the high content integrity of c_{LH}, despite this pervasive corruption on the "presence" dimension.

The process 7 is a call to procedure A whose body uses the same predicate $1 \vee 2$, which enables it to source from alternative channels c_{LL} and c'_{LL}. The input content, no matter from which source channel, will be output over the channel c_{HL}. The process is not secure if the environment can block the two inputs at the same time. However, c_{LL} and c'_{LL} might represent sources that are geographically distant or that fail with drastically different causes, which can be modeled by an *environment strategy* (e.g., [RHS12, MC12]) that always provides at least one of the inputs when the computation proceeds to the composite input binder. The procedure call will be secure under that strategy.

We end this section with a conceptualization of *presence integrity* and *content integrity*, although a more technical characterization comes along with our security property to be presented later.

- Presence integrity: for each i, whether the *existence* of the i-th output/input over channel c in a finite sequence π of communication actions can be influenced by the attacker
- Content integrity: for each i, whether the content of the i-th output/input over channel c can be influenced by the attacker, *in case the input/output exists in a finite sequence π of communication actions*

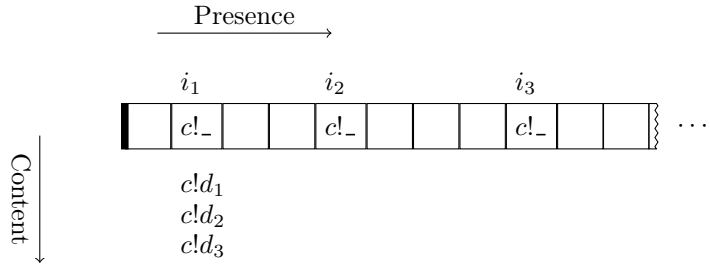


Figure 4.2: The Presence Dimension and the Content Dimension

Note that it is not only whether an input/output on a channel c is eventually available, that matters, but how many times it occurs in each computation sequence, since we are concerned with nonterminating computations and looping behaviors: the processes 6 and 7 in Fig. 4.1 are such examples[1].

It is illustrated in Figure 4.2 that under the impact of the attacker, the i-th use of channel c for output can happen alternatively at global indices i_1, i_2 and i_3, and in each case, the output content can also vary within $\{d_1, d_2, d_3\}$. Note that the perturbation in presence does not necessarily cause variation in content, and vice versa.

Remark 4.1 *The most intuitive viewpoint on corruption in data content may be the introduction of certain fragments provided by an attacker into the content, or the substitution of these fragments for original ones. This pertains mainly to explicit flows and to the authenticity of the content provider, and is a special case of our view of content corruption as influence on its possibilities.*

4.2 An Environment-Parameterized Semantics of the Quality Calculus

The semantics to be presented in this section is parameterized on the strategy of the computation environment (in the spirit of [WJ90, MC12]), to facilitate the specification of open systems. This is unusual for process calculi, since from a purely theoretical point of view, the environment is nothing but a process. However, as has been mentioned in Section 2.3, we distinguish the environment (a strategy) from the system (a process) to ease the abstract specification of the former, and to allow the former to have different traits than the latter.

In this environment-parameterized semantics, processes and binders make transitions together with sequences $\pi \in \Pi$. Each such sequence contains a separator $\triangle$, which delimits the environment's past actions interacting with the

[1] On the other hand, [ZM05] addresses the problem of whether the eventual *availability* of data in a variable can depend on variables of low *integrity/availability*.

process, and optionally a future communication attempt. Each communication action/attempt is represented by an "abstract binder" $\hat{b} \in AB$:

$$\hat{b} ::= c?[\,] \mid c?c' \mid c!c' \mid \Box.$$

The abstract binders $c?[\,]$ and $c?c'$ ($c \in \mathbf{Ch}$ and $c' \in \mathbf{Dt}$) represent a pending input and a completed input, respectively, *of the environment*; on the other hand, $c!c'$ ($c \in \mathbf{Ch}$ and $c' \in \mathbf{Dt}$) represents either a pending output or a completed output, also *of the environment*. In addition, $\Box$ represents the suspension of any communication by one step.

We write $[\pi]_{\triangle}$ for the prefix of π up to and including the $\triangle$ in it, and $\Pi_{\triangle}$ for the set $\{[\pi]_{\triangle} \mid \pi \in \Pi\}$. We introduce next environment strategies

$$\delta : \Pi_{\triangle} \to 2^{AB \setminus \{c?c' \mid c \in \mathbf{Ch} \wedge c' \in \mathbf{Dt}\}}$$

that constitute the set **Strat**. For each $\pi \in \Pi_{\triangle}$, $\delta(\pi)$ gives the set of abstract binders that represent the environment's intended ways of "exercising" the specification for one more step. Note that in the case where the environment attempts to perform an input, an abstract binder given by $\delta(-)$ contains a hole rather than any concrete content — the environment can prescribe over which channel it wants to receive content from the process, but not which content the process has to provide.

The transition relation for processes is given in Table 4.3. As in the previous chapter, we make use of an unspecified evaluation relation $\triangleright$ for terms and expressions. Each rule is of the form $\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle$ or $\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\text{env}} \langle P,\ \pi'\rangle$, representing the transition of process P under environment π into process P', turning the environment into π'. The action performed is either a communication α, or a lookahead "env" by the environment. This transition relation is defined assuming the same structural congruence $\equiv$ of Table 3.3 as the standard semantics presented in Section 3.1 did.

In the rules for communication in Table 4.3, $b ::_v \theta$ is still defined by Table 3.5 of Chapter 3. The communication rules build on transition rules for binder, which will be explained later. The process $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ is abbreviated as $CS(e, y, P_1, P_2)$, and $Ch(\alpha)$ represents the set of channels occurring in α.

The second last transition rule of Table 4.3 says that when a process P cannot perform any communication action when the environment attempts to use the abstract binder α for the next interaction, we allow P to do a $\Box$-step, signaling that there is one step of delay. At the same time, the observational history of the environment is extended by a $\Box$, recording the observation of this delay.

The last transition rule says that the environment can make its next interaction attempt when its observational history ends with a $\triangle$: it can only

Table 4.3: The Environment-Parameterized Transition Relation for Processes

$\boxed{\alpha \neq \Box}$

$$\frac{\langle b,\ \pi\rangle \xrightarrow{c!c'} \langle b',\ \pi'\rangle}{\delta \vdash \langle b.P,\ \pi\rangle \xrightarrow{c!c'} \langle P',\ \pi'\rangle} \text{ where } P' = \begin{cases} P\theta & (\text{if } b' ::_{\mathrm{tt}} \theta) \\ b'.P & (\text{otherwise}) \end{cases}$$

$$\frac{\langle b,\ \pi\rangle \xrightarrow{c?c'} \langle b',\ \pi'\rangle}{\delta \vdash \langle b.P,\ \pi\rangle \xrightarrow{c?c'} \langle P',\ \pi'\rangle} \text{ where } P' = \begin{cases} P\theta & (\text{if } b' ::_{\mathrm{tt}} \theta) \\ b'.P & (\text{otherwise}) \end{cases}$$

$$\frac{\delta \vdash \langle P_1,\ \triangle c?[\,]\rangle \xrightarrow{c!c'} \langle P_1', \pi_1'\rangle \quad \delta \vdash \langle P_2,\ \triangle c!c'\rangle \xrightarrow{c?c'} \langle P_2',\ \pi_2'\rangle}{\delta \vdash \langle P_1|P_2,\ \pi\rangle \xrightarrow{\tau} \langle P_1'|P_2',\ \pi\rangle}$$

$$\frac{e \triangleright \mathsf{some}(c) \quad \delta \vdash \langle P_1[c/y],\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle CS(e, y, P_1, P_2), \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle} \qquad \frac{e \triangleright \mathsf{none} \quad \delta \vdash \langle P_2,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle CS(e, y, P_1, P_2), \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}$$

$$\frac{t \triangleright \mathbf{tt} \quad \delta \vdash \langle P_1,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle \mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle} \qquad \frac{t \triangleright \mathbf{ff} \quad \delta \vdash \langle P_2,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle \mathsf{if}\ t\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}$$

$$\frac{\vec{e} \triangleright \vec{w} \quad \delta \vdash \langle P[\vec{w}/\vec{x}],\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle \mathsf{A}(\vec{e}),\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle} \text{ if } A(\vec{x}) \triangleq P$$

$$\frac{\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle}{\delta \vdash \langle (\nu c)P,\ \pi\rangle \xrightarrow{\alpha} \langle (\nu c)P',\ \pi'\rangle} \text{ if } c \notin Ch(\alpha)$$

$$\frac{\delta \vdash \langle P_1,\ \pi\rangle \xrightarrow{\alpha} \langle P_2,\ \pi'\rangle}{\delta \vdash \langle P_1|P,\ \pi\rangle \xrightarrow{\alpha} \langle P_2|P,\ \pi'\rangle} \qquad \frac{P_1 \equiv P_2 \quad \delta \vdash \langle P_2,\ \pi\rangle \xrightarrow{\alpha} \langle P_3,\ \pi'\rangle \quad P_3 \equiv P_4}{\delta \vdash \langle P_1,\ \pi\rangle \xrightarrow{\alpha} \langle P_4,\ \pi'\rangle}$$

$$\frac{\neg(\exists c, c', \alpha, P', \pi' : (\alpha = c!c' \vee \alpha = c?c') \ \wedge\ \delta \vdash \langle P,\ \pi_\triangle\beta\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle)}{\delta \vdash \langle P,\ \pi_\triangle\beta\rangle \xrightarrow{\Box} \langle P,\ \pi.\Box_\triangle\rangle}$$

$$\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi.\beta\rangle \ \text{ if } \pi = [\pi]_\triangle \wedge \beta \in \delta(\pi)$$

"prescribe" the most imminent interaction, without further predication of the future.

The transition relation for binders is given in Table 4.4. Each transition rule is of the form $\langle b,\ \pi\rangle \xrightarrow{\alpha} \langle b',\ \pi'\rangle$, representing that the binder b performs the communication action α ($\alpha \neq \tau$) under the environment π and becomes b', turning the environment into π'. The movement of $\triangle$ in the resulting environment records the communication action that has just happened. As in the semantics of Section 3.1, the intermediate binder $[c : \mathsf{some}(c')/x]$ (essentially an extension to the binder syntax) is used to record the data involved in the communications that have already happened.

Table 4.4: The Environment-Parameterized Transition Relation for Binders

$$\frac{t \triangleright c}{\langle t?x, \pi_{\triangle} c!c' \rangle \xrightarrow{c?c'} \langle [c : \mathsf{some}(c')/x], \pi.c!c'_{\triangle} \rangle}$$

$$\frac{t \triangleright c \quad t' \triangleright c'}{\langle t!t'\{x\}, \pi_{\triangle} c?[\,] \rangle \xrightarrow{c!c'} \langle [c : \mathsf{some}(c')/x], \pi.c?c'_{\triangle} \rangle}$$

$$\frac{\langle b_j,\ \pi \rangle \xrightarrow{\alpha} \langle b'_j,\ \pi' \rangle}{\langle \&_q(b_1, ..., b_j, ..., b_n),\ \pi \rangle \xrightarrow{\alpha} \langle \&_q(b_1, ..., b'_j, ..., b_n),\ \pi' \rangle}$$

We illustrate the semantics presented above in Example 4.1, where

$$\delta_{\mathrm{ALL}} = \lambda \pi_{\triangle}.\{c!c' \mid c \in \mathbf{Ch} \wedge c' \in \mathbf{Dt}\} \cup \{c?[\,] \mid c \in \mathbf{Ch}\} \cup \{\Box\},$$

i.e., δ_{ALL} is the strategy that allows the environment to produce any legitimate abstract binder with any observation it has made.

Example 4.1 *The procedure call M in Fig. 4.1 has the following transition sequence.*

$$\begin{aligned}
\delta_{\mathrm{ALL}} \vdash \langle M,\ {}_{\triangle} \rangle &\xrightarrow{\mathrm{env}} \langle M,\ {}_{\triangle} c_{LL}!d \rangle \\
&\xrightarrow{c_{LL}?d} \langle \begin{array}{l} \mathsf{case\ none\ of\ some}(y_1) : c'_{LH}!y_1.M \\ \mathsf{else\ case\ some}(d)\ \mathsf{of\ some}(y_2) : c''_{LL}!y_2.M\ \mathsf{else}\ 0 \end{array} ,\ c_{LL}!d_{\triangle} \rangle \\
&\xrightarrow{\mathrm{env}} \langle \begin{array}{l} \mathsf{case\ none\ of\ some}(y_1) : c'_{LH}!y_1.M \\ \mathsf{else\ case\ some}(d)\ \mathsf{of\ some}(y_2) : c''_{LL}!y_2.M\ \mathsf{else}\ 0 \end{array} ,\ c_{LL}!d_{\triangle} c''_{LL}?x \rangle \\
&\xrightarrow{c''_{LL}!d} \langle M,\ c_{LL}!d.c''_{LL}?d_{\triangle} \rangle
\end{aligned}$$

We elaborate slightly on the second step above. According to the transition rules for binders, we have $\langle c_{LL}?x_2,\ {}_{\triangle} c_{LL}!d \rangle \xrightarrow{c_{LL}?d} \langle [c_{LL} : \mathsf{some}(d)/x_2],\ c_{LL}!d_{\triangle} \rangle$, which gives rise to $\langle \&_{1\vee 2}(c_{LH}?x_1, c_{LL}?x_2),\ {}_{\triangle} c_{LL}!d \rangle \xrightarrow{c_{LL}?d} \langle \&_{1\vee 2}(c_{LH}?x_1, [c_{LL} : \mathsf{some}(d)/x_2]),\ c_{LL}!d_{\triangle} \rangle$. Using the evaluation rules for binders (Table 3.5), we have

$$\&_{1\vee 2}(c_{LH}?x_1, [c_{LL} : \mathsf{some}(d)/x_2]) ::_{\mathbf{tt}} [\mathsf{none}/x_1][\mathsf{some}(d)/x_2].$$

Hence by the transition rule for $b.P$, with b is taken to be $\&_{1\vee 2}(c_{LH}?x_1, [c_{LL} : \mathsf{some}(d)/x_2])$, the second transition is derived. □

For a polarity $\rho \in \{!, ?\}$, we write $\tilde{\rho}$ for its opposite polarity. Thus we have $\tilde{!} = ?$ and $\tilde{?} = !$. We also overload the operator $\widetilde{\cdot}$ on communication actions and write $\widetilde{c!c'}$ for $c?[\,]$ and $\widetilde{c?c'}$ for $c!c'$.

The environment-parameterized semantics has the following correspondence with the original semantics presented in Section 3.1. Roughly speaking, if an action sequence contains at most one external communication, then the action sequence can be directly performed by a process in the original semantics, if

and only if it can be performed by the process in the new semantics, after a corresponding interaction attempt made by the environment.

Lemma 4.1 *For all processes P, P', actions α_1, α_2, ..., and α_n such that there is at most one $i \in \{1,...,n\}$ for which $\alpha_i \neq \tau$, and $\forall i \in \{1,...,n\} : \alpha_i \neq \Box$, histories π such that $\pi = [\pi]_\triangle$, and π_0 such that $\forall i \in \{1,...,n\} : \alpha_i \neq \tau \Rightarrow \pi_0 = \pi.\widetilde{\alpha_i}$, the following are equivalent:*

1. $P \stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow} P'$, *and*
2. $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle \wedge \exists \pi_0' : \delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \stackrel{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$.

The proof of Lemma 4.1 is given in Appendix A.

The following lemma is fairly obvious; we nevertheless state it here since it will be used in our proofs to be presented in Section 4.4. For a communication action $\alpha \neq \tau$, we write $dt(\alpha)$ for the data communicated in α.

Lemma 4.2 *The following statements hold.*

1. *If $\delta \vdash \langle b,\ \pi.\beta\rangle \xrightarrow{\alpha} \langle b',\ \pi'\rangle \wedge \alpha \neq \tau \wedge \alpha \neq \Box$, then $[\pi.\beta]_\triangle = \pi \wedge \beta = \widetilde{\alpha} \wedge \pi' = \pi.(ch(\alpha)\widetilde{\rho(\alpha)}dt(\alpha))_\triangle$.*
2. *If $\delta \vdash \langle P,\ \pi.\beta\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle \wedge \alpha \neq \tau \wedge \alpha \neq \Box$, then $[\pi.\beta]_\triangle = \pi \wedge \beta = \widetilde{\alpha} \wedge \pi' = \pi.(ch(\alpha)\widetilde{\rho(\alpha)}dt(\alpha))_\triangle$.*

Lemma 4.2 can be proved with a straightforward induction on the semantic derivations, with 2 proved with the help of 1.

We will use the more compact notation $\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env},\alpha} \langle P',\ \pi'\rangle$ to represent

$$\exists \pi_0 : \delta \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle \wedge \delta \vdash \langle P,\ \pi_0\rangle \xrightarrow{\alpha} \langle P',\ \pi'\rangle.$$

We will also use $\mathsf{rch}(P)$ (rch is short for "reachable channels") to represent the channel, polarity pairs of all possible communications that can be performed by a derivative of $\langle P,\ \triangle\rangle$ under the strategy δ_{ALL}, i.e.,

$$\mathsf{rch}(P) = \{(c,\rho) \mid \exists P', \pi', c' : \delta_{\mathrm{ALL}} \vdash \langle P,\ \triangle\rangle \rightarrow^* \langle P', \pi'\rangle \xrightarrow{c\rho c'}\}.$$

4.3 δ-Security

In this section, we build up to our noninterference condition for behavioral integrity. We introduce the *classification mapping* $\mathcal{E}$ such that $\mathcal{E}^\circ$ and $\mathcal{E}^\bullet$ keep track of the presence levels and content levels, respectively, for communication channels. In our definitions and propositions, we tacitly assume that all variables that are not explicitly quantified are in fact *universally* quantified.

We start by introducing a way of indexing into traces: $\pi @_i^{c,\rho}$ is (n, c') if the i-th communication over channel c with polarity ρ in π is the n-th communication

overall in π, and the content of the communication is c'. All the indices start with 0. If the number of communications over c with polarity ρ in π is less than or equal to i, then $\pi@_i^{c,\rho}$ is $\bot$. This is formalized in Definition 4.3 and illustrated in Example 4.2.

Definition 4.3 ($\pi@_i^{c,\rho}$) $\pi@_i^{c,\rho} = \pi@_{i,0}^{c,\rho}$, where $i \geq 0$ and $\pi@_{i,0}^{c,\rho}$ is defined by

$$\pi@_{i,n}^{c,\rho} = \begin{cases} \bot & (\mathit{if}\ \pi = \epsilon) \\ (n, c') & (\mathit{if}\ \exists \pi' : \pi = c\rho c'.\pi' \wedge i = 0) \\ \pi'@_{i-1,n+1}^{c,\rho} & (\mathit{if}\ \exists c' : \pi = c\rho c'.\pi' \wedge i \neq 0) \\ \pi'@_{i,n+1}^{c,\rho} & (\mathit{if}\ \exists c'', \rho'', c''' : \pi = c''\rho''c'''.\pi' \wedge (c'' \neq c \vee \rho'' \neq \rho)) \end{cases}$$

Example 4.2 *Consider the trace* $\pi = c_{LL}!d.c''_{LL}?d_{\triangle}$ *left by the environment from Example 4.1. We have* $\pi@_0^{c''_{LL},?} = (1, d)$ *and* $\pi@_i^{c''_{LL},?} = \bot$ *whenever* $i \geq 1$. □

We define the trace correspondence relation $W_{\mathcal{E}}$ as follows, where $|\pi|$ stands for the length of π. This definition builds on another relation $W_{l_2}^{l_1}$ that is explained immediately after it. The presence and content of communications in traces related by $W_{\mathcal{E}}$ are supposed to reflect the integrity levels of their channels.

Definition 4.4 ($W_{\mathcal{E}}$) $\pi_1\ W_{\mathcal{E}}\ \pi_2$ if and only if π_1 and π_2 are finite, $|\pi_1| = |\pi_2|$, and $\forall i \geq 0 : \forall c, \rho : \pi_1@_i^{c,\rho}\ W_{\mathcal{E}^\bullet(c)}^{\mathcal{E}^\circ(c)}\ \pi_2@_i^{c,\rho}$.

In prose, two traces related by $W_{\mathcal{E}}$ are required to have the same finite length, and the i-th occurrences of communication over channel c, with polarity ρ, are required to be related by $W_{\mathcal{E}^\bullet(c)}^{\mathcal{E}^\circ(c)}$, for each c and ρ. The latter relation is in turn defined as follows, where $c'_1 \doteq c'_2$ if and only if $c'_1 = c'_2$ or $c'_2 = [\,]$.

Definition 4.5 ($W_{l_2}^{l_1}$) $(n_1, c'_1)\ W_{l_2}^{l_1}\ (n_2, c'_2)$ if and only if

$$\begin{array}{rcll} (n_1, c'_1)\ W_{l_2}^{l_1}\ (n_2, c'_2) & \text{iff} & (l_1 = H \Rightarrow n_1 = n_2) \wedge (l_2 = H \Rightarrow c'_1 \doteq c'_2) \\ (n_1, c'_1)\ W_{l_2}^{l_1}\ \bot & \text{iff} & l_1 = L \\ \bot\ W_{l_2}^{l_1}\ (n_2, c'_2) & \text{iff} & l_1 = L \\ \bot\ W_{l_2}^{l_1}\ \bot & \text{iff} & \mathsf{true} \end{array}$$

It can be seen that for a channel c with high presence integrity, the i-th occurrences of input/output over c need to have the same overall index in their respective traces. On the other hand, for a channel c with high content integrity, the i-th occurrences of input/output over c need to have equivalent content. This corresponds tightly to our description of "presence integrity" and "content integrity" in Section 4.1. The reason that $\doteq$ is used for relating content, instead of $=$, is the potential existence of holes in traces of the form $...{}_{\triangle}c?[\,]$.

The weak transition introduced in Chapter 3 will not be used directly in our noninterference property, but encapsulated within the transition $\overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}}$ introduced in Definition 4.6. It boils down to the weak transition $\overset{\widehat{\alpha'}}{\Longrightarrow}$ in case α is a communication with high presence integrity or a τ; otherwise to the single transition $\overset{\alpha'}{\longrightarrow}$ (where τ's are not allowed).

Definition 4.6 ($\overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}}$) $\overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}} = \begin{cases} \overset{\alpha'}{\longrightarrow} & (\text{if } \alpha = \square \vee \exists c, \rho, c' : \alpha = c\rho c' \wedge \mathcal{E}^\circ(c) = L) \\ \overset{\widehat{\alpha'}}{\Longrightarrow} & (\text{otherwise}) \end{cases}$

We then define the notion of δ-bisimulation, where $\delta \in \mathbf{Strat}$ is a strategy.

Definition 4.7 (δ-Bisimulation) A δ-bisimulation is a symmetric relation R on configurations such that if $\langle P_1, \pi_1\rangle \; R \; \langle P_2, \pi_2\rangle$, $\delta \vdash \langle P_1, \pi_1\rangle \xrightarrow{\text{env},\alpha} \langle P_1', \pi_1'\rangle$, $\delta \vdash \langle P_2, \pi_2\rangle \xrightarrow{\text{env}} \langle P_2, \pi_{20}\rangle$, and $\pi_1' \; W_{\mathcal{E}} \; \pi_{20}$, then we have

$$\exists P_2', \pi_2', \alpha' : \delta \vdash \langle P_2, \pi_{20}\rangle \overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}} \langle P_2', \pi_2'\rangle \wedge [\pi_1']_\triangle \; W_{\mathcal{E}} \; [\pi_2']_\triangle \wedge \langle P_1', [\pi_1']_\triangle\rangle \; R \; \langle P_2', [\pi_2']_\triangle\rangle.$$

If two configurations $\langle P_1, \; \pi_1\rangle$ and $\langle P_2, \; \pi_2\rangle$ are related by a δ-bisimulation R, then after $\langle P_1, \; \pi_1\rangle$ interacts with the environment δ for one step, and the environment makes an interaction attempt with $\langle P_2, \; \pi_2\rangle$, such that the interaction and the attempt meet the integrity classes of their channels ($\pi_1' \; W_{\mathcal{E}} \; \pi_{20}$), the configuration $\langle P_2, \; \pi_{20}\rangle$ can simulate the interaction made by $\langle P_1, \; \pi_1\rangle$, in a way that meets the integrity classes of the channels involved (as required by $[\pi_1']_\triangle \; W_{\mathcal{E}} \; [\pi_2']_\triangle$).

The definition of δ-bisimulation introduces two universally quantified transitions, before simulating the first one with an existentially quantified transition. This pattern, previously adopted in [NN13], is rare in the literature.

We then define δ-bisimilarity ($\underset{\delta}{\sim}$) as the union of all δ-bisimulations (which is itself a δ-bisimulation). Note that δ-bisimilarity is *not* reflexive. In fact, our noninterference condition identifies the δ-security of a process P with the self-relatedness of $\langle P, \; \triangle\rangle$ in $\underset{\delta}{\sim}$, as stated in Definition 4.8.

Definition 4.8 (δ-Security) A process P is δ-secure, denoted by $Sec_\delta(P)$, if and only if $\langle P, \; \triangle\rangle \underset{\delta}{\sim} \langle P, \; \triangle\rangle$.

To arrive at a better understanding of δ-security, we introduce in Definition 4.9 the notion of *kernel* δ-bisimulation, which constrains the pairs of observational histories further than δ-bisimulation does. Proposition 4.10 then says that kernel δ-bisimulations can be used to characterize δ-security equally well.

Definition 4.9 (Kernel δ-Bisimulation) A δ-bisimulation R is said to be a kernel δ-bisimulation if and only if $\langle P_1,\ \pi_1\rangle\ R\ \langle P_2,\ \pi_2\rangle$ implies $knl_{\mathcal{E}}(\pi_1, \pi_2)$, where $knl_{\mathcal{E}}(\pi_1, \pi_2)$ represents $\pi_1\ W_{\mathcal{E}}\ \pi_2$, $[\pi_1]_{\triangle} = \pi_1$, and $[\pi_2]_{\triangle} = \pi_2$.

Thus the relatedness of some π_1 and some π_2 in a kernel δ-bisimulation provides the further information that π_1 and π_2 are purely observational histories and related by $W_{\mathcal{E}}$. Note that $knl_{\mathcal{E}}(\_, \_)$ is *symmetric* because the position of $\triangle$ in the related histories implies that the $\doteq$ used to define $W_{\mathcal{E}}$ boils down to plain equality.

Proposition 4.10 *There is a δ-bisimulation R such that $\langle P,\ \triangle\rangle\ R\ \langle P,\ \triangle\rangle$, if and only if there is a kernel δ-bisimulation R' such that $\langle P,\ \triangle\rangle\ R'\ \langle P,\ \triangle\rangle$.*

Proof. By Definition 4.9, we directly know that if there is a kernel δ-bisimulation R' such that $\langle P,\ \triangle\rangle\ R'\ \langle P,\ \triangle\rangle$, then there is a δ-bisimulation $R = R'$ relating $\langle P,\ \triangle\rangle$ to itself.

For the other direction, suppose there is δ-bisimulation R in which $\langle P,\ \triangle\rangle$ is related to itself. We simply construct

$$R' = \{(\langle P_1,\ \pi_1\rangle, \langle P_2,\ \pi_2\rangle) \mid (\langle P_1,\ \pi_1\rangle, \langle P_2,\ \pi_2\rangle) \in R \wedge knl_{\mathcal{E}}(\pi_1, \pi_2)\}.$$

It is not difficult to show that R' is a kernel δ-bisimulation relating $\langle P,\ \triangle\rangle$ to itself. □

For a δ-secure process P, the implications of the existence of a *kernel* δ-bisimulation R such that $\langle P,\ \triangle\rangle\ R\ \langle P,\ \triangle\rangle$ are:

1. A communication α with high presence integrity needs to be simulated by a communication over the *same* channel, possibly together with τ's. In case the channel also has high content integrity, the content of the simulating communication should be the same as that of α. If the channel has low content integrity, on the other hand, then the bisimulation should continue under all contents possibly attempted by the environment, that are not necessarily the same as that of α.

2. A communication α with low presence integrity, or a $\Box$-transition, is simulated by a communication over a channel also of low presence integrity, or by a $\Box$-transition. If the channel of α, say c, has high content integrity, and it is being used for the i-th time with polarity ρ, then the content of α needs to agree with the content of the communication occurring on c with polarity ρ for the i-th time in the second execution, *in case that communication exists*. A similar requirement is imposed on the simulating communication, when its channel has high content integrity.

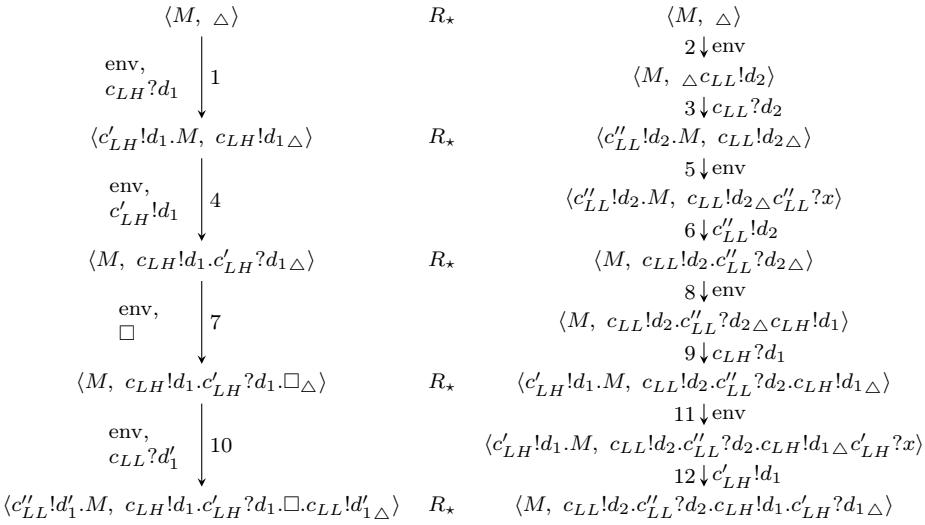


Figure 4.5: Partial Unfolding of the Kernel Bisimulation for Multiplexer

3. A τ can only be simulated by a (possibly empty) sequence of τ's. This is because when a τ-transition is made from a configuration $\langle P_i,\ \pi_i\rangle$ ($i \in \{1,2\}$), the $\triangle$ in π does not move, which is not the case with observable communications. By Definition 4.7 and 4.9, it is obvious that $|\pi'_1| = |\pi_{20}|$. Hence if a τ is not simulated only by τ's, $[\pi'_1]_\triangle$ and $[\pi'_2]_\triangle$ will not have the same length and $[\pi'_1]_\triangle\ W_\mathcal{E}\ [\pi'_2]_\triangle$ will not hold.

The δ_{ALL}-security/insecurity of processes 1-6 in Figure 4.1 of Section 4.1 agrees with the claims based on intuition in the same section, with an unconstrained environment. And process 7 is δ_{ALT}-secure, where δ_{ALT} characterizes an environment that provides content over at least one of c_{LL} and c'_{LL} whenever the process is ready for input from these two alternative channels and attempts interaction on any legitimate abstract binder otherwise:

$$\delta_{\mathrm{ALT}}(\pi) = \begin{cases} \{c_1!d \mid c_1 \in \{c_{LL}, c'_{LL}\} \wedge d \in \mathbf{Dt}\} & (\text{if } \pi = \triangle \vee \exists \pi', d' : \pi = \pi'.c_{HL}?d'_\triangle) \\ AB \setminus \{c?c' \mid c \in \mathbf{Ch} \wedge c' \in \mathbf{Dt}\} & (\text{otherwise}) \end{cases}$$

We demonstrate in Example 4.3 that some of the requirements imposed by $\langle M,\ \triangle\rangle\ R_\star\ \langle M,\ \triangle\rangle$ are fullfilled, to aid in the reader's intuition.

Example 4.3 *Figure 4.5 contains a partial unfolding of* $\langle M,\ \triangle\rangle\ R_\star\ \langle M,\ \triangle\rangle$ *where* $R_\star$ *is a kernel* δ_{ALL}*-bisimulation. For each pair* $\langle P_1,\ \pi_1\rangle$ *and* $\langle P_2,\ \pi_2\rangle$ *related by* $R_\star$ *in Figure 4.5,* $\pi_1\ W_\mathcal{E}\ \pi_2$ *holds. After transitions 1 and 2, the environment has made the attempt to interact with the process on two different channels* c_{LH} *and* c_{LL}*. This is allowed since* $c_{LH}!d_{1\triangle}\ W_\mathcal{E}\ \triangle c_{LL}!d_2$ *holds.*

The process M can indeed perform an input over c_{LH}, resulting in transition 1. This transition needs to be simulated by either an input over c_{LL}, or a $\Box$-transition in case such an input cannot be performed. We are in the former situation and transition 1 is thus simulated by transition 3. Note that according to Definition 4.6, the simulation of low presence communications should be done without using τ's. This is because such simulation is actually used to introduce interference, rather than to demonstrate resilience to it. And τ-transitions are conventionally used to weaken the requirement for a process to be resilient to interference. We then direct our attention to transitions 7, 8 and 9. The environment intentionally resists communication with the process in transition 7. On the other hand, it attempts to feed some content over c_{LH} to the process through transition 8. That content is restricted to d_1 since only then it holds that $c_{LH}!d_1.c'_{LH}?d_1.\Box_{\triangle}\ W_{\mathcal{E}}\ c_{LL}!d_2.c''_{LL}?d_2{}_{\triangle}c_{LH}!d_1$. Intuitively, the input over c_{LH} is blocked for a while in the second execution, but it needs to happen with the same content d_1 since the channel has high content integrity. For transitions 10, 11 and 12, the attempt of the environment to input from the process over channel c'_{LH} in transition 11 is satisfied with the content d_1, resulting in transition 12. The latter transition is a legitimate simulation of transition 10 since the content d_1 is the same as that of transition 4 — the corresponding communication over c'_{LH} in the first execution. $\Box$

We now formally prove that the process M is δ_{ALL}-secure.

Example 4.4 *Let $\phi(\pi_1, \pi_2)$ stand for the condition*

$$
\begin{array}{l}
knl_{\mathcal{E}}(\pi_1, \pi_2) \wedge \\
\forall i \in \{1,2\}, c_{\mathrm{a}}, c_{\mathrm{b}} : \\
\quad (c_{LH}!c_{\mathrm{a}} \textit{ is followed immediately by } c'_{LH}\rho c_{\mathrm{b}} \textit{ in } \pi_i \downarrow \{c_{LH}, c'_{LH}\} \\
\qquad\qquad \Longrightarrow\ \rho =? \wedge c_{\mathrm{a}} = c_{\mathrm{b}}).
\end{array}
$$

In addition, let $Cs(e_1, e_2)$ be the process

$$
\begin{array}{l}
\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{some}(y_1) : \\
\quad c'_{LH}!y_1.M \\
\mathsf{else\ case}\ e_2\ \mathsf{of}\ \mathsf{some}(y_2) : \\
\quad c''_{LL}!y_2.M \\
\mathsf{else}\ 0
\end{array}
$$

We then construct the binary relation R_{sym} that is the symmetric closure of the

following relation $R_\star$.

$$\begin{aligned}
R_\star = \{&(\langle M,\ \pi_1\rangle, \langle M,\ \pi_2\rangle) \mid \phi(\pi_1,\pi_2)\} \cup \\
&\{(\langle Cs(\mathsf{some}(c'_{\mathrm{a}}),\mathsf{none}),\ \pi_1\rangle, \langle M,\ \pi_2\rangle) \mid \pi_1 = ...c_{LH}!c'_{\mathrm{a}\triangle} \wedge \phi(\pi_1,\pi_2)\} \cup \\
&\{(\langle Cs(\mathsf{none},\mathsf{some}(c'_{\mathrm{a}})),\ \pi_1\rangle, \langle M,\ \pi_2\rangle) \mid \pi_1 = ...c_{LL}!c'_{\mathrm{a}\triangle} \wedge \phi(\pi_1,\pi_2)\} \cup \\
&\{(\langle Cs(\mathsf{some}(c'_{\mathrm{a}}),\mathsf{none}),\ \pi_1\rangle, \langle Cs(\mathsf{some}(c'_{\mathrm{b}}),\mathsf{none}),\ \pi_2\rangle) \mid \\
&\qquad \pi_1 = ...c_{LH}!c'_{\mathrm{a}\triangle} \wedge \pi_2 = ...c_{LH}!c'_{\mathrm{b}\triangle} \wedge \phi(\pi_1,\pi_2)\} \cup \\
&\{(\langle Cs(\mathsf{none},\mathsf{some}(c'_{\mathrm{a}})),\ \pi_1\rangle, \langle Cs(\mathsf{none},\mathsf{some}(c'_{\mathrm{b}})),\ \pi_2\rangle) \mid \\
&\qquad \pi_1 = ...c_{LL}!c'_{\mathrm{a}\triangle} \wedge \pi_2 = ...c_{LL}!c'_{\mathrm{b}\triangle} \wedge \phi(\pi_1,\pi_2)\} \cup \\
&\{(\langle Cs(\mathsf{some}(c'_{\mathrm{a}}),\mathsf{none}),\ \pi_1\rangle, \langle Cs(\mathsf{none},\mathsf{some}(c'_{\mathrm{b}})),\ \pi_2\rangle) \mid \\
&\qquad \pi_1 = ...c_{LH}!c'_{\mathrm{a}\triangle} \wedge \pi_2 = ...c_{LL}!c'_{\mathrm{b}\triangle} \wedge \phi(\pi_1,\pi_2)\}
\end{aligned}$$

It can be shown that R_{sym} *is a (kernel)* δ_{ALL}*-bisimulation relating* $\langle M,\ \triangle\rangle$ *to itself, which implies the* δ_{ALL}*-security of* M. □

A partial order can be built on the set **Strat** of environment strategies, characterizing their relative aggressiveness (Definition 4.11), which has its impact on the strength of the security condition (Theorem 4.12).

Definition 4.11 (Aggressiveness of Environment) Environment δ_2 is said to be more aggressive than δ_1, denoted $\delta_1 \leq \delta_2$, if $\forall \pi \in \Pi_\triangle : \delta_1(\pi) \subseteq \delta_2(\pi)$.

Proposition 4.12 (Monotonicity) *δ-bisimilarity is anti-monotonic in δ, i.e., for all δ_1, δ_2 such that $\delta_1 \leq \delta_2$, it holds that $\underset{\delta_2}{\sim} \subseteq \underset{\delta_1}{\sim}$.*

Proof. Given δ_1 and δ_2 such that $\delta_1 \leq \delta_2$ holds, it is not difficult to show that each δ_2-bisimulation also qualifies as a δ_1-bisimulation. □

This monotonicity result may look counter-intuitive since δ appears to be used both positively and negatively in Definition 4.7. However, $\delta \vdash \langle P_2, \pi_{20}\rangle \overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}} \langle P'_2, \pi'_2\rangle$ if and only if $\delta' \vdash \langle P_2, \pi_{20}\rangle \overset{\widehat{\alpha'}}{\underset{\alpha}{\Longrightarrow}} \langle P'_2, \pi'_2\rangle$ for all $\delta' \in$ **Strat**. In other words, δ is not actually *used* in the derivation of the transition sequence from $\langle P_2,\ \pi_{20}\rangle$.

Corollary 4.13 *The permissiveness of δ-security is anti-monotonic in δ, i.e.,*

$$\forall \delta_1, \delta_2 \in \mathbf{Strat} : \delta_1 \leq \delta_2 \wedge Sec_{\delta_2}(P) \ \Rightarrow\ Sec_{\delta_1}(P).$$

Reducing the possibilities of alternative behaviors of an entity under the same situation can be viewed as a form of refinement (called "horizontal" in [RG01, BPR04]). Then Corollary 4.13 can be interpreted as: δ-security is robust under the horizontal refinement of δ.

We will discuss deeper theoretical properties of our security condition in Section 4.4, focusing on δ_{ALL}-security. It will be seen that the most pessimistic assumption about the environment, captured by its most aggressive strategy δ_{ALL}, is in line with classical process-algebraic conditions like SBNDC, and also facilitates the compositional verification of the security property.

4.4 Theoretical Properties of δ-Security

We establish a connection between δ-security and the security property SBNDC [FG01] that we have recast in the form of self-bisimilarity in Section 3.4. We then discuss the compositionality of δ-security.

4.4.1 Degeneration to SBNDC

We show that δ_{ALL}-security coincides with the classical process-algebraic property SBNDC when channels always have the same integrity levels for "presence" and "content", that are taken to be their integrity levels. This result indicates that our security property is a generalization of an existing, mature line of work.

Based on Corollary 3.15, for all processes P, $P \sim_{\triangleright} P$ can be used to characterize $P \in \mathit{SBNDC}$. Now the gap between $P \sim_{\triangleright} P$ and $\langle P,\ \triangle\rangle \underset{\delta_{\text{ALL}}}{\sim} \langle P,\ \triangle\rangle$ lies in the following aspects:

1. the former is expressed with respect to one holistic integrity environment $\mathcal{I}$, while the latter with respect to the presence environment $\mathcal{E}^{\circ}$ and the content environment $\mathcal{E}^{\bullet}$, and
2. the former is expressed in a process-centered way, whereas the latter through the environment strategy.

We formulate the assumptions that the integrity of a channel is either its presence integrity or its content integrity, wherever they agree, with the following characterization of $\mathcal{I}$. Using $\mathcal{E}^{\circ}$ instead of $\mathcal{E}^{\bullet}$ in the first case below would obviously have the same effect.

$$\mathcal{I}(c) = \begin{cases} \mathcal{E}^{\circ}(c) & \text{if } \mathcal{E}^{\circ}(c) = \mathcal{E}^{\bullet}(c) \\ \bot & \text{otherwise} \end{cases}$$

We then have Lemma 4.14 and Lemma 4.15, which are the key lemmas establishing our degeneration result, by bridging the gap between the process-centered $\sim_{\triangleright}$ and the environment-aware $\underset{\delta_{\text{ALL}}}{\sim}$.

Lemma 4.14 *Suppose process P satisfies $\forall c, \rho$ s.t. $(c, \rho) \in \mathsf{rch}(P) : \mathcal{E}^{\circ}(c) = \mathcal{E}^{\bullet}(c)$. Then $P \sim_{\triangleright} P \Rightarrow \langle P,\ \triangle\rangle \underset{\delta_{\text{ALL}}}{\sim} \langle P,\ \triangle\rangle$.*

PROOF. Assume that $P \sim_{\triangleright} P$ holds. We construct the following binary relation on configurations:

$$R_\star \triangleq \{(\langle P_1,\ \pi_1\rangle, \langle P_2,\ \pi_2\rangle) \mid P_1 \sim_{\triangleright} P_2 \ \wedge\ P \to^* P_1 \ \wedge\ P \to^* P_2 \ \wedge\ knl_{\mathcal{E}}(\pi_1, \pi_2)\}$$

This relation is shown to be a (kernel) δ_{ALL}-bisimulation as follows. First, $R_\star$ is symmetric, which follows from the symmetry of $\_ \sim_{\triangleright} \_$ and $knl_{\mathcal{E}}(\_, \_)$. Second, take arbitrary π_1', α, π_{20} such that

$$\begin{aligned} &\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_1\rangle \xrightarrow{\mathrm{env},\alpha} \langle P_1',\ \pi_1'\rangle \\ &\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi_{20}\rangle \\ &\pi_1'\ W_{\mathcal{E}}\ \pi_{20} \end{aligned} \tag{4.1}$$

By Lemma 4.1, $P_1 \xrightarrow{\alpha} P_1'$. According to the precondition, there are the following cases about α.

- $\alpha = c\rho c'$ for some c, ρ, c' such that $\mathcal{E}^\circ(c) = L \wedge \mathcal{E}^\bullet(c) = L$. By Lemma 4.2, we have $\pi_1' = \pi_1.c\widetilde{\rho}c'_{\triangle}$. By $P_1 \sim_{\triangleright} P_2$, there exists some P_2' such that $P_2' \equiv P_2$ and $P_1' \sim_{\triangleright} P_2'$.
  - If $\pi_{20} = \pi_2.\Box$, then by Lemma 4.2, it is not difficult to see that $\langle P_2,\ \pi_{20}\rangle$ cannot perform an output or input. Hence there exists the transition
    $$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{\Box} \langle P_2',\ \pi_2.\Box_{\triangle}\rangle.$$
    By $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$, we have $\pi_1'\ W_{\mathcal{E}}\ \pi_2.\Box_{\triangle}$, and thus $knl_{\mathcal{E}}(\pi_1', \pi_2.\Box_{\triangle})$ and
    $$(\langle P_1',\ \pi_1'\rangle, \langle P_2',\ \pi_2.\Box_{\triangle}\rangle) \in R_\star.$$
  - On the other hand, suppose $\pi_{20} = \pi_2.c''\rho'c'''$ for some c'', ρ' and c'''.
    - If $\langle P_2,\ \pi_{20}\rangle$ cannot perform any output or input, then there exists the transition
      $$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{\Box} \langle P_2'',\ \pi_2.\Box_{\triangle}\rangle$$
      for some $P_2'' \equiv P_2$, and the subsequent reasoning is similar to the case where $\pi_{20} = \pi_2.\Box$.
    - If there exists transition
      $$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{c''\tilde{\rho}'c''''} \langle P_2''',\ \pi_2'\rangle \tag{4.2}$$
      for some P_2''' and c'''', then by $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$ we have $\mathcal{E}^\circ(c'') = L$. By the precondition of this lemma we then have $\mathcal{E}^\bullet(c'') = L$. By Lemma 4.1, $P_2 \xrightarrow{c''\tilde{\rho}'c''''} P_2'''$. Then $P_2' \xrightarrow{c''\tilde{\rho}'c''''} P_2'''$ since $P_2' \equiv P_2$. Since $P_1' \sim_{\triangleright} P_2'$, we have $P_2' \sim_{\triangleright} P_1'$ by the symmetry of $\sim_{\triangleright}$. We thus have $P_2''' \sim_{\triangleright} P_1''$ for some $P_1'' \equiv P_1'$. Therefore $P_1'' \sim_{\triangleright} P_2'''$, and $P_1' \sim_{\triangleright} P_2'''$ by Lemma 3.13. It is not difficult to see that $\pi_1'\ W_{\mathcal{E}}\ \pi_2'$ and furthermore $knl_{\mathcal{E}}(\pi_1', \pi_2')$. Therefore we have
      $$(\langle P_1',\ \pi_1'\rangle, \langle P_2''',\ \pi_2'\rangle) \in R_\star.$$
      Hence transition (4.1) can be simulated by transition (4.2).

- $\alpha = c\rho c'$ for some c, ρ, c' such that $\mathcal{E}^\circ(c) = H \wedge \mathcal{E}^\bullet(c) = H$. By Lemma 4.2, $\pi_1' = \pi_1.c\widetilde{\rho c'}_\triangle$. By $P_1 \sim_\triangleright P_2$, there exists $P_2 \stackrel{\widehat{\alpha}}{\Longrightarrow} P_2'$ as simulation of the transition $P_1 \stackrel{\alpha}{\longrightarrow} P_1'$. Since $\pi_1' \ W_\mathcal{E} \ \pi_{20}$, we have $\pi_{20} = \pi_2.\widetilde{\alpha}$. Hence by Lemma 4.1, we have

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \stackrel{\widehat{\alpha}}{\Longrightarrow} \langle P_2',\ \pi_2'\rangle$$

for some π_2'. It is not difficult to see that $\pi_1' \ W_\mathcal{E} \ \pi_2'$ and furthermore $knl_\mathcal{E}(\pi_1', \pi_2')$. Therefore we have

$$(\langle P_1',\ \pi_1'\rangle, \langle P_2',\ \pi_2'\rangle) \in R_\star.$$

- $\alpha = \tau$. By $P_1 \sim_\triangleright P_2$, there exists $P_2 \stackrel{\widehat{\tau}}{\Longrightarrow} P_2'$ as simulation of the transition $P_1 \stackrel{\alpha}{\longrightarrow} P_1'$. By Lemma 4.1, we have

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \stackrel{\widehat{\tau}}{\Longrightarrow} \langle P_2',\ \pi_2'\rangle$$

as simulation of transition (4.1) in $R_\star$. It is not difficult to see that $[\pi_1']_\triangle = \pi_1$, $[\pi_2']_\triangle = \pi_2$, and thus $knl_\mathcal{E}([\pi_1']_\triangle, [\pi_2']_\triangle)$ holds. Therefore we have

$$(\langle P_1',\ [\pi_1']_\triangle\rangle, \langle P_2',\ [\pi_2']_\triangle\rangle) \in R_\star.$$

- $\alpha = \Box$. We have $\pi_1' = \pi_1.\Box_\triangle$ and $P_1' \equiv P_1$. Then for $\pi_1' \ W_\mathcal{E} \ \pi_{20}$ to hold, either $\pi_{20} = \pi_2.\Box$ or $\pi_{20} = \pi_2.c\rho c'$ for some c, ρ and c', where $\mathcal{E}^\circ(c) = L$.
  - If $\pi_{20} = \pi_2.\Box$, then there exists the transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \stackrel{\Box}{\longrightarrow} \langle P_2,\ \pi_2.\Box_\triangle\rangle.$$

    We have $P_1' \sim_\triangleright P_2$ by $P_1' \equiv P_1$ and Lemma 3.13. We also have $\pi_1' \ W_\mathcal{E} \ \pi_2.\Box_\triangle$, $\pi_1' = [\pi_1']_\triangle$ and $\pi_2.\Box_\triangle = [\pi_2.\Box_\triangle]_\triangle$. We therefore have $knl_\mathcal{E}(\pi_1', \pi_2.\Box_\triangle)$ and

$$(\langle P_1',\ \pi_1'\rangle, \langle P_2,\ \pi_2.\Box_\triangle\rangle) \in R_\star.$$

  - If $\pi_{20} = \pi_2.c\rho c'$ with $\mathcal{E}^\circ(c) = L$, depending on whether $\langle P_2,\ \pi_{20}\rangle$ can perform communication on c, we have the following cases.
    - If $\langle P_2,\ \pi_{20}\rangle$ cannot perform communication on c, then the reasoning is similar to the case where $\pi_{20} = \pi_2.\Box$.
    - If $\langle P_2,\ \pi_{20}\rangle$ can perform communication on c, then there exists transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \stackrel{\alpha'}{\longrightarrow} \langle P_2',\ \pi_2'\rangle. \tag{4.3}$$

      By Lemma 4.2, $\widetilde{\alpha'} = c\rho c'$, and $\pi_2' = \pi_2.c\rho dt(\alpha')_\triangle$. By Lemma 4.1, we have $P_2 \stackrel{\alpha'}{\longrightarrow} P_2'$. Hence there exists $P_1'' \equiv P_1$ such that $P_1'' \sim_\triangleright P_2'$. Therefore we have $P_1 \sim_\triangleright P_2'$ by Lemma 3.13. By $P_1' \equiv P_1$, we obtain $P_1' \sim_\triangleright P_2'$. Since $\pi_1' = [\pi_1']_\triangle$, $\pi_2.c\rho dt(\alpha')_\triangle = [\pi_2.c\rho dt(\alpha')_\triangle]_\triangle$ and $\pi_1' \ W_\mathcal{E} \ \pi_2.c\rho dt(\alpha')_\triangle$ hold, we have

$$(\langle P_1',\ \pi_1'\rangle, \langle P_2',\ \pi_2'\rangle) \in R_\star.$$

      Hence transition 4.1 can be simulated by transition 4.3.

Finally, it is not difficult to see that $(\langle P,\ \triangle\rangle, \langle P,\ \triangle\rangle) \in R_\star$. Hence we have $\langle P,\ \triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P,\ \triangle\rangle$. □

Lemma 4.15 *Suppose process P satisfies $\forall c, \rho$ s.t. $(c,\rho) \in \mathsf{rch}(P) : \mathcal{E}^\circ(c) = \mathcal{E}^\bullet(c)$. Then $\langle P,\ \triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P,\ \triangle\rangle \Rightarrow P \sim_\rhd P$.*

Proof. Assume that $\langle P,\ \triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P,\ \triangle\rangle$ holds. Construct the following binary relation on processes:

$$R_\star \triangleq \{(P_1,P_2) \mid \langle P_1,\ \pi_1\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2\rangle \ \wedge\ P \to^* P_1 \ \wedge\ P \to^* P_2 \ \wedge\ knl_{\mathcal{E}}(\pi_1,\pi_2)\}$$

This relation is shown to be a $\rhd$-bisimulation as follows. First, $R_\star$ is symmetric, which follows from the symmetry of $\_ \underset{\delta_{\mathrm{ALL}}}{\sim} \_$ and $knl_{\mathcal{E}}(\_,\_)$. Second, suppose we have the following transition:

$$P_1 \xrightarrow{\alpha} P_1' \tag{4.4}$$

We know by Lemma 4.1 that there exists some π_{10} such that $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_1\rangle \xrightarrow{\mathrm{env}} \langle P_1,\ \pi_{10}\rangle$ and $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\alpha} \langle P_1',\ \pi_1'\rangle$. The following case analysis is performed on α.

- Case $\mathcal{I}(ch(\alpha)) = L$. By Lemma 4.2, we have $\pi_{10} = \pi_1.\widetilde{\alpha}$, and $\pi_1' = \pi_1.ch(\alpha)\widetilde{\rho(\alpha)dt(\alpha)}$. By the definition of $\mathcal{I}$ and the precondition, we have $\mathcal{E}^\circ(ch(\alpha)) = L$ and $\mathcal{E}^\bullet(ch(\alpha)) = L$. Take $\pi_{20} = \pi_2.\Box$. Since $\Box \in \delta_{\mathrm{ALL}}([\pi_2]_\triangle)$, there exists the transition $\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi_{20}\rangle$. By $\pi_1\ W_{\mathcal{E}}\ \pi_2$, and the structure of π_1' and π_{20}, we have $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$. By $\langle P_1,\ \pi_1\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2\rangle$, the transition from $\langle P_1,\ \pi_{10}\rangle$ can be simulated from $\langle P_2,\ \pi_{20}\rangle$. This simulation must be with
$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{\Box} \langle P_2,\ \pi_2'\rangle,$$
resulting in $\langle P_1',\ \pi_1'\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2.\Box_\triangle\rangle$ and $\pi_1'\ W_{\mathcal{E}}\ \pi_2.\Box_\triangle$. We thus have $knl_{\mathcal{E}}(\pi_1',\Box_\triangle)$. It is obvious that $P_2 \equiv P_2$ and $(P_1',P_2) \in R_\star$. Hence the transition (4.4) can be simulated by inaction of P_2 in $R_\star$.
- Case $\mathcal{I}(ch(\alpha)) = H$. By Lemma 4.2, we have $\pi_{10} = \pi_1.\widetilde{\alpha}$, and $\pi_1' = \pi_1.ch(\alpha)\widetilde{\rho(\alpha)dt(\alpha)}$. By the definition of $\mathcal{I}$ and the precondition, we have $\mathcal{E}^\circ(ch(\alpha)) = H$ and $\mathcal{E}^\bullet(ch(\alpha)) = H$. We have $\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi_2.\widetilde{\alpha}\rangle$ since $\widetilde{\alpha} \in \delta_{\mathrm{ALL}}([\pi_2]_\triangle)$. Take $\pi_{20} = \pi_2.\widetilde{\alpha}$, we have $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$. By $\langle P_1,\ \pi_1\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2\rangle$, the transition from $\langle P_1,\ \pi_{10}\rangle$ can be simulated from $\langle P_2,\ \pi_{20}\rangle$. Suppose this simulation is
$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xRightarrow{\widehat{\alpha'}} \langle P_2',\ \pi_2'\rangle,$$

resulting in π_2' such that $\pi_1' \; W_{\mathcal{E}} \; \pi_2'$. By Lemma 4.2, $\pi_{20} = \pi_2.\widetilde{\alpha'}$, and $\pi_2' = ch(\alpha')\widetilde{\rho(\alpha')}dt(\alpha')$. Hence $\pi_1 \; W_{\mathcal{E}} \; \pi_2$ and $\pi_1' \; W_{\mathcal{E}} \; \pi_2'$ give $ch(\alpha) = ch(\alpha')$, $\rho(\alpha) = \rho(\alpha')$ and $dt(\alpha) = dt(\alpha')$, which implies $\alpha' = \alpha$. By Lemma 4.1, there exists some P_2' such that

$$P_2 \stackrel{\widehat{\alpha}}{\Longrightarrow} P_2' \tag{4.5}$$

It is not difficult to see that $(P_1', P_2') \in R_\star$. Hence the transition (4.4) can be simulated by transition (4.5) in $R_\star$.

- Case $\alpha = \tau$. Take arbitrary π_{20} such that $\delta_{\mathrm{ALL}} \vdash \langle P_2, \; \pi_2 \rangle \stackrel{\mathrm{env}}{\longrightarrow} \langle P_2, \; \pi_{20} \rangle$ and $\pi_1' \; W_{\mathcal{E}} \; \pi_{20}$. By $\langle P_1, \; \pi_1 \rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2, \; \pi_2 \rangle$, there exists the transition $\delta_{\mathrm{ALL}} \vdash \langle P_2, \; \pi_2 \rangle \stackrel{\widehat{\tau}}{\Longrightarrow} \langle P_2', \; \pi_2' \rangle$ such that $\langle P_1', \; [\pi_1']_\triangle \rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2', \; [\pi_2']_\triangle \rangle$, and $[\pi_1']_\triangle \; W_{\mathcal{E}} \; [\pi_2']_\triangle$. It is not difficult to see that $[\pi_1']_\triangle = \pi_1$ and $[\pi_2']_\triangle = \pi_2$. Hence we have $(P_1', P_2') \in R_\star$. By Lemma 4.1, we also have

  $$P_2 \stackrel{\widehat{\tau}}{\Longrightarrow} P_2',$$

  which qualifies as a simulation of transition (4.4) in $R_\star$.

Now since $(P, P) \in R_\star$, we have $P \sim_\triangleright P$. □

Using Lemma 4.14 and Lemma 4.15 above, and Corollary 3.15, the main theorem of this section can finally be obtained.

Theorem 4.16 *Suppose process P satisfies $\forall c, \rho$ s.t. $(c, \rho) \in \mathsf{rch}(P) : \mathcal{E}^\circ(c) = \mathcal{E}^\bullet(c)$. Then $P \in SBNDC \iff Sec_{\delta_{\mathrm{ALL}}}(P)$.*

The degeneration result presented above demonstrates that SBNDC, as one of the classical process-algebraic noninterference properties, actually has the implicit (pessimistic) assumption of the most aggressive environment.

4.4.2 Compositionality

We study the compositionality of δ_{ALL}-security under restriction and parallel composition.

Restriction In the case of $(\nu c)\cdot$, we have the following theorem saying that the δ_{ALL}-security of P implies the δ_{ALL}-security of $(\nu c)P$ as long as the channel c cannot itself be communicated as data.

Theorem 4.17 *If $Sec_{\delta_{\mathrm{ALL}}}(P)$, and $c \notin \mathbf{Dt}$, then $Sec_{\delta_{\mathrm{ALL}}}((\nu c)P)$.*

PROOF. For convenience in naming, we rename the channel in the theorem statement into c_0. Construct the following relation, where $Ch(\pi)$ is the set of all channels appearing in π:

$$\begin{aligned} R_\star = \{(\langle P_1',\ \pi_1\rangle, \langle P_2',\ \pi_2\rangle) \mid \\ & \delta_{\mathrm{ALL}} \vdash \langle P,\ \triangle\rangle \to^* \langle P_1,\ \pi_1\rangle \ \wedge\ \delta_{\mathrm{ALL}} \vdash \langle P,\ \triangle\rangle \to^* \langle P_2,\ \pi_2\rangle \ \wedge \\ & \langle P_1,\ \pi_1\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2\rangle \ \wedge\ knl_{\mathcal{E}}(\pi_1, \pi_2) \ \wedge \\ & c_0 \notin Ch(\pi_1) \cup Ch(\pi_2) \ \wedge\ P_1' \equiv (\nu c_0)P_1 \ \wedge\ P_2' \equiv (\nu c_0)P_2\}. \end{aligned}$$

It is obvious that $(\langle(\nu c_0)P,\ \triangle\rangle, \langle(\nu c_0)P,\ \triangle\rangle) \in R_\star$. We next show that $R_\star$ is a δ_{ALL}-bisimulation. Take an arbitrary pair $(\langle P_1^\circ,\ \pi_1\rangle, \langle P_2^\circ,\ \pi_2\rangle)$ from $R_\star$. We have $P_1^\circ \equiv (\nu c_0)P_1$ and $P_2^\circ \equiv (\nu c_0)P_2$ for some P_1 and P_2, such that

$$\langle P_1,\ \pi_1\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2,\ \pi_2\rangle \tag{4.6}$$

$$knl_{\mathcal{E}}(\pi_1, \pi_2) \tag{4.7}$$

$$c_0 \notin Ch(\pi_1) \cup Ch(\pi_2) \tag{4.8}$$

Suppose

$$\delta_{\mathrm{ALL}} \vdash \langle P_1^\circ,\ \pi_1\rangle \xrightarrow{\mathrm{env},\alpha} \langle P_1'',\ \pi_1'\rangle, \tag{4.9}$$

$\pi_{20} = \pi_2.\alpha'$, and $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$. Then there exists some P_1' such that $P_1'' \equiv (\nu c_0)P_1'$, and

$$\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_1\rangle \xrightarrow{\mathrm{env},\alpha} \langle P_1',\ \pi_1'\rangle. \tag{4.10}$$

We distinguish between three cases for α.

- $\alpha = c\rho c_1$ where $\mathcal{E}^\circ(c) = H$. We have $\pi_1' = \pi_1.c\widetilde{\rho}c_{1\triangle}$; hence $\pi_{20} = \pi_2.c\widetilde{\rho}c_2'$ for some c_2'. By (4.6), there exists the transition

  $$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \overset{\widehat{c\rho c_2}}{\Longrightarrow} \langle P_2',\ \pi_2.c\widetilde{\rho}c_{2\triangle}\rangle, \tag{4.11}$$

  for some c_2 and P_2', such that

  $$\langle P_1',\ \pi_1'\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2',\ \pi_2.c\widetilde{\rho}c_{2\triangle}\rangle,$$

  and

  $$knl_{\mathcal{E}}(\pi_1', \pi_2.c\widetilde{\rho}c_{2\triangle}).$$

  From transition (4.9) we know that $c \neq c_0$ and $c_1 \neq c_0$. From $c_0 \notin \mathbf{Dt}$, we know that $c_2 \neq c_0$. Hence $c_0 \notin Ch(\pi_1') \cup Ch(\pi_2.c\widetilde{\rho}c_{2\triangle})$. And we also have

  $$\delta_{\mathrm{ALL}} \vdash \langle P_2^\circ,\ \pi_{20}\rangle \overset{\widehat{c\rho c_2}}{\Longrightarrow} \langle(\nu c_0)P_2',\ \pi_2.c\widetilde{\rho}c_{2\triangle}\rangle, \tag{4.12}$$

  and

  $$(\langle P_1'',\ \pi_1'\rangle, \langle(\nu c_0)P_2',\ \pi_2.c\widetilde{\rho}c_{2\triangle}\rangle) \in R_\star.$$

- $\alpha = c\rho c_1$ where $\mathcal{E}^\circ(c) = L$. We have $\pi_1' = \pi_1.c\widetilde{\rho}c_{1\triangle}$; hence $\pi_{20} = \pi_2.c'\rho'c_2'$ for some c', ρ' and c_2' (that can be $[\,]$) or $\pi_{20} = \pi_2.\Box$. In the first case, by $\pi_1'\ W_{\mathcal{E}}\ \pi_{20}$, we have $\mathcal{E}^\circ(c') = L$.

– Suppose $\delta_{\mathrm{ALL}} \vdash \langle P_2^\circ,\ \pi_{20}\rangle \xrightarrow{c'\rho' c_2''} \langle P_2'',\ \pi_2'\rangle$ for some c_2'', P_2'' and π_2'. We have $c' \neq c_0$, $P_2'' \equiv (\nu c_0)P_2'$ for some P_2', such that $\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{c'\rho' c_2''} \langle P_2',\ \pi_2'\rangle$. We also have $\pi_2' = \pi_2.c'\widetilde{\rho'}c_{2\triangle}''$. By (4.6), there exists a transition from $\langle P_2,\ \pi_{20}\rangle$ as simulation of (4.10). Since the communication action $c'\rho'c_2''$ can be performed from $\langle P_2,\ \pi_{20}\rangle$, the simulation must be of the form

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{c'\rho' c_2} \langle P_2''',\ \pi_2.c'\widetilde{\rho'}c_{2\triangle}\rangle \tag{4.13}$$

for some c_2 and P_2'''. We have

$$\langle P_1',\ \pi_1'\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2''',\ \pi_2.c'\widetilde{\rho'}c_{2\triangle}\rangle,$$

and

$$knl_{\mathcal{E}}(\pi_1', \pi_2.c'\widetilde{\rho'}c_{2\triangle}).$$

By transition (4.9) we have $c_0 \neq c$ and $c_0 \neq c_1$. By $c_0 \notin \mathbf{Dt}$, we have $c_2 \neq c_0$. Considering $c' \neq c_0$ obtained earlier, we have $c_0 \notin Ch(\pi_1') \cup Ch(\pi_2.c'\widetilde{\rho'}c_{2\triangle})$. There also exists the transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_2^\circ,\ \pi_{20}\rangle \xrightarrow{c'\rho' c_2} \langle (\nu c_0)P_2''',\ \pi_2.c'\widetilde{\rho'}c_{2\triangle}\rangle,$$

which qualifies as the simulation of transition (4.9), with

$$(\langle P_1'',\ \pi_1'\rangle, \langle (\nu c_0)P_2''',\ \pi_2.c'\widetilde{\rho'}c_{2\triangle}\rangle) \in R_\star.$$

– Suppose

$$\delta_{\mathrm{ALL}} \vdash \langle P_2^\circ,\ \pi_{20}\rangle \xrightarrow{\Box} \langle P_2'',\ \pi_2.\Box_\triangle\rangle \tag{4.14}$$

for some P_2''. We have $P_2'' \equiv P_2^\circ \equiv (\nu c_0)P_2$. There exists the transition $\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi_2.\Box\rangle$. It holds that $\pi_1'\ W_{\mathcal{E}}\ \pi_2.\Box$. Hence by (4.6), there exists a transition from $\langle P_2,\ \pi_2.\Box\rangle$ that simulates transition (4.10). This transition must be of the form

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2.\Box\rangle \xrightarrow{\Box} \langle P_2''',\ \pi_2.\Box_\triangle\rangle,$$

where $P_2''' \equiv P_2$. We have $\langle P_1',\ \pi_1'\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_2''',\ \pi_2.\Box_\triangle\rangle$, $knl_{\mathcal{E}}(\pi_1', \pi_2.\Box_\triangle)$ and $c_0 \notin Ch(\pi_1') \cup Ch(\pi_2.\Box_\triangle)$. It also holds that $P_2'' \equiv (\nu c_0)P_2'''$. Hence transition (4.14) qualifies as the simulation of transition (4.9), with

$$(\langle P_1'',\ \pi_1'\rangle, \langle P_2'',\ \pi_2.\Box_\triangle\rangle) \in R_\star.$$

- $\alpha = \tau$. Trivial.
- $\alpha = \Box$. We have $P_1'' \equiv (\nu c_0)P_1$. We also have

$$\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_1\rangle \xrightarrow{\mathrm{env}} \langle P_1,\ \pi_1.\Box\rangle \xrightarrow{\Box} \langle P_1,\ \pi_1.\Box_\triangle\rangle. \tag{4.15}$$

– Suppose $\langle P_2^\circ,\ \pi_{20}\rangle$ cannot perform any communication, which means there exists the transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_2^\circ,\ \pi_{20}\rangle \xrightarrow{\Box} \langle (\nu c_0)P_2,\ \pi_2.\Box_\triangle\rangle. \tag{4.16}$$

Since $\Box \in \delta_{\mathrm{ALL}}(\pi_2)$, we have $\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi'_{20}\rangle$, where $\pi'_{20} = \pi_2.\Box$. By (4.6), and $\pi_1.\Box_\triangle\ W_{\mathcal{E}}\ \pi'_{20}$, the transition sequence (4.15) can be simulated. The simulation must be of the form

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2.\Box\rangle \xrightarrow{\Box} \langle P'_2,\ \pi_2.\Box_\triangle\rangle,$$

where $P'_2 \equiv P_2$. We have

$$\langle P_1,\ \pi_1.\Box_\triangle\rangle\ W_{\mathcal{E}}\ \langle P'_2,\ \pi_2.\Box_\triangle\rangle,$$

$knl_{\mathcal{E}}(\pi_1.\Box_\triangle, \pi_2.\Box_\triangle)$, and $c \notin Ch(\pi_1.\Box_\triangle) \cup Ch(\pi_2.\Box_\triangle)$. It also holds that $(\nu c_0)P_2 \equiv (\nu c_0)P'_2$.
Therefore we have

$$(\langle P''_1,\ \pi_1.\Box_\triangle\rangle, \langle (\nu c_0)P_2,\ \pi_2.\Box_\triangle\rangle) \in R_\star,$$

which means transition (4.9) can be simulated by (4.16) in $R_\star$.

– Suppose $\langle P_2^\circ,\ \pi_{20}\rangle$ can perform a communication. The reasoning needed is analogous to the first sub-case of $\alpha = c\rho c_1$ with $\mathcal{E}^\circ(c) = L$.

This completes the proof. □

To see the relevance of the condition $c \notin \mathbf{Dt}$ of Theorem 4.17, consider the following example.

Example 4.5 *Suppose $c_0 \in \mathbf{Dt}$. The process $(\nu c_0)(c_{HL}?x)$ is δ_{ALL}-insecure, despite the δ_{ALL}-security of the process $c_{HL}?x$. Intuitively, the presence of the input over c_{HL} is interfered with by whether the low integrity input content is c_0.*

Formally, assume per absurdum that $\langle (\nu c_0)(c_{HL}?x),\ \triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle (\nu c_0)(c_{HL}?x),\ \triangle\rangle$. We then have

$$\delta_{\mathrm{ALL}} \vdash \langle (\nu c_0)(c_{HL}?x),\ \triangle\rangle \xrightarrow{\mathrm{env}, c_{HL}?d_0} \langle (\nu c_0)(0),\ c_{HL}!d_0\triangle\rangle \tag{4.17}$$
$$\delta_{\mathrm{ALL}} \vdash \langle (\nu c_0)(c_{HL}?x),\ \triangle\rangle \xrightarrow{\mathrm{env}} \langle (\nu c_0)(c_{HL}?x),\ \triangle c_{HL}!c_0\rangle$$
$$c_{HL}!d_0\triangle\ W_{\mathcal{E}}\ \triangle c_{HL}!c_0$$

However, the only possible transition from $\langle (\nu c_0)(c_{HL}?x),\ \triangle c_{HL}!c_0\rangle$ is of the form

$$\delta_{\mathrm{ALL}} \vdash \langle (\nu c_0)(c_{HL}?x),\ \triangle c_{HL}!c_0\rangle \xrightarrow{\Box} \langle P',\ \Box_\triangle\rangle,$$

where $P' \equiv (\nu c_0)(c_{HL}?x)$, which cannot be a proper simulation of transition (4.17). □

The condition $c \notin \mathbf{Dt}$ of Theorem 4.17 is by no means a *necessary condition*. For instance, when $c_0 \in \mathbf{Dt}$, despite the δ_{ALL}-insecurity of the process $(\nu c_0)(c_{HL}?x)$ of Example 4.5, the process $(\nu c_0)(c_{LL}?x)$ is still δ_{ALL}-secure.

Parallel Composition The notion of δ_{ALL}-security is not fully compositional under $\cdot \mid \cdot$. Nevertheless, this is key to spotting the insecurity of the example process 4,

$$c_{LH}?x[y].c'_{LH}!y \mid c'_{LH}!d,$$

given in Section 4.1, since the processes $c_{LH}?x[y].c'_{LH}!y$ and $c'_{LH}!d$ are themselves δ_{ALL}-secure. We give formal arguments for the insecurity of this parallel composition below.

Example 4.6 *Assume per absurdum that* $Sec_{\delta_{\mathrm{ALL}}}(c_{LH}?x[y].c'_{LH}!y \mid c'_{LH}!d)$ *holds, that is, there exists a* δ_{ALL}*-bisimulation relation* $R_\star$ *such that*

$$\langle c_{LH}?x[y].c'_{LH}!y|c'_{LH}!d,\ \triangle\rangle\ R\ \langle c_{LH}?x[y].c'_{LH}!y|c'_{LH}!d,\ \triangle\rangle$$

Given the existence of the transition

$$\delta_{\mathrm{ALL}} \vdash \langle c_{LH}?x[y].c'_{LH}!y|c'_{LH}!d,\ \triangle\rangle \xrightarrow{\mathrm{env},c_{LH}?d'} \langle c'_{LH}!d'|c'_{LH}!d,\ c_{LH}!d'_{\triangle}\rangle,$$

and the fact that $c_{LH}!d'_{\triangle}\ W_{\mathcal{E}}\ \triangle\Box$*, it is required that*

$$\langle c'_{LH}!d'|c'_{LH}!d, c_{LH}!d'_{\triangle}\rangle\ R_\star\ \langle c_{LH}?x[y].c'_{LH}!y|c'_{LH}!d, \Box_{\triangle}\rangle.$$

Now given the existence of the transition

$$\delta_{\mathrm{ALL}} \vdash \langle c'_{LH}!d'|c'_{LH}!d,\ c_{LH}!d'_{\triangle}\rangle \xrightarrow{\mathrm{env},c'_{LH}!d'} \langle c'_{LH}!d,\ c_{LH}!d'.c'_{LH}?d'_{\triangle}\rangle,$$

and the fact that $c_{LH}!d'.c'_{LH}?d'_{\triangle}\ W_{\mathcal{E}}\ \Box_{\triangle}c'_{LH}?[\,]$*, the communication behaviour* $c'_{LH}!d'$ *of the left-hand process needs to be simulated by the right-hand process, resulting in environment histories still related by* $W_{\mathcal{E}}$*. However, this is impossible. The process* $c_{LH}?x[y].c'_{LH}!y \mid c'_{LH}!d$ *is therefore* δ_{ALL}*-insecure.* □

We then discuss *sufficient conditions* for δ_{ALL}-security to be compositional under $\cdot \mid \cdot$.

A process P is *deterministic* with respect to output over a channel c, denoted by $\mathsf{det}(P, c)$, if

$$\delta_{\mathrm{ALL}} \vdash \langle P,\ \triangle\rangle \rightarrow^* \langle P',\ \pi'\rangle \wedge (\forall i \in \{1,2\} : \delta_{\mathrm{ALL}} \vdash \langle P',\ \pi'\rangle \xrightarrow{c!c'_i} \langle P'_i,\ \pi'_i\rangle)$$
$$\Downarrow$$
$$c'_1 = c'_2$$

We then have the following theorem for the compositionality of δ_{ALL}-security.

Theorem 4.18 (Compositionality) *Suppose* $\{\vec{c'}\} \cap \mathbf{Dt} = \emptyset$, $Sec_{\delta_{\mathrm{ALL}}}(P_1)$, *and* $Sec_{\delta_{\mathrm{ALL}}}(P_2)$. *Then we have* $Sec_{\delta_{\mathrm{ALL}}}((\nu\vec{c'})(P_1|P_2))$, *provided*

$$\begin{array}{l}\forall i \in \{1,2\}, c_{LH} \in \mathbf{Ch}: \\ \qquad (c_{LH},\rho_1) \in \mathsf{rch}(P_i) \wedge (c_{LH},\rho_2) \in \mathsf{rch}(P_{3-i}) \\ \qquad\qquad \Downarrow \\ \qquad\quad \rho_1 \neq \rho_2 \wedge \mathsf{det}(P_i, c_{LH}) \wedge c_{LH} \in \{\vec{c'}\}\end{array}$$

Proof. See Appendix A. □

In prose, given two processes P_1 and P_2 that are both δ_{ALL}-secure, the process $(\nu\vec{c'})(P_1|P_2)$ is δ_{ALL}-secure under the following conditions:

1. No LH-channels are used by both P_1 and P_2 with the same polarity (note that the process 4 given in Section 4.1 does not meet this requirement).

2. For each LH-channel c used by P_1 and P_2 with different polarities, P_1 and P_2 must be deterministic with respect to output on c, and c must be among the set $\{\vec{c'}\}$ of channels over which there is a top-level restriction; thus the input side always sources from the output side, never from the environment.

3. No channel in $\vec{c'}$ can be communicated as data.

The "top-level restriction" required in Definition 2 is motivated by the example below.

Example 4.7 *Consider the process* $P_\star$, *which is*

$$c_{LL}?x.c_{LH}!d_1 \mid c_{LH}?x_1[y_1].c'_{LH}!y_1.$$

In $P_\star$, *no* LH*-channel is used by both parallel components with the same polarity, and determinacy with respect to output is also enjoyed. However, there is no top-level restriction over* c_{LH}, *which is used in both parallel processes with opposite polarities. It is not difficult to see that* $c_{LL}?x.c_{LH}!d_1$ *and* $c_{LH}?x[y].c'_{LH}!y$ *are themselves* δ_{ALL}*-secure. However, we can show by contradiction that* $P_\star$ *is not* δ_{ALL}*-secure.*

Assume per absurdum that $Sec_{\delta_{\mathrm{ALL}}}(P_\star)$ *holds; that is,* $\langle P_\star,\ \triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_\star,\ \triangle\rangle$. *We have*

$$\begin{array}{l}\delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \triangle\rangle \xrightarrow{\mathrm{env}, c_{LL}?d} \langle c_{LH}!d_1 \mid c_{LH}?x_1[y_1].c'_{LH}!y_1,\ c_{LL}!d_\triangle\rangle \\ \delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \triangle\rangle \xrightarrow{\mathrm{env}} \langle P_\star,\ \triangle\Box\rangle \\ c!d_\triangle\ W_{\mathcal{E}}\ \triangle\Box\end{array}$$

and the only possible simulation is[2]

$$\delta_{\mathrm{ALL}} \vdash \langle P_\star,\ {}_\triangle\Box\rangle \xrightarrow{\Box} \langle P_\star,\ \Box_\triangle\rangle,$$

resulting in $\langle c_{LH}!d_1 \mid c_{LH}?x_1[y_1].c'_{LH}!y_1,\ c_{LL}!d_\triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_\star,\ \Box_\triangle\rangle$.

We then have

$$\begin{array}{l}\delta_{\mathrm{ALL}} \vdash \langle c_{LH}!d_1 \mid c_{LH}?x_1[y_1].c'_{LH}!y_1,\ c_{LL}!d_\triangle\rangle \xrightarrow{\mathrm{env},\tau} \langle c'_{LH}!d_1,\ c_{LL}!d_\triangle\Box\rangle\\ \delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \Box_\triangle\rangle \xrightarrow{\mathrm{env}} \langle P_\star,\ \Box_\triangle\Box\rangle\\ c_{LL}!d_\triangle\Box\ W_{\mathcal{E}}\ \Box_\triangle\Box\end{array}$$

and the only possible simulation is ($\epsilon \in \tau^*$)

$$\delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \Box_\triangle\Box\rangle \xrightarrow{\epsilon} \langle P_\star,\ \Box_\triangle\Box\rangle,$$

resulting in $\langle c'_{LH}!d_1,\ c_{LL}!d_\triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_\star,\ \Box_\triangle\rangle$.

We also have

$$\begin{array}{l}\delta_{\mathrm{ALL}} \vdash \langle c'_{LH}!d_1,\ c_{LL}!d_\triangle\Box\rangle \xrightarrow{\mathrm{env},\Box} \langle c'_{LH}!d_1,\ c_{LL}!d.\Box_\triangle\rangle\\ \delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \Box_\triangle\rangle \xrightarrow{\mathrm{env}} \langle P_\star,\ \Box_\triangle c_{LH}!d_2\rangle\\ c_{LL}!d.\Box_\triangle\ W_{\mathcal{E}}\ \Box_\triangle c_{LH}!d_2\end{array}$$

and the only possible simulation is

$$\delta_{\mathrm{ALL}} \vdash \langle P_\star,\ \Box_\triangle c_{LH}!d_2\rangle \xrightarrow{c_{LH}?d_2} \langle c_{LL}?x.c_{LH}!d_1 \mid c'_{LH}!d_2,\ \Box.c_{LH}!d_{2\triangle}\rangle,$$

resulting in

$$\langle c'_{LH}!d_1,\ c_{LL}!d.\Box_\triangle\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle c_{LL}?x.c_{LH}!d_1 \mid c'_{LH}!d_2,\ \Box.c_{LH}!d_{2\triangle}\rangle.$$

We now have

$$\begin{array}{l}\delta_{\mathrm{ALL}} \vdash \langle c'_{LH}!d_1,\ c_{LL}!d.\Box_\triangle\rangle \xrightarrow{\mathrm{env},c'_{LH}!d_1} \langle 0,\ c_{LL}!d.\Box.c'_{LH}?d_{1\triangle}\rangle\\ \delta_{\mathrm{ALL}} \vdash \langle c_{LL}?x.c_{LH}!d_1 \mid c'_{LH}!d_2,\ \Box.c_{LH}!d_{2\triangle}\rangle \xrightarrow{\mathrm{env}}\\ \qquad\qquad \langle c_{LL}?x.c_{LH}!d_1 \mid c'_{LH}!d_2,\ \Box.c_{LH}!d_{2\triangle}c'_{LH}?[\,]\rangle\\ c_{LL}!d.\Box.c'_{LH}?d_{1\triangle}\ W_{\mathcal{E}}\ \Box.c_{LH}!d_{2\triangle}c'_{LH}?[\,]\end{array}$$

The only "simulation" allowed by the semantics leads to the observational history $\Box.c_{LH}!d_2.c'_{LH}?d_{2\triangle}$. *But* $c_{LL}!d.\Box.c'_{LH}?d_{1\triangle}$ *and* $\Box.c_{LH}!d_2.c'_{LH}?d_{2\triangle}$ *are not related by* $W_{\mathcal{E}}$. *This contradiction disproves the* δ_{ALL}*-security of* $P_\star$. □

A consequence of Theorem 4.18 is: if no channel having low presence integrity and high content integrity is reachable, then δ_{ALL}-security is fully parallel-compositional.

[2]Strictly speaking all $P'_\star$ such that $P'_\star \equiv P_\star$ can be resulted. Since $P'_\star$ allows exactly the same behaviors as $P_\star$ does, we avoid this verbosity without sacrificing the validity of our argumentation.

$$
\begin{array}{lll}
\mathit{SINK} & \triangleq & \&_2(c_{LL}?x_1, c_{HH}?x_2[y_2]).\\
& & (\nu c_{\mathrm{f}})(c_{\mathrm{i}}!(y_2, c_{\mathrm{f}}).c_{\mathrm{f}}?x_{\mathrm{f}}.\\
& & \quad \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) :\\
& & \quad\quad (\nu c_{\mathrm{e}}, c_{\mathrm{r}})(c_{\mathrm{p}}!(c_{\mathrm{e}}, c_{\mathrm{r}}).c_{\mathrm{r}}?x_3[y_3].\\
& & \quad\quad\quad \&_2(c_{LH}!y_3\{x'_3\}, c'_{HH}?x_{\mathrm{t}}).\\
& & \quad\quad\quad \mathsf{case}\ x'_3\ \mathsf{of}\ \mathsf{some}(y'_3) :\\
& & \quad\quad\quad\quad (\nu c'_{\mathrm{e}}, c'_{\mathrm{r}})(c_{\mathrm{d}}!(c'_{\mathrm{e}}, c'_{\mathrm{r}}).c'_{\mathrm{r}}?x_4[y_4].\mathit{SINK})\\
& & \quad\quad\quad \mathsf{else}\ \mathit{SINK})\\
& & \quad \mathsf{else}\ c'_{HH}?x_{\mathrm{t}}.\mathit{SINK})
\end{array}
$$

Figure 4.6: The "Realization" of Sink Channels with Low Presence Integrity and High Content Integrity

Corollary 4.19 *Suppose* $\forall c, \rho : (c, \rho) \in \mathsf{rch}(P_1) \cup \mathsf{rch}(P_2) \Rightarrow \mathcal{E}^{\circ}(c) \sqsupseteq \mathcal{E}^{\bullet}(c)$. *Then* $\mathit{Sec}_{\delta_{\mathrm{ALL}}}(P_1|P_2)$ *can be deduced from* $\mathit{Sec}_{\delta_{\mathrm{ALL}}}(P_1)$ *and* $\mathit{Sec}_{\delta_{\mathrm{ALL}}}(P_2)$.

In summary, the results presented above help elucidate the points below.

1. If δ-security had been fully compositional, it would not have uncovered certain insecure dependencies of high integrity content on low integrity presence.
2. The notion of δ-security is fully compositional for processes that do not make use of LH-channels.

Remark 4.2 *Trace-based properties are more often non-compositional in comparison to bisimulation-based properties [MS03]. It is not surprising that the compositionality of* δ_{ALL}*-security is conditional, since the histories of the observation of the environment are considered, and "out-of-sync" correspondence within the two histories may be needed when channels with* LH*-integrity are involved.*

4.5 Some Examples, and Discussion

On LH-Channels

We illustrate that LH-channels can be induced from channels that are LL and HH by a concrete process. The procedure *SINK* in Fig. 4.6 mimics the potential congestion of the high integrity data source c_{HH} using a queue: output of the oldest element suspended in the queue is attempted through the sink channel c_{LH} only when the low integrity switch c_{LL} is on. Recall that the $\&_2(\_, \_)$ can be passed if and only if the second communication is successful.

The channels c_{i}, c_{d}, and c_{p} are interfaces for the operations "insert" ("enque"), "delete" ("deque"), and "peek" (the non-destructive inspection of the oldest

$$QUE(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}) \triangleq (\nu c_{\mathrm{g}})(E(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, \mathsf{some}(c_{\mathrm{g}})) \mid G(\mathsf{some}(c_{\mathrm{g}})))$$

$$G(x_{\mathrm{g}}[c_{\mathrm{g}}]) \triangleq c_{\mathrm{g}}?(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}).E(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{g}}) \mid G(x_{\mathrm{g}})$$

$$\begin{array}{l} E(x_{\mathrm{i}}[c_{\mathrm{i}}], x_{\mathrm{d}}[c_{\mathrm{d}}], x_{\mathrm{p}}[c_{\mathrm{p}}], x_{\mathrm{g}}[c_{\mathrm{g}}]) \triangleq \\ \quad c_{\mathrm{i}}?(x, x_{\mathrm{f}})[\_, c_{\mathrm{f}}]. \\ \quad\quad (\nu c'_{\mathrm{i}}, c'_{\mathrm{d}}, c'_{\mathrm{p}})(c_{\mathrm{g}}!(c'_{\mathrm{i}}, c'_{\mathrm{d}}, c'_{\mathrm{p}}).c_{\mathrm{f}}!\checkmark.F(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x, \mathsf{some}(c'_{\mathrm{i}}), \mathsf{some}(c'_{\mathrm{d}}), \mathsf{some}(c'_{\mathrm{p}}), x_{\mathrm{g}})) \\ \quad + c_{\mathrm{d}}?(x_{\mathrm{e}}, x_{\mathrm{r}})[c_{\mathrm{e}}, \_].c_{\mathrm{e}}!\checkmark.E(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{g}}) \\ \quad + c_{\mathrm{p}}?(x'_{\mathrm{e}}, x'_{\mathrm{r}})[c'_{\mathrm{e}}, \_].c'_{\mathrm{e}}!\checkmark.E(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{g}}) \end{array}$$

$$\begin{array}{l} F(x_{\mathrm{i}}[c_{\mathrm{i}}], x_{\mathrm{d}}[c_{\mathrm{d}}], x_{\mathrm{p}}[c_{\mathrm{p}}], x_{\mathrm{k}}[c_{\mathrm{k}}], x'_{\mathrm{i}}[c'_{\mathrm{i}}], x'_{\mathrm{d}}[c'_{\mathrm{d}}], x'_{\mathrm{p}}[c'_{\mathrm{p}}], x_{\mathrm{g}}) \triangleq \\ \quad c_{\mathrm{i}}?(x, x_{\mathrm{f}})[y, c_{\mathrm{f}}]. \\ \quad\quad (\nu c'_{\mathrm{f}})(c'_{\mathrm{i}}!(y, c'_{\mathrm{f}}) \mid c'_{\mathrm{f}}?x'.c_{\mathrm{f}}!\checkmark.F(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{k}}, x'_{\mathrm{i}}, x'_{\mathrm{d}}, x'_{\mathrm{p}}, x_{\mathrm{g}})) + \\ \quad c_{\mathrm{d}}?(x_{\mathrm{e}}, x_{\mathrm{r}})[\_, c_{\mathrm{r}}]. \\ \quad\quad (\nu c'_{\mathrm{e}}, c'_{\mathrm{r}})(c'_{\mathrm{d}}!(c'_{\mathrm{e}}, c'_{\mathrm{r}}) \mid \\ \quad\quad (c'_{\mathrm{e}}?x''.c_{\mathrm{r}}!c_{\mathrm{k}}.E(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{g}}) + c'_{\mathrm{r}}?x'''.c_{\mathrm{r}}!c_{\mathrm{k}}.F(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x''', x'_{\mathrm{i}}, x'_{\mathrm{d}}, x'_{\mathrm{p}}, x_{\mathrm{g}}))) + \\ \quad c_{\mathrm{p}}?(x''_{\mathrm{e}}, x''_{\mathrm{r}})[\_, c''_{\mathrm{r}}].c''_{\mathrm{r}}!c_{\mathrm{k}}.F(x_{\mathrm{i}}, x_{\mathrm{d}}, x_{\mathrm{p}}, x_{\mathrm{k}}, x'_{\mathrm{i}}, x'_{\mathrm{d}}, x'_{\mathrm{p}}, x_{\mathrm{g}}) \end{array}$$

Figure 4.7: Specification of FIFO Queue

element) of the queue specified by the procedure *QUE* in Figure 4.7. The procedure *SINK* waits on the input over c_{HH} for the composite binder on the first line to be passed. When that happens, the input data over c_{HH} is enqued, with the completion of the "enque" operation signaled on c_{f}. If the input over c_{LL} was also successful, then outputting the head of the queue is attempted, with a high integrity timeout signal that is supposed to arrive over c'_{HH}. If the output is successful before the timeout, then the data item of the output is deleted from the queue. In the "peek" and "deque" operations, the channels c_{e} and c'_{e} are sent to the queue for the latter to signal back whether it is already an empty queue when the operations are invoked. In our case the non-emptiness of the queue is an invariant and hence neither c_{e} nor c'_{e} is subsequently used. The process $(\nu c_{\mathrm{i}}, c_{\mathrm{d}}, c_{\mathrm{p}})(SINK \mid QUE(\mathsf{some}(c_{\mathrm{i}}), \mathsf{some}(c_{\mathrm{d}}), \mathsf{some}(c_{\mathrm{p}})))$ is δ_{ALL}-secure but we elide the detailed arguments.

Remark 4.3 *Disregarding recursion, the construction becomes as succinct as the following process:*

$$\&_2(c_{LL}?x_1, c_{HH}?x_2[y_2]).\ \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : c_{LH}!y_2\ \mathsf{else}\ 0$$

The output over c_{LH} may never happen. But given its presence the content integrity of the output content is high.

Further Information on the FIFO Queue The implementation detail of the queue is irrelevant for our example in Figure 4.6 to work. In fact, all

the communications with the queue are τ-actions produced by a derivative of $(\nu c_{\rm i}, c_{\rm d}, c_{\rm p})(SINK \mid QUE(\mathsf{some}(c_{\rm i}), \mathsf{some}(c_{\rm d}), \mathsf{some}(c_{\rm p})))$. However, we provide an example implementation in Figure 4.7, to demonstrate that such a FIFO queue is realizable in our calculus. This FIFO queue is adapted from the priority queue discussed in [SW01]. A peek operation that returns but does not remove the head of the queue is added. A call to the procedure E represents the tail of the queue, while each call to F as a parallel component stands for an ordinary cell, where x_k contains the content stored in that cell. For the other parameters of F, $x_{\rm i}$, $x_{\rm d}$ and $x_{\rm p}$ correspond to the interfaces of the current cell, and $x'_{\rm i}$, $x'_{\rm d}$ and $x'_{\rm p}$ correspond to the interfaces of the oldest of all cells newer than the current one, or of the tail (a call to E) if no newer cell exists. For $QUE(\mathsf{some}(c_{\rm i}), \mathsf{some}(c_{\rm d}), \mathsf{some}(c_{\rm p}))$, $c_{\rm i}$, $c_{\rm d}$ and $c_{\rm p}$ are invariably the interfaces to the oldest cell, or to the tail if the queue is empty. Finally, the procedure G is a replicator that produces new tails after insertion operations.

Secure Composition

We now consider making the multiplexer process (process 6 in Fig. 4.1) source from the channel c_{LH} in Figure 4.6.

Let

$$SRC \triangleq (\nu c_{\rm i}, c_{\rm d}, c_{\rm p})(SINK \mid QUE(\mathsf{some}(c_{\rm i}), \mathsf{some}(c_{\rm d}), \mathsf{some}(c_{\rm p}))).$$

The process under consideration is $(\nu c_{LH})(SRC|M)$. It is not difficult to see that $\mathsf{det}(SRC, c_{LH})$ and $\mathsf{det}(M, c_{LH})$ hold. Hence we can deduce the validity of $Sec_{\delta_{\rm ALL}}((\nu c_{LH})(SRC|M))$ by Theorem 4.18 and the $\delta_{\rm ALL}$-security of SRC and M, as long as $c_{LH} \notin \mathbf{Dt}$.

Message Authentication Codes

We provide a solid example on how the use of Message Authentication Codes gives rise to LH-channels, which was briefly mentioned in the beginning of this chapter.

The process $MAC(\mathsf{some}(k_*))$ in Figure 4.8 verifies the MAC of the input before it proceeds with outputting it on channel c_{LH}. Suppose there is a data stream $\vec{c}_* = c_0, c_1, ...,$ and the strategy $\delta_{\rm MAC}$ is such that

$$\begin{array}{l} \forall \pi \textit{ s.t. } \pi \textit{ contains } i \textit{ inputs over } c'_{LH}\ (i \geq 0): \\ \quad \delta_{\rm MAC}(\pi) \quad = \quad \{c_{LL}!(c_i, f(k_*, c_i))\} \cup \\ \qquad\qquad\quad\ \ \{c_{LL}!(c_{\rm att}, c'_{\rm att}) \mid c_{\rm att}, c'_{\rm att} \in \mathbf{Dt} \ \wedge\ c'_{\rm att} \neq f(k_*, c_{\rm att})\} \cup \\ \qquad\qquad\quad\ \ \{c_{LH}?[\,], c'_{LH}?[\,], \Box\}. \end{array}$$

The constraint $c'_{\rm att} \neq f(k_*, c_{\rm att})$ expresses that it is impossible[3] for the attacker to provide the appropriate $c_{\rm att}$ and $c'_{\rm att}$ that can pass the MAC check, since

[3]This is an approximative argument without using probabilities.

it does not know k_*. The process $MAC(\mathsf{some}(k_*))$ is δ_{MAC}-secure: each time an output is performed on c_{LH}, the output content must be the next item on the data stream $\vec{c}_*$. The LH-classification of the sink channel c_{LH} displays the improvement in the integrity of the content delivered to the end user due to the use of Message Authentication Codes, despite the fact that this content can still be delayed by the attacker.

$$
\begin{array}{rcl}
MAC(K) & \triangleq & c_{LL}?(x_1,x_2)[y_1,y_2].\\
& & \mathsf{case}\ K\ \mathsf{of}\ \mathsf{some}(k):\\
& & \quad \mathsf{if}\ y_2 = f(k,y_1)\ \mathsf{then}\\
& & \quad\quad c_{LH}!y_1.c'_{LH}!\checkmark.MAC(K)\\
& & \quad \mathsf{else}\ MAC(K)\\
& & \mathsf{else}\ 0
\end{array}
$$

Figure 4.8: Message Authentication Codes Boost Content Integrity

Although it is not the main purpose of this example, the procedure MAC also illustrates a form of *endorsement* (upgrade of integrity level by additional trust, e.g., [SS09]) in the content dimension, that is supported by the use of cryptography.

4.5.1 On Confidentiality

We are in a position to further explain having developed our theory for integrity, rather than confidentiality. It has been illustrated by the example in Figure 4.6 that a *concrete process* can influence the presence of communication over a sink channel, without influencing the communication content. This is depicted by the sub-figure on the left of Figure 4.9. For confidentiality, a channel c_{SP} with secret presence and public content would correspond to our channel with LH-integrity. Assuming the existence of c_{SP} and developing the same theory (our δ-bisimulation and its related properties) would not be problematic. However, it is difficult to come up with a possibilistic process that leaks the content of c_{SP} properly, without leaking the presence of communication over it, unless other channels also with confidential presence and public content are used. This can be informally explained as follows. The proper leakage of the input content over c_{SP} can only be achieved via c_{PP} rather than c_{SS}. However, the inspection of this input content leads to a context with confidential presence, in which c_{PP} cannot be placed. Going back to Figure 4.9, the situation on the right might not exist for possibilistic processes. Hence the meaning of "confidential presence, public content" would be harder to justify as opposed to "low integrity presence, high integrity content".

Moving to a *probabilistic* setting, a process corresponding to the right sub-figure of Figure 4.9 can be formulated much more easily. An example is shown

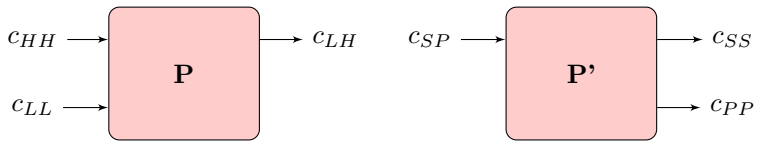


Figure 4.9: Sink Channel c_{LH} versus Source Channel c_{SP}

below.

$$
\begin{array}{l}
\&_2(c_{SP}\,?\,x_1, c'_{PP}\,?\,x_2).\\
\mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1):\\
\quad c_{PP}\,!\,y_1\\
\mathsf{else}\\
\quad c_{PP}\,!\,rand()
\end{array}
$$

The function $rand()$ above generates a random number. Suppose the random number generated by $rand()$ has the same distribution as that of the input content over c_{SP}. Then the secrete presence of the input over c_{SP} will be properly concealed by the public input c'_{PP}, whereas the content over c_{SP} will be properly revealed through the public output c_{PP}.

Remark 4.4 *In a possibilistic setting, we have given intuitive reasons that the confidentiality counterpart of LH-integrity labels may not exist for possibilistic processes and environments. No formal arguments have been developed. This is not a serious problem since the results of this chapter would be directly applicable to confidentiality if such labels should exist.*

4.6 Related Work

In a number of existing developments (e.g., [OCC06, CH08, RHS12, MC12]), strategies are used to express noninterference-style secuirty properties. This goes back to Wittbold and Johnson's "nondeducibility on strategies" [WJ90], a re-interpretation of Sutherland's "nondeducibility" [Sut86]. It was remarked in [WJ90] that strategies make possible a "hook-up" (compositionality) result.

In our case, the use of strategies allows for a clear distinction between the environment attempting to introduce perturbation ($\forall \pi_{20}\ s.t.\ \delta \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\text{env}} \langle P_2,\ \pi_{20}\rangle$) on the presence/content of communications over channels with low integrity, and the process striving to stay unaffected in terms of the presence/-content of behaviors with high integrity, under such perturbation ($\exists \alpha' : \delta \vdash \langle P_2, \pi_{20}\rangle \overset{\widehat{\alpha'}}{\underset{\cdots}{\Longrightarrow}} \ldots$). In addition, as is the case for [WJ90], the power of the strategy δ_{ALL} is also a key enabler of the compositionality of δ_{ALL}-security.

All the aforementioned works discuss trace-based noninterference properties, and [RHS12], in particular, establishes the connection of proposed security properties with existing trace-based properties for interactive programs. On the other hand, we use environment strategies together with a bisimulation-based property, whose connection with the bisimulation-based process-algebraic property SBNDC is established.

The distinction between the presence and content of communication is considered in several existing developments on confidentiality [SM02, RHS12, RS14]. Thus the case where presence is more sensitive than content is not dealt with in the security properties proposed. Sabelfeld and Mantel [SM02] gave a bisimulation-based, timing-sensitive notion of security, without employing environment strategies. Rafnsson and Sabelfeld [RS14] studied bisimulation-based security properties for interactive programs, modeling strategies as part of systems, and elaborating on the compositional proof of security.

Chapter 5

Content-Dependent Flow Policies

In this chapter, we focus on the *content* of communication and storage, and develop an information flow analysis dealing with security policies that are dependent on such content. It is important what information flow security amounts to in this setting; hence a good part of this chapter will be concerned with the discussion of content-dependent noninterference, which is enforced by our type-based analysis.

In the classical setting, a channel/variable is globally always confidential or always public. In flow-sensitive security [HS06], a variable can have different security levels at different program points (essentially the security level depends on the content of the program counter). With the use of our content-dependent security policies, a channel/variable can have different security levels depending on its content. That is to say, the association between content and security level is made explicit in the security types.

Content-dependent information flow is not new; however, the bulk of existing work [BS06, BS10, ADZ+12, LC13, LC15] in this direction deals with sequential languages. As we will demonstrate shortly in Section 5.1, enforcing content-dependent policies in a concurrent language with communicating processes is practically meaningful. However, this is not a trivial task: for instance, the security property needs to be concerned with not only the memory, but also communication actions, and the contents of both the memory and the communications should be involved in the selection of policies at different points of execution.

In this chapter, we will focus on interpreting and enforcing content-dependent policies and correspondingly certain features potentially inducing complexity

will be avoided in the development. First of all, we will work in a less expressive, simple concurrent programming language, compared with the Quality Calculus used in the previous chapters. As previously mentioned, simple programming languages have also been the setting for the study of other existing theories [BS06, BS10, ADZ+12, LC13, LC15] on content-dependency. Technically, the distinguishing feature of our language is the admittance of *concurrent processes* that *communicate synchronously* with each other. Concerning communication, the distinction between "presence" and "content" is still a key element. However, we dispense with tackling the case where "presence" is more sensitive than content, and accordingly perform our development for *confidentiality* (as opposed to integrity). This simplification also enables us to shift back to a process-centered (as opposed to an environment-aware) style in the formulation of our semantics and security properties.

In Section 5.1, we give a motivating example for the use of content-dependent flow policies in concurrent, communicating systems. In Section 5.2, we present the syntax and semantics of our language. In Section 5.3, we formalize our policies. In Section 5.4, we introduce the reader to a bisimulation-based, progress-sensitive noninterference property that underpins content-dependent information flow security, and show that this property is compositional. In Section 5.5, we present our information flow type system, and prove its soundness with respect to the noninterference property just introduced. The use of a specific scheduler can incur "refinement attacks", just like in the content-independent scenario; hence we extend our work to deal with deterministic schedulers in Section 5.7.

A proof-of-concept type checker is implemented[1] in the OCaml language [OL]. Since our type system is concerned with assertions about the contents of memory, logical formulas that correspond to the negation of these assertions are emitted. Their unsatisfiability (which can be directly obtained from the SMT solver Z3 [DMB08]) implies well-typedness.

5.1 Motivating Example

In this chapter, we will consider a different multiplexer example than that of Process 6 in Section 4.1. This example involves two partitions — hosting a multiplexer process and a demultiplexer process, respectively — that communicate with each other over a shared channel.

In Figure 5.1, the multiplexer process (S_{M}) wraps up the source data in x_1 or x_2, along with tags 1 and 2 respectively, and forwards it over the dyadic channel c to the demultiplexer process (S_{D}). The demultiplexer will then unwrap the

[1] The implementation is accessible at
http://orbit.dtu.dk/files/123041336/phd388_Li_X_cpchecker.zip

data and forward it to the sinks z_1 or z_2, depending on the tag value.

```
Multiplexer (S_M) :   while tt do
                        c!(1, x_1);
                        c!(2, x_2)

Demultiplexer (S_D) : while tt do
                        c?(y, z);
                        if y = 1
                          then z_1 := z
                          else z_2 := z
```

Figure 5.1: The Code for the Multiplexer and the Demultiplexer

Suppose the variable x_1 is confidential, whereas x_2 is public. The information flow analysis should then reveal that z_1 needs to be confidential, while z_2 can be public.

For modularity reasons, a *type-based analysis* needs to assign a confidentiality level to the channel c for S_{M} and S_{D} to be analyzed separately. In the demultiplexer process S_{D}, both z_1 and z_2 obtain data from c, depending on whether the tag is 1 or not. It would then be desirable for the type system to have the knowledge that *either* c is confidential and communicating $(1, \_)$, *or* c is public and communicating $(2, \_)$. This is precisely what our disjunctive policies aim to capture, in the setting of concurrent systems.

Moreover, when it is said that "c is confidential", what is actually meant is that it communicates confidential *content*. The observation of the mere *presence* of any communication action over c, without observing the communicated content, does not jeopardize the confidentiality of x_1. We thus distinguish between the presence and content of channel communication (e.g., [SM02, RS14, LNN15]) as in Chapter 4, for a more fine-grained, permissive enforcement of our disjunctive policies. In a confidentiality setting, we follow the usually assumed constraint that presence can be no more confidential than content.

The modern IT systems onboard airplanes are divided into domains that provide different functionalities ranging from aircraft control to passenger entertainment. Domains are further sub-divided into partitions that can each host a different runtime, which is enabled by virtualization techniques. Each communication path between two partitions belonging to different domains is required to go through a common "security gateway" that sanitizes data traffic [MPTB12]. The shared channel c of Figure 5.1 essentially models a gateway that directs data from x_1 in one source partition to z_1 in one destination partition, and directs data from x_2 in another source partition to z_2 in another destination partition. Correspondingly, the tags 1 and 2 represent the addresses of the destinations, which are needed for the gateway to properly maintain the different data paths.

Table 5.2: The Simple Concurrent Language with Communication

a	::=	$n \mid x \mid a_1\ op\ a_2$
b	::=	$\mathbf{tt} \mid a_1\ rel\ a_2$
S	::=	$\mathsf{skip} \mid \mathsf{nil} \mid x := a \mid S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ b\ \mathsf{do}\ S \mid c?\vec{x} \mid c!\vec{a}$
Σ	::=	$i : S_i \mid \Sigma_1 \| \Sigma_2 \mid \Sigma \setminus \Omega$

5.2 A Simple Concurrent Language

In this section, we introduce the syntax and semantics of a simple concurrent language that we will use throughout our development of this chapter.

5.2.1 Syntax

A system Σ consists of a fixed number of concurrent processes. All variables are local to their own processes, and information sharing is achieved by means of communication.

All processes are assumed to have distinct identifiers in $\{1, 2, ...\}$. For a system Σ with the set $Pid(\Sigma)$ of process identifiers, its set of variables can thus be denoted by $\mathbf{Var}_\Sigma = \biguplus_{i \in Pid(\Sigma)} \mathbf{Var}_i$, where the process with identifier i can only use variables from $\mathbf{Var}_i$. For communication, the set of polyadic channels is **PCh** and the set of component channels is

$$\mathbf{Ch} = \{c.1, \cdots, c.m \mid c \in \mathbf{PCh}, \text{ and } c \text{ has arity } m\}.$$

The use of polyadic channels supports the grouping of multiple fields of a message, such that each field is communicated via a different component channel belonging to the same polyadic channel.

The syntax of our language is given in Table 5.2. We write x, y, z for variables, X for sets of variables, c for either a polyadic channel name or a component channel name (it will always be clear from the context which is the case), n for unspecified constants, *op* for unspecified arithmetic operators, *rel* for unspecified relational operators, and **tt** for the boolean constant denoting truth. The set of variables contained in an arithmetic expression a (resp. boolean expression b) is $fv(a)$ (resp. $fv(b)$). When it is unambiguous from the context, we may also use the term "channel" to refer to polyadic channels.

A sequential process S can be an empty operation skip, a terminated process nil, an assignment $x := a$, a sequential composition $S_1; S_2$, a conditional branch if b then S_1 else S_2, a loop while b do S, or a communication binder. In

particular, the binder $c!\vec{a}$ outputs the vector $\vec{a}$ of arithmetic expressions over the polyadic channel c, and the binder $c?\vec{x}$ inputs from the polyadic channel c into the vector $\vec{x}$ of variables. Note that the process nil is not in the surface syntax and *cannot* be used directly by the programmer. It serves as a hint in the semantic rule for sequential composition $S_1; S_2$ that the execution of S_1 has terminated and that of S_2 can begin, which helps ensure the uniformity of the semantic configurations. This will become clear in the next subsection.

Systems Σ are composed of concurrent, communicating processes. The construct $i : S_i$ represents a process running the statement S_i, with process identifier i. The construct $\Sigma_1||\Sigma_2$ represents two systems running concurrently. We require $Pid(\Sigma_1) \cap Pid(\Sigma_2) = \emptyset$ for the well-formedness of $\Sigma_1||\Sigma_2$. Finally the construct $\Sigma \setminus \Omega$ is the system that can perform all the input/output actions of Σ provided that those actions are not over the polyadic channels in Ω. This last construct is similar to the restriction operator [Mil89] of CCS, whose introduction allows one to specify whether each channel used by a process is shared with another process or with the environment.

5.2.2 Semantics

The structural operational semantics of our language is presented in Table 5.3. The transitions are annotated with the actions being performed, including communications. At the current stage, an action α takes one of three forms: $c!\vec{v}$ (for output over c), $c?\vec{v}$ (for input over c) or τ (for the remaining cases) where $\vec{v} \in \mathbf{Val}^\star$ denotes the value vector being communicated over the polyadic channel. We tacitly assume that arities match without having explicitly to require this in the semantics for output and input.

The general form of the transitions for processes is $\vdash_i \langle S; \sigma\rangle \xrightarrow{\alpha} \langle S'; \sigma'\rangle$, where i is the identifier of the process being executed, and $\sigma, \sigma' \in \mathbf{Var}_i \to \mathbf{Val}$. We make use of two semantic functions $\mathcal{A}$ and $\mathcal{B}$ that give the results of the evaluation of the expressions as their first arguments in the memories as their second arguments. The transition rules are fairly standard. Note that we no longer make explicit use of an environment strategy as in Chapter 4, and the rule for input suggests that any value vector can potentially be received from the environment.

Lifting the semantics to systems, the configurations take the form $C = \langle \Sigma; \sigma\rangle$, where we tacitly assume that $\sigma \in \mathbf{St}_\Sigma$, and $\mathbf{St}_\Sigma$ is $\mathbf{Var}_\Sigma \to \mathbf{Val}$. The transitions are of the form $\langle \Sigma; \sigma\rangle \xrightarrow{\alpha}_\eta \langle \Sigma'; \sigma'\rangle$ where η is a non-empty list of identifiers for the processes executed. For a mapping A, we write $\mathbf{D}_A$ for its domain. For two mappings A and B such that $\mathbf{D}_A \cap \mathbf{D}_B = \emptyset$, we denote by $A \uplus B$ the mapping with domain $\mathbf{D}_A \uplus \mathbf{D}_B$, such that

$$(A \uplus B)(i) = \begin{cases} A(i) & (\text{if } i \in \mathbf{D}_A) \\ B(i) & (\text{if } i \in \mathbf{D}_B) \end{cases}.$$

Table 5.3: Small-step Semantics of Processes and Systems.

$$\vdash_i \langle \mathsf{skip}; \sigma\rangle \xrightarrow{\tau} \langle \mathsf{nil}; \sigma\rangle \qquad \vdash_i \langle x := a; \sigma\rangle \xrightarrow{\tau} \langle \mathsf{nil}; \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]\rangle$$

$$\vdash_i \langle c!\vec{a}; \sigma\rangle \xrightarrow{c!\vec{v}} \langle \mathsf{nil}; \sigma\rangle \text{ if } \vec{v} = \mathcal{A}[\![\vec{a}]\!]\sigma \qquad \vdash_i \langle c?\vec{x}; \sigma\rangle \xrightarrow{c?\vec{v}} \langle \mathsf{nil}; \sigma[\vec{x} \mapsto \vec{v}]\rangle$$

$$\frac{\vdash_i \langle S_1; \sigma\rangle \xrightarrow{\alpha} \langle S_1'; \sigma'\rangle}{\vdash_i \langle S_1; S_2; \sigma\rangle \xrightarrow{\alpha} \langle S_1'; S_2; \sigma'\rangle} \text{ if } S_1' \neq \mathsf{nil} \qquad \frac{\vdash_i \langle S_1; \sigma\rangle \xrightarrow{\alpha} \langle \mathsf{nil}; \sigma'\rangle}{\vdash_i \langle S_1; S_2; \sigma\rangle \xrightarrow{\alpha} \langle S_2; \sigma'\rangle}$$

$$\vdash_i \langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; \sigma\rangle \xrightarrow{\tau} \langle S_1; \sigma\rangle \text{ if } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$$

$$\vdash_i \langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; \sigma\rangle \xrightarrow{\tau} \langle S_2; \sigma\rangle \text{ if } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$$

$$\vdash_i \langle \mathsf{while}\ b\ \mathsf{do}\ S; \sigma\rangle \xrightarrow{\tau} \langle (S; \mathsf{while}\ b\ \mathsf{do}\ S); \sigma\rangle \text{ if } \mathcal{B}[\![b]\!]\sigma = \mathbf{tt}$$

$$\vdash_i \langle \mathsf{while}\ b\ \mathsf{do}\ S; \sigma\rangle \xrightarrow{\tau} \langle \mathsf{nil}; \sigma\rangle \text{ if } \mathcal{B}[\![b]\!]\sigma = \mathbf{ff}$$

$$\frac{\vdash_i \langle S_i; \sigma\rangle \xrightarrow{\alpha} \langle S_i'; \sigma'\rangle}{\langle i : S_i; \sigma\rangle \xrightarrow{\alpha}_i \langle i : S_i'; \sigma'\rangle} \qquad \frac{\langle \Sigma; \sigma\rangle \xrightarrow{\alpha}_\eta \langle \Sigma'; \sigma'\rangle}{\langle \Sigma \setminus \Omega; \sigma\rangle \xrightarrow{\alpha}_\eta \langle \Sigma' \setminus \Omega; \sigma'\rangle} \text{ if } ch(\alpha) \notin \Omega$$

$$\frac{\langle \Sigma_1; \sigma_1\rangle \xrightarrow{c\rho\vec{v}}_i \langle \Sigma_1'; \sigma_1'\rangle \quad \langle \Sigma_2; \sigma_2\rangle \xrightarrow{c\bar{\rho}\vec{v}}_j \langle \Sigma_2'; \sigma_2'\rangle}{\langle \Sigma_1 || \Sigma_2; \sigma_1 \uplus \sigma_2\rangle \xrightarrow{\tau}_{i,j} \langle \Sigma_1' || \Sigma_2'; \sigma_1' \uplus \sigma_2'\rangle} \text{ where } \rho \in \{!, ?\}$$

$$\frac{\langle \Sigma_1; \sigma_1\rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1'\rangle}{\langle \Sigma_1 || \Sigma_2; \sigma_1 \uplus \sigma_2\rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1' || \Sigma_2; \sigma_1' \uplus \sigma_2\rangle} \qquad \frac{\langle \Sigma_2; \sigma_2\rangle \xrightarrow{\alpha}_\eta \langle \Sigma_2'; \sigma_2'\rangle}{\langle \Sigma_1 || \Sigma_2; \sigma_1 \uplus \sigma_2\rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1 || \Sigma_2'; \sigma_1 \uplus \sigma_2'\rangle}$$

Then the transition rules for systems are mostly self-explanatory. In particular, the second rule says that $\langle \Sigma \setminus \Omega;\ \sigma\rangle$ can perform an action α of $\langle \Sigma; \sigma\rangle$ if the channel used by α is not in Ω.

Example 5.1 *The combination of the multiplexer and demultiplexer considered in Section 5.1 can be represented by the system* $\Sigma_{\mathrm{MD}} = (1 : S_{\mathrm{M}} \,||\, 2 : S_{\mathrm{D}}) \setminus \{c\}$. □

5.3 Content-Dependent Flow Policies

As demonstrated by our motivating example, the meaningfulness of content-dependent flow policies is twofold:

1. They add to the permissiveness of the modular enforcement of information flow policies such as using type systems. And the enforcement is still justifiable by a properly formulated noninterference property.

2. The "information paths" inside a system are displayed in more detail, rendering code more easily understandable and less error-prone.

For systems Σ, we introduce policy environments $\mathcal{P}$ such that for each $i \in Pid(\Sigma)$, $\mathcal{P}(i)$ is a set of policies for the variables in $\mathbf{Var}_i$. Each variable policy $\mathfrak{P} \in \mathcal{P}(i)$ consists of two components[2], $\mathfrak{P}_\mathsf{S} : \mathbf{Var}_i \to \{H, L\}$ and $\mathfrak{P}_\mathsf{V} : \mathbf{Lab}_\mathbf{F}$, where $\mathfrak{P}_\mathsf{S}$ contains the confidentiality level of each variable in $\mathbf{Var}_i$ and $\mathfrak{P}_\mathsf{V}$ is a logical formula describing the possible values of these variables. Given a set $X \subseteq \mathbf{Var}_i$, we define $\mathfrak{P}_\mathsf{S}[X]$ as $\bigsqcup_{x \in X} \mathfrak{P}_\mathsf{S}(x)$. We denote by $\mathfrak{P} \,\mathsf{E}\, \mathcal{P}$ the fact that $\mathfrak{P} \in \{(\biguplus_{i \in \mathbf{D}_\mathcal{P}} \mathfrak{P}_{i\mathsf{S}}, \bigwedge_{i \in \mathbf{D}_\mathcal{P}} \mathfrak{P}_{i\mathsf{V}}) \mid \forall i \in \mathbf{D}_\mathcal{P} : \mathfrak{P}_i \in \mathcal{P}(i)\}$.

We also allow the specification of a (global) set $\mathcal{P}^{\mathrm{ch}}$ of channel policies. The set $\mathcal{P}^{\mathrm{ch}}$ has a distinguished member $\mathfrak{P}^\circ : \mathbf{PCh} \to \{H, L\}$ that gives the confidentiality level of the "presence" of communications over each polyadic channel (note that unlike $\mathfrak{P}_\mathsf{S}$ and $\mathfrak{P}_\mathsf{V}$, $\mathfrak{P}^\circ$ is not a component of $\mathfrak{P}$). Apart from $\mathfrak{P}^\circ$, there is at least one content policy $\mathfrak{P}^\bullet \in \mathcal{P}^{\mathrm{ch}}$. Each $\mathfrak{P}^\bullet$ has two components $\mathfrak{P}^\bullet_\mathsf{S} : \mathbf{Ch} \to \{H, L\}$ and $\mathfrak{P}^\bullet_\mathsf{V} : \mathbf{Ch} \to \mathbf{Lab}_\mathbf{V}$, where for each component channel c, $\mathfrak{P}^\bullet_\mathsf{S}(c)$ is the confidentiality level of the communication contents over c, and $\mathfrak{P}^\bullet_\mathsf{V}(c)$ is the set of values potentially communicated over c.

We will use $\mathcal{P}^{\mathrm{ch}}_\bullet$ to represent $\{\mathfrak{P}^\bullet \mid \mathfrak{P}^\bullet \in \mathcal{P}^{\mathrm{ch}}\}$, i.e., the set of *content* policies for channels. Hereafter, the parameterization on $\mathcal{P}^{\mathrm{ch}}$ in our formulations will often be elided since $\mathcal{P}^{\mathrm{ch}}$ is treated as a global constant. The distinguished presence policy $\mathfrak{P}^\circ \in \mathcal{P}^{\mathrm{ch}}$ will be left implicit for the same reason.

For a variable policy $\mathfrak{P}$, $\mathfrak{P}_\mathsf{V}$ can capture "relational constraints" between different variables. For instance, given an output binder $c!(x-y)$, it is legitimate to have $P_\mathsf{V} = (p(x) = p(y))$ where $p(-)$ is the *parity* function. Correspondingly, a channel policy $\mathfrak{P}^\bullet$ may come with *the set of even numbers* for $\mathfrak{P}^\bullet_\mathsf{V}(c.1)$.

The structure of our policy environments and policies are shown in Figure 5.4.

Example 5.2 *We will use the following policies for the multiplexer example presented in Section 5.1, where $\mathbb{Z}$ is the set of all integers.*

$$\mathcal{P}_{\mathrm{MD}}(1) = \{\, \mathfrak{P}_{\mathrm{m}} = (x_1 : H;\ x_2 : L,\ \mathbf{tt}) \,\}$$

$$\begin{aligned}\mathcal{P}_{\mathrm{MD}}(2) = \{\, &\mathfrak{P}^1_{\mathrm{d}} = (y : L;\ z : H;\ z_1 : H;\ z_2 : L,\ y = 1),\\ &\mathfrak{P}^2_{\mathrm{d}} = (y : L;\ z : L;\ z_1 : H;\ z_2 : L,\ y \neq 1) \,\}\end{aligned}$$

$$\begin{aligned}\mathcal{P}^{\mathrm{ch}}_{\mathrm{MD}} = \{\, &\mathfrak{P}^\circ = (c : L),\\ &\mathfrak{P}^\bullet_1 = (c.1 : L;\ c.2 : H,\ c.1 : \{1\};\ c.2 : \mathbb{Z}),\\ &\mathfrak{P}^\bullet_2 = (c.1 : L;\ c.2 : L,\ c.1 : \{2\};\ c.2 : \mathbb{Z}) \,\}\end{aligned}$$

For convenience of reference, the policies are named. Take the policy $\mathfrak{P}_{\mathrm{m}} \in \mathcal{P}(1)$ for example, we have $\mathfrak{P}_{\mathrm{m}\mathsf{V}} = \mathbf{tt}$ and $\mathfrak{P}_{\mathrm{m}\mathsf{S}} = [x_1 \mapsto H][x_2 \mapsto L]$. The

[2]The letter $\mathfrak{P}$ is the Fraktur variant of P, which has itself been taken to represent processes of the Quality Calculus in previous chapters.

$$\mathcal{P} = [1 \mapsto \mathcal{P}(1)] \uplus ... \uplus [n \mapsto \mathcal{P}(n)]$$
$$\mathcal{P}(i) = \{\mathfrak{P} \mid \mathfrak{P} = (\mathfrak{P}_{\mathsf{S}}, \mathfrak{P}_{\mathsf{V}}),\ \text{where } \mathfrak{P}_{\mathsf{S}} \in \mathbf{Var}_i \to \{H, L\},\ \mathfrak{P}_{\mathsf{V}} \in \mathbf{Lab_F}\}$$

$$\mathcal{P}^{\mathrm{ch}} = \{\mathfrak{P}^{\circ}, \mathfrak{P}^{\bullet}_1, \mathfrak{P}^{\bullet}_2, ..., \mathfrak{P}^{\bullet}_m\}$$
$$\mathfrak{P}^{\circ} \in \mathbf{PCh} \to \{H, L\}$$
$$\mathfrak{P}^{\bullet}_j = (\mathfrak{P}^{\bullet}_{j\mathsf{S}}, \mathfrak{P}^{\bullet}_{j\mathsf{V}}),\ \text{where } \mathfrak{P}^{\bullet}_{j\mathsf{S}} = \mathbf{Ch} \to \{H, L\},\ \text{and } \mathfrak{P}^{\bullet}_{j\mathsf{V}} = \mathbf{Ch} \to \mathbf{Lab_V}$$

Figure 5.4: The Structure of Policy Environments and Policies

syntax with colons and semi-colons is used for confidentiality policy components such as $\mathfrak{P}_{\mathrm{m}\mathsf{S}}$ *for conciseness.*

The policies $\mathfrak{P}^1_{\mathrm{d}}$ *and* $\mathfrak{P}^2_{\mathrm{d}}$ *clearly associate the condition* $y = 1$ *with* z *being confidential* ($z : H$)*, and the condition* $y \neq 1$ *with* z *being public* ($z : L$)*. The policies* $\mathfrak{P}^{\bullet}_1$ *and* $\mathfrak{P}^{\bullet}_2$ *relate the content communicated over the first component of the polyadic channel* c *with the different confidentiality levels of the second component of* c*.* □

We next define the satisfaction of variable policies by states ($\sigma \models \mathfrak{P}$), and the satisfaction of channel policies by actions ($\alpha \models_\rho \mathfrak{P}^{\bullet}$). Our concern here is what policies are relevant according to the memory content or communication content.

Definition 5.1 (Satisfaction)

$$\sigma \models \mathfrak{P} \ \triangleq\ \sigma \models \mathfrak{P}_{\mathsf{V}} \quad (\sigma \text{ is a model of the formula } \mathfrak{P}_{\mathsf{V}})$$
$$\alpha \models_\rho \mathfrak{P}^{\bullet} \ \triangleq\ \mathfrak{P}^{\bullet} \in \mathcal{P}^{\mathrm{ch}} \wedge \forall c, \vec{v} : \alpha = c\rho\vec{v} \Rightarrow \forall j\ s.t.\ 1 \leq j \leq |\vec{v}| : v_j \in \mathfrak{P}^{\bullet}_{\mathsf{V}}(c.j)$$

For channel policies, the satisfaction relation $\models_\rho$ is parameterized with a polarity ρ. The intuition is that the check on content is turned on only when the polarity of α is ρ. In this case it is required that the j-th value communicated over the polyadic channel of α should indeed be described by the value component of $\mathfrak{P}^{\bullet}$ for the component channel $c.j$. If α does not have the polarity ρ, then nothing is required. This mechanism for switching on/off the content-check is key to making the formulation of our communication-aware bisimulation (Section 5.4) compact.

It is desirable to require that the selection of policies cannot be decided by the confidential information contained in states and communication content. This is because confidentiality levels have *access control* implications. In more

detail, if confidential information had interference on the policies in use, a public observer would be able to deduce information about confidential variables based on changes in its access rights to certain variables.

We define the predicate $nip(\_)$ (which will be used in our type system in Section 5.5) to express this requirement, with the help of the notations $\sigma_1 \overset{\mathcal{K}}{=} \sigma_2$ and $c\rho'\vec{v}_1 \underset{\rho}{\overset{\mathcal{K}}{=}} c\rho'\vec{v}_2$. The notation $\sigma_1 \overset{\mathcal{K}}{=} \sigma_2$ represents that σ_1 and σ_2 have the same domain and map each variable that is *low with respect to every policy in* $\mathcal{K}$ to the same value. Similarly, $c\rho'\vec{v}_1 \underset{\rho}{\overset{\mathcal{K}}{=}} c\rho'\vec{v}_2$ says that if ρ is the same as ρ', then the component channels of c that are *low with respect to every policy in* $\mathcal{K}$ should communicate the same values.

Definition 5.2 (Low-Equivalence over Sets)

$$\sigma_1 \overset{\mathcal{K}}{=} \sigma_2 \triangleq \mathbf{D}_{\sigma_1} = \mathbf{D}_{\sigma_2} \wedge \forall x \in \mathbf{D}_{\sigma_1} : (\forall \mathfrak{P} \in \mathcal{K} : \mathfrak{P}_{\mathsf{S}}(x) = L) \Rightarrow \sigma_1(x) = \sigma_2(x)$$

$$c\rho'\vec{v}_1 \underset{\rho}{\overset{\mathcal{K}}{=}} c\rho'\vec{v}_2 \triangleq \forall j : (\rho = \rho' \wedge \forall \mathfrak{P}^{\bullet} \in \mathcal{K} : \mathfrak{P}^{\bullet}_{\mathsf{S}}(c.j) = L) \Rightarrow v_{1j} = v_{2j}$$

Definition 5.3 ($nip(\_)$)

$$\begin{aligned} nip(\mathcal{K}) \triangleq\ & \forall \sigma_1, \sigma_2 : \sigma_1 \overset{\mathcal{K}}{=} \sigma_2 \Rightarrow (\forall \mathfrak{P} \in \mathcal{K} : \sigma_1 \models \mathfrak{P} \Leftrightarrow \sigma_2 \models \mathfrak{P}) \wedge \\ & \forall c, \vec{v}, \vec{v}' : c!\vec{v} \underset{!}{\overset{\mathcal{P}^{\mathrm{ch}}_{\bullet}}{=}} c!\vec{v}' \Rightarrow (\forall \mathfrak{P}^{\bullet} : c!\vec{v} \models_! \mathfrak{P}^{\bullet} \Leftrightarrow c!\vec{v}' \models_! \mathfrak{P}^{\bullet}). \end{aligned}$$

Solid semantic underpinnings of our content-dependent flow policies will be provided by the noninterference properties to be developed in the next section and Section 5.6.

5.4 Concurrent, Content-Dependent Noninterference

We introduce a bisimulation-based, compositional noninterference property that accounts for both the communications performed by a system and the modification of memory content. Building on Definition 5.1 and Definition 5.2 from the previous section, some auxiliary concepts and notations are first introduced in Figure 5.5.

We extend our transition labels α with *inaction* ϵ and *suspension* $\square$. We also introduce *action schemas* β where the inputs come with holes rather than data. The idea is analogous to that of the abstract binders used in Chapter 4: the data to be received is under the environment's control. The difference is: action schemas represent possibly incomplete actions of *systems*, while abstract binders are possibly unfulfilled communication attempts of the *environment*. For

Actions and action schemas:

$$\begin{aligned}\alpha &::= c!\vec{v} \mid c?\vec{v} \mid \tau \mid \epsilon \mid \Box\\ \beta &::= c!\vec{v} \mid c?[\,] \mid \tau \mid \epsilon \mid \Box\\ \gamma &::= \alpha \mid \beta\end{aligned}$$

Decorated transitions:

$$\begin{aligned} C \overset{\alpha}{\twoheadrightarrow}_\eta C' \;\triangleq\;& C \xrightarrow{\alpha}_\eta C' \vee\\ & \alpha = \epsilon \wedge C' = C \vee\\ & (\forall \alpha' : C \xrightarrow{\alpha'} \Rightarrow h\_prc(\alpha')) \wedge \alpha = \Box \wedge C' = C\\ h\_prc(\alpha') \triangleq\;& \exists c, \rho, \vec{v} : \alpha' = c\rho\vec{v} \wedge \mathfrak{P}^\circ(c) = H \end{aligned}$$

Low equivalence of configurations:

$$\langle \Sigma_1; \sigma_1\rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{\equiv}} \langle \Sigma_2; \sigma_2\rangle \;\triangleq\; \mathfrak{P}\,\mathsf{E}\,\mathcal{P} \wedge Pid(\Sigma_1) = Pid(\Sigma_2) = \mathbf{D}_{\mathcal{P}} \wedge \sigma_1 \models \mathfrak{P} \wedge \sigma_2 \models \mathfrak{P} \wedge \sigma_1 \overset{\{\mathfrak{P}\}}{=} \sigma_2$$

Low equivalence of actions and actions/action schemas ($\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$):

$$\begin{aligned} & c\rho'\vec{v} \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \epsilon \;\text{ if } \mathfrak{P}^\circ(c) = H \wedge c\rho'\vec{v} \models_\rho \mathfrak{P}^\bullet\\ & c\rho'\vec{v}_1 \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} c\rho'\vec{v}_2 \;\text{ if } \mathfrak{P}^\circ(c) = L \wedge c\rho'\vec{v}_1 \models_\rho \mathfrak{P}^\bullet \wedge c\rho'\vec{v}_2 \models_\rho \mathfrak{P}^\bullet \wedge c\rho'\vec{v}_1 \overset{\{\mathfrak{P}^\bullet\}}{\underset{\rho}{=}} c\rho'\vec{v}_2\\ & \tau \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \tau\\ & \tau \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \Box \end{aligned}$$

Hole filling ($\beta(\vec{v}')$):

$$c!\vec{v}(\vec{v}') = c!\vec{v} \quad c?[\,](\vec{v}') = c?\vec{v}' \quad \beta(\vec{v}') = \beta, \text{ if } \beta \in \{\tau, \epsilon, \Box\}$$

Figure 5.5: Auxiliary Definitions for Content-Dependent Noninterference

transitions, we write $C \overset{\alpha}{\twoheadrightarrow}_\eta C'$ to represent that there *may be* a transition made by processes listed in η from the configuration C to the configuration C'. In more detail, there are three cases — where there is indeed a transition performing the action α, where there is a zero-step transition performing the action ϵ, and where there is suspension by blocked communications with confidential presence, represented by $\Box$.

Low-equivalence of configurations are defined with respect to particular policies $\mathfrak{P}\,\mathsf{E}\,\mathcal{P}$ in Figure 5.5. The main constraints are that P should be satisfied by the states of the two configurations, and that the values of variables that are

low under $\mathfrak{P}_{\mathsf{S}}$ should be equal.

To be able to relate the communications performed in the two executions in our bisimulation-based property, we introduce a notion of low equivalence ($\overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}}$) between actions α and actions/action schemas γ. The relation $\overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}}$ is the smallest one satisfying the rules in Figure 5.5. Concerning the "presence" of communication, $\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$ requires that a communication with confidential presence should correspond to inaction (ϵ), which implies among others the absence of communication on the same channel. This follows the pattern of our $\rhd$-bisimulation presented in Chapter 3. It is worth pointing out that the $\vec{v}_2$ in the same definition can be the unary vector $[\,]$. Although Definition 5.1 has not been explicitly extended to take care of holes, the bisimulation to be given in Definition 5.5 will use $\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$ in such a way that the check $[\,] \in \mathfrak{P}^\bullet_{\mathsf{V}}(c.1)$ can never be reached.

It is perhaps most reasonable for a τ to (be simulated by) correspond to a τ. This is described by the third line in the definition of $\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$. However, not all intuitively secure programs adhere to this strict pattern, as demonstrated by the following example.

Example 5.3 *Consider degenerate policy environments $\mathcal{P}_0$ and $\mathcal{P}_0^{\mathrm{ch}}$ such that*

$$\begin{aligned}\mathcal{P}_0(1) &= \{(h : H,\ \mathbf{tt})\}\\ \mathcal{P}_0(2) &= \{(h' : H, \mathbf{tt})\}\\ \mathcal{P}_0^{\mathrm{ch}} &= \{(c : H; c' : H),\ (c.1 : H; c'.1 : H, c.1 : \mathbb{Z}; c'.1 : \mathbb{Z})\}\end{aligned}$$

The following system is then intuitively secure under $\mathcal{P}_0$ and $\mathcal{P}_0^{\mathrm{ch}}$, since no security class is public. However, considering different branches of the if *being taken due to different initial values of h, the τ action arising out of* skip *should correspond to the confidential communication $c!2$, rather than another τ.*

$$1 : \mathsf{if}\ h > 1\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ c!2$$

Similarly, the following system is intuitively secure. However, the τ action produced by synchronization over c corresponds to suspension of execution, since $c'!2$ cannot synchronize with any communication binder.

$$(1 : \mathsf{if}\ h > 1\ \mathsf{then}\ c!2\ \mathsf{else}\ c'!2 \,||\, 2 : c?h') \setminus \{c, c'\}$$

This thus motivates the inclusion of $\tau \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \Box$ in the defining rules for $\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$. $\Box$

We are now in a position to define our noninterference property, termed "communication-aware security" (CA-security). In Definition 5.4, $- \overset{\mathrm{com}}{\underset{\mathcal{P}}{\sim}} -$ is the union of all communication-aware bisimulations (CA-bisimulations) that are in turn characterized in Definition 5.5.

Definition 5.4 (CA-Security) $Sec_{\mathrm{com}}(\Sigma, \mathcal{P})$ if and only if for all σ_1, σ_2, and $\mathfrak{P}$, if $\langle \Sigma; \sigma_1 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} \langle \Sigma; \sigma_2 \rangle$, then $(\langle \Sigma; \sigma_1 \rangle, \mathfrak{P}) \overset{\mathrm{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma; \sigma_2 \rangle, \mathfrak{P})$.

Definition 5.5 (CA-Bisimulation)
A CA-bisimulation $R_{\mathcal{P}}$ is a symmetric relation such that $(C_1, \mathfrak{P})\ R_{\mathcal{P}}\ (C_2, \mathfrak{P})$ implies $C_1 \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} C_2$ and the following:

$$\begin{array}{l} \forall \alpha, \eta, C_1' \ s.t.\ C_1 \xrightarrow{\alpha}_{\eta} C_1' : \\ \quad \exists \mathfrak{P}_!^{\bullet}, \beta : \alpha \overset{\mathfrak{P}_!^{\bullet}}{\underset{!}{\sim}} \beta \ \wedge \\ \quad\quad \forall \mathfrak{P}_?^{\bullet}, \vec{v} \ s.t.\ \alpha \overset{\mathfrak{P}_?^{\bullet}}{\underset{?}{\sim}} \beta(\vec{v}) : \\ \quad\quad\quad \exists C_2', \mathfrak{P}' : C_2 \overset{\beta(\vec{v})}{\twoheadrightarrow}_{\eta} C_2' \ \wedge \ (C_1', \mathfrak{P}')\ R_{\mathcal{P}}\ (C_2', \mathfrak{P}'). \end{array}$$

In prose, a symmetric relation $R_{\mathcal{P}}$ qualifies as a CA-bisimulation if for a pair $(C_1, \mathfrak{P})$ and $(C_2, \mathfrak{P})$ related by $R_{\mathcal{P}}$, and a transition performing action α from C_1, involving processes in η, there exists a policy $\mathfrak{P}_!^{\bullet}$ and an action schema β low-equivalent to α concerning output, and for all value vectors $\vec{v}$ and policies $\mathfrak{P}_?^{\bullet}$ such that $\beta(\vec{v})$ is low-equivalent to α concerning input, there exists a simulation of $\xrightarrow{\alpha}_{\eta}$ by $\overset{\beta(\vec{v})}{\twoheadrightarrow}_{\eta}$ from C_2, and a policy $\mathfrak{P}'$ whose pairings with the configurations reached are still related under $R_{\mathcal{P}}$. The universal quantification over $\vec{v}$ introduces variation on any input content that is constrained by $- \overset{\mathfrak{P}_?^{\bullet}}{\underset{?}{\sim}} -$ to be over the component channels that are confidential according to $\mathfrak{P}_?^{\bullet}$. Note that the ! and ? in $\mathfrak{P}_!^{\bullet}$ and $\mathfrak{P}_?^{\bullet}$ merely serve to signify that the two policies are respectively concerned with regulating output and input (removing them and using $\mathfrak{P}^{\bullet}$ at both places does not change the definition), while the use of ! and ? below the $\sim$ is a parameterization.

Example 5.4 *To aid the reader's intuition, we provide a* partial *unfolding of a CA-bisimulation for the system* $2 : S_{\mathrm{D}}$. *Note that a proof of the CA-security of* $2 : S_{\mathrm{D}}$ *is* not *the aim here. We represent by* $\sigma_{v_1 v_2 v_3 v_4}$ *the local state* $[y \mapsto v_1][z \mapsto v_2][z_1 \mapsto v_3][z_2 \mapsto v_4]$, *and by* $\mathcal{P}_{\mathrm{D}}$ *the policy environment* $[2 \mapsto \mathcal{P}_{\mathrm{MD}}(2)]$.

We have $\langle 2 : S_{\mathrm{D}}; \sigma_{2070} \rangle \overset{\mathfrak{P}_{\mathrm{d}}^2}{\underset{\mathcal{P}_{\mathrm{D}}}{=}} \langle 2 : S_{\mathrm{D}}; \sigma_{2080} \rangle$. *Hence one of the conditions that* $Sec_{\mathrm{com}}(2 : S_{\mathrm{D}}, \mathcal{P}_{\mathrm{D}})$ *calls for is the existence of CA-bisimulation* $R_{\star}$ *such that*

$$(\langle 2 : S_{\mathrm{D}}; \sigma_{2070} \rangle, \mathfrak{P}_{\mathrm{d}}^2)\ R_{\star}\ (\langle 2 : S_{\mathrm{D}}; \sigma_{2080} \rangle, \mathfrak{P}_{\mathrm{d}}^2).$$

Suppose

$$\langle 2 : S_{\mathrm{D}}; \sigma_{2070} \rangle \xrightarrow{\tau}_{2} \langle 2 : c?(y, z); \underline{if}; \underline{wh}; \sigma_{2070} \rangle \tag{5.1}$$

There exist $\mathfrak{P}_1^\bullet$ *and* τ*, such that* $\tau \overset{\mathfrak{P}_1^\bullet}{\underset{!}{\sim}} \tau$*. Pick for instance* $\mathfrak{P}_?^\bullet = \mathfrak{P}_1^\bullet$ *and* $\vec{v} = (0,0)$*, for which* $\tau \overset{\mathfrak{P}_1^\bullet}{\underset{?}{\sim}} \tau(0,0)$*. Simulation of* (5.1) *is required with the action* $\tau(0,0) = \tau$*. The only possibility is* $\langle 2 : S_{\mathrm{D}}; \sigma_{2080}\rangle \xrightarrow{\tau}_2 \langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2080}\rangle$*. Since* $\sigma_{2070} \models \mathfrak{P}_{\mathrm{d}}^2$ *but* $\sigma_{2070} \not\models \mathfrak{P}_{\mathrm{d}}^1$*, the following is required:*

$$(\langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2070}\rangle, \mathfrak{P}_{\mathrm{d}}^2)\ R_\star\ (\langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2080}\rangle, \mathfrak{P}_{\mathrm{d}}^2).$$

This further necessitates the condition below, which can be verified easily:

$$\langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2070}\rangle \overset{\mathfrak{P}_{\mathrm{d}}^2}{\underset{\mathcal{P}_{\mathrm{D}}}{=}} \langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2080}\rangle.$$

Suppose

$$\langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2070}\rangle \overset{c?(1,k_1)}{\longrightarrow}_2 \langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_170}\rangle, \tag{5.2}$$

where k_1 *is an integer. There should exist some* $\mathfrak{P}_!^\bullet$ *and* β *such that* $c?(1,k_1) \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta$*. Since* $\mathfrak{P}^\circ(c) = L$*,* $\beta = c?[\,]$*. Pick for instance* $\mathfrak{P}_?^\bullet = \mathfrak{P}_1^\bullet$*, and* $\vec{v}$*,* $c?(1,k_1) \overset{\mathfrak{P}_1^\bullet}{\underset{?}{\sim}} c?[\,](\vec{v})$ *implies* $v_1 = 1$ *since* $\mathfrak{P}_1^\bullet(c.1) = L$*. Hence a simulation of* (5.2) *with action* $c?[\,](1,k_2)$ *is required for all* $k_2 \in \mathbb{Z}$ *(*$\mathfrak{P}_1^\bullet(c.2) = H$*). It can only be of the form* $\langle 2 : c?(y,z); \underline{if}; \underline{wh}; \sigma_{2080}\rangle \overset{c?(1,k_2)}{\longrightarrow}_2 \langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_280}\rangle$*. And the following is required*

$$(\langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_170}\rangle, \mathfrak{P}_{\mathrm{d}}^1)\ R_\star\ (\langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_280}\rangle, \mathfrak{P}_{\mathrm{d}}^1).$$

This further requires $\langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_170}\rangle \overset{\mathfrak{P}_{\mathrm{d}}^1}{\underset{\mathcal{P}_{\mathrm{D}}}{=}} \langle 2 : \underline{if}; \underline{wh}; \sigma_{1k_280}\rangle$*, which holds.*

Going through two more "lock steps", the following is required.

$$\begin{aligned}(\langle 2 : z_1 := z; \underline{wh}; \sigma_{1k_170}\rangle, \mathfrak{P}_{\mathrm{d}}^1)\ R_\star\ (\langle 2 : z_1 := z; \underline{wh}; \sigma_{1k_280}\rangle, \mathfrak{P}_{\mathrm{d}}^1)\\ (\langle 2 : S_{\mathrm{D}}; \sigma_{1k_1k_10}\rangle, \mathfrak{P}_{\mathrm{d}}^1)\ R_\star\ (\langle 2 : S_{\mathrm{D}}; \sigma_{1k_2k_20}\rangle, \mathfrak{P}_{\mathrm{d}}^1)\end{aligned} \tag{5.3}$$

And we still have $\langle 2 : S_{\mathrm{D}}; \sigma_{1k_1k_10}\rangle \overset{\mathfrak{P}_{\mathrm{d}}^1}{\underset{\mathcal{P}_{\mathrm{D}}}{=}} \langle 2 : S_{\mathrm{D}}; \sigma_{1k_2k_20}\rangle$ *as required by* (5.3)*.*

We stop this demonstration here. □

In Example 5.4, the systems are always the same on both sides of $R_\star$, which is a special case due partly to $\mathcal{E}^\circ(c) = L$ in $\mathcal{P}_{\mathrm{MD}}^{\mathrm{ch}}$.

In CA-bisimulation, the treatment of output and input is given separately in the two lines in the middle of the formulation.

$$\begin{aligned}&\ldots\\ &\quad \exists \mathfrak{P}_!^\bullet, \beta : \alpha \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta \wedge\\ &\qquad \forall \mathfrak{P}_?^\bullet, \vec{v}\ s.t.\ \alpha \overset{\mathfrak{P}_?^\bullet}{\underset{?}{\sim}} \beta(\vec{v}) :\\ &\qquad\quad \ldots\end{aligned}$$

The channel policies ($\mathfrak{P}^{\bullet}_{!}$ and $\mathfrak{P}^{\bullet}_{?}$) used to related the communication actions/schemas are not kept along with the configurations, since communications are instantaneous behaviors. The pattern that there should exist a channel policy for the output content, and that the simulation can be performed under all potential channel policies for the input content, resembles that of rely-guarantee reasoning [Jon81], and leads to the preservation of security under $||$ — the second compositionality result given below. Note that since $\mathfrak{P}^{\bullet}_{?}$ does not occur in the last line of Definition 5.5, the second last line of the definition can also be written as "$\forall \vec{v}$ *s.t.* $(\exists \mathfrak{P}^{\bullet}_{?} : \alpha \overset{\mathfrak{P}^{\bullet}_{?}}{\underset{?}{\sim}} \beta(\vec{v}))$:".

Compositionality of CA-Security CA-security is compositional with respect to the two non-trivial constructors, $\cdot \setminus \Omega$ and $\cdot||\cdot$, of systems.

Theorem 5.6 (Compositionality) *For Σ_1 with policy environment $\mathcal{P}_1$, and Σ_2 with policy environment $\mathcal{P}_2$, such that $\mathbf{D}_{\mathcal{P}_1} \cap \mathbf{D}_{\mathcal{P}_2} = \emptyset$,*

1. $Sec_{\mathrm{com}}(\Sigma_1, \mathcal{P}_1) \implies \forall \Omega \subseteq \mathbf{PCh} : Sec_{\mathrm{com}}(\Sigma_1 \setminus \Omega, \mathcal{P}_1)$, *and*
2. $Sec_{\mathrm{com}}(\Sigma_1, \mathcal{P}_1) \wedge Sec_{\mathrm{com}}(\Sigma_2, \mathcal{P}_2) \implies Sec_{\mathrm{com}}(\Sigma_1 || \Sigma_2, \mathcal{P}_1 \uplus \mathcal{P}_2)$.

Given the information flow type system developed in Section 5.5, a compositionality result of the security property appears to play a weaker role in directly supporting compositional security analyses. However, this result is a key element in proving the soundness of our information flow type system. Aside from this, it opens up new possibilities for establishing the security of individual components using alternative approaches, and obtaining the security of the whole system "for free".

5.5 Typing Information Flow

We specify a type system for ensuring that a system Σ respects the information flow policies given by $\mathcal{P}$ (such that $\mathbf{D}_{\mathcal{P}} = Pid(\Sigma)$) and $\mathcal{P}^{\mathrm{ch}}$. To deal with the value components $\mathfrak{P}_{\mathsf{V}}$ of policies $\mathfrak{P}$, the type system is integrated with a Hoare logic for reasoning about the values of variables [Apt81, Apt84]. The typing rules for processes and systems are specified in Table 5.6 in order.

5.5.1 The Typing of Processes

The judgements of the type system for processes take the form

$$X, l_1 \vdash_{\mathcal{K}} \{\phi\}\, S\, \{\phi'\} : Y, l_2.$$

Here X is a set of variables that may incur implicit flows [DD77], Y is a set of variables whose information can be leaked through *progress*, $\mathcal{K}$ is the set

Table 5.6: Information Flow Type System for Processes and Systems.

$$X, l \vdash_{\mathcal{K}} \{\phi\}\ \mathsf{nil}\ \{\phi\} : \emptyset, L$$

$$X, l \vdash_{\mathcal{K}} \{\phi\}\ \mathsf{skip}\ \{\phi\} : \emptyset, L$$

$$X, l \vdash_{\mathcal{K}} \{\phi[a/x]\}\ x := a\ \{\phi\} : \emptyset, L$$
$$\text{if } \forall \mathfrak{P} \in \mathcal{K} : \mathfrak{P}_{\mathsf{V}} \wedge \phi[a/x] \Rightarrow \exists \mathfrak{P}' \in \mathcal{K} : \mathfrak{P}[x \mapsto l \sqcup \mathfrak{P}_{\mathsf{S}}[fv(a) \cup X]]_{\mathsf{S}} \preceq \mathfrak{P}'[a/x]_{\mathsf{V}}$$

$$\frac{X, l \vdash_{\mathcal{K}} \{\phi\}\ S_1\ \{\rho\} : Y_1, l_1 \quad X \cup Y_1, l \sqcup l_1 \vdash_{\mathcal{K}} \{\rho\}\ S_2\ \{\psi\} : Y_2, l_2}{X, l \vdash_{\mathcal{K}} \{\phi\}\ S_1; S_2\ \{\psi\} : Y_1 \cup Y_2, l_1 \sqcup l_2}$$

$$\frac{X \cup fv(b), l \vdash_{\mathcal{K}} \{\phi \wedge b\}\ S_1\ \{\psi\} : Y_1, l_1 \quad X \cup fv(b), l \vdash_{\mathcal{K}} \{\phi \wedge \neg b\}\ S_2\ \{\psi\} : Y_2, l_2}{X, l \vdash_{\mathcal{K}} \{\phi\}\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \{\psi\} : Y_1 \cup Y_2 \cup fv(b), l_1 \sqcup l_2}$$

$$\frac{Y, l \vdash_{\mathcal{K}} \{\phi \wedge b\}\ S\ \{\phi\} : Y, l}{X, l \vdash_{\mathcal{K}} \{\phi\}\ \mathsf{while}\ b\ \mathsf{do}\ S\ \{\phi \wedge \neg b\} : Y, l} \quad \text{if } X \cup fv(b) \subseteq Y$$

$$X, l \vdash_{\mathcal{K}} \{\phi\}\ c!\vec{a}\ \{\phi\} : \emptyset, l' \quad \text{if } l \sqsubseteq \mathfrak{P}^{\circ}(c) \sqsubseteq l' \text{ and}$$
$$\forall \mathfrak{P} \in \mathcal{K} : \mathfrak{P}_{\mathsf{V}} \wedge \phi \Rightarrow (\mathfrak{P}_{\mathsf{S}}[X] \sqsubseteq \mathfrak{P}^{\circ}(c) \wedge\ \exists \mathfrak{P}' \in \mathcal{K}, \mathfrak{P}^{\bullet} \in \mathcal{P}^{\mathrm{ch}} :$$
$$\mathfrak{P}[(c.j \mapsto \mathfrak{P}_{\mathsf{S}}(a_j))_j]_{\mathsf{S}} \preceq (\mathfrak{P}'_S \uplus \mathfrak{P}^{\bullet}_{\mathsf{S}},\ \mathfrak{P}'_{\mathsf{V}} \wedge \textstyle\bigwedge_j a_j \in \mathfrak{P}^{\bullet}_{\mathsf{V}}(c.j)))$$

$$X, l \vdash_{\mathcal{K}} \{\forall \vec{x} : \phi\}\ c?\vec{x}\ \{\phi\} : \emptyset, l' \quad \text{if } l \sqsubseteq \mathfrak{P}^{\circ}(c) \sqsubseteq l' \text{ and}$$
$$\forall \mathfrak{P} \in \mathcal{K} : \mathfrak{P}_{\mathsf{V}} \wedge (\forall \vec{x} : \phi) \Rightarrow (\mathfrak{P}_{\mathsf{S}}[X] \sqsubseteq \mathfrak{P}^{\circ}(c) \wedge\ \forall \mathfrak{P}^{\bullet} \in \mathcal{P}^{\mathrm{ch}} : \forall \vec{v}\ s.t. \textstyle\bigwedge_j v_j \in \mathfrak{P}^{\bullet}_{\mathsf{V}}(c.j) :$$
$$\exists \mathfrak{P}' \in \mathcal{K} : \mathfrak{P}[(x_j \mapsto \mathfrak{P}^{\bullet}_{\mathsf{S}}(c.j) \sqcup \mathfrak{P}^{\circ}(c))_j]_{\mathsf{S}} \preceq \mathfrak{P}'[(\underline{v_j}/x_j)_j]_{\mathsf{V}})$$

$$\frac{X', l'_1 \vdash_{\mathcal{K}} \{\phi'\}\ S\ \{\psi'\} : Y', l'_2}{X, l_1 \vdash_{\mathcal{K}} \{\phi\}\ S\ \{\psi\} : Y, l_2} \quad \text{if} \quad \begin{array}{l}(\phi \Rightarrow \phi') \wedge (\psi' \Rightarrow \psi) \wedge \\ X \subseteq X' \wedge Y' \subseteq Y \wedge l_1 \sqsubseteq l'_1 \wedge l'_2 \sqsubseteq l_2\end{array}$$

$$\frac{\emptyset, L \vdash_{\mathcal{K}} \{\phi\}\ S_i\ \{\psi\} : Y, l'}{[i \mapsto \mathcal{K}] \vdash \{[i \mapsto \phi]\}\ i : S_i\ \{[i \mapsto \psi]\}} \text{ if } nip(\mathcal{K})$$

$$\frac{\mathcal{P} \vdash \{\Phi\}\ \Sigma\ \{\Psi\}}{\mathcal{P} \vdash \{\Phi\}\ \Sigma \setminus \Omega\ \{\Psi\}} \qquad \frac{\mathcal{P}_1 \vdash \{\Phi_1\}\ \Sigma_1\ \{\Psi_1\} \quad \mathcal{P}_2 \vdash \{\Phi_2\}\ \Sigma_2\ \{\Psi_2\}}{\mathcal{P}_1 \uplus \mathcal{P}_2 \vdash \{\Phi_1 \uplus \Phi_2\}\ \Sigma_1 || \Sigma_2\ \{\Psi_1 \uplus \Psi_2\}}$$

of variable policies for the process S, and ϕ and ϕ' are the pre- and post-conditions of S in the form of logical formulae over the variables local to S. In addition, l_1 and l_2 are the levels of information that can be leaked through blocked communication attempts (due to inability of synchronization), before reaching S, and within S, respectively. The levels l_1 and l_2 become H when encountering communication channels whose presence levels are H.

In Table 5.6, $\mathfrak{P} \preceq \mathfrak{P}'$ represents $\mathfrak{P}_\mathsf{S} \sqsubseteq \mathfrak{P}'_\mathsf{S} \wedge \mathfrak{P}_\mathsf{V} \Rightarrow \mathfrak{P}'_\mathsf{V}$, where $\mathfrak{P}_\mathsf{S} \sqsubseteq \mathfrak{P}'_\mathsf{S}$ if and only if $\forall u \in \mathbf{D}_{\mathfrak{P}_\mathsf{S}} \cap \mathbf{D}_{\mathfrak{P}'_\mathsf{S}} : \mathfrak{P}_\mathsf{S}(u) \sqsubseteq \mathfrak{P}'_\mathsf{S}(u)$. We write $\mathfrak{P}[x \mapsto l]_\mathsf{S}$ for $(\mathfrak{P}_\mathsf{S}[x \mapsto l], \mathfrak{P}_\mathsf{V})$, which is an update if $x \in \mathbf{D}_{\mathfrak{P}_\mathsf{S}}$ and an extension otherwise, $\mathfrak{P}[u/x]_\mathsf{V}$ for $(\mathfrak{P}_\mathsf{S}, \mathfrak{P}_\mathsf{V}[u/x])$ where u is an arithmetic expression or a component channel, $\mathfrak{P} \wedge f$ for $(\mathfrak{P}_\mathsf{S}, \mathfrak{P}_\mathsf{V} \wedge f)$ where f is a logical formula, and $\underline{v}$ for the numeral of the value v.

The Hoare logic used is not very different from a standard one for sequential programs, because our variables are local. Most typing rules strengthen a pre-condition ϕ to the formula $\phi \wedge \mathfrak{P}_\mathsf{V}$ that allows one to select the relevant variable policies $\mathfrak{P}$ using their content information $\mathfrak{P}_\mathsf{V}$. We elaborate on the rules for assignment, output and input.

As mentioned earlier, the nil construct is *not* part of the surface syntax, but a semantic cue. Still a trivial typing rule is devised for nil in order to have a subject reduction result.

The typing rule for assignment requires the existence of a post-policy $\mathfrak{P}'$ for each selected pre-policy $\mathfrak{P}$. This policy $\mathfrak{P}'$ should satisfy $l \sqcup \mathfrak{P}_\mathsf{S}[fv(a) \cup X] \sqsubseteq \mathfrak{P}'_\mathsf{S}(x)$, and for all variables y different than x, $\mathfrak{P}_\mathsf{S}(y) \sqsubseteq \mathfrak{P}'_\mathsf{S}(y)$ should hold. Requiring $l \sqsubseteq \mathfrak{P}'_\mathsf{S}(x)$ and $\mathfrak{P}_\mathsf{S}[X] \sqsubseteq \mathfrak{P}'_\mathsf{S}(x)$ is to capture *implicit flows* [DD77]. On the other hand, under the pre-condition $\mathfrak{P}_\mathsf{V} \wedge \phi[a/x]$, it is required that $\mathfrak{P}_\mathsf{V} \Rightarrow \mathfrak{P}'_\mathsf{V}[a/x]$. In other words, $\mathfrak{P}_\mathsf{V} \wedge \phi[a/x] \Rightarrow \mathfrak{P}'_\mathsf{V}[a/x]$ should hold. This guarantees that for a state σ satisfying $\mathfrak{P}$ and the precondition $\phi[a/x]$, the post state derived from σ after the assignment satisfies the post policy $\mathfrak{P}'$.

Example 5.5 *For the assignment $z_1 := z$ in the demultiplexer process S_D of Figure 5.1, we have $\{y\}, L \vdash_{\mathcal{P}_\mathrm{MD}(2)} \{y = 1\}\, z_1 := z\, \{y = 1\} : \emptyset, L$. Essentially, it needs to be shown that no matter if $\mathfrak{P}$ is instantiated with $\mathfrak{P}^1_\mathrm{d}$ or $\mathfrak{P}^2_\mathrm{d}$, we can find an appropriate policy in $\mathcal{P}_\mathrm{MD}(2)$ for the instantiation of $\mathfrak{P}'$, satisfying the side conditions of the typing rule for assignment. First instantiate $\mathfrak{P}$ with $\mathfrak{P}^1_\mathrm{d}$. We still use $\mathfrak{P}^1_\mathrm{d}$ for $\mathfrak{P}'$, and the side condition specializes to $y = 1 \Rightarrow \mathfrak{P}^1_\mathrm{d}[z_1 \mapsto L \sqcup \mathfrak{P}^1_{\mathrm{d}\mathsf{S}}[\{y, z\}]] \preceq \mathfrak{P}^1_\mathrm{d}[z/z_1]_\mathsf{V}$. This condition further expands to the following, which holds.*

$$y = 1 \quad \Rightarrow \quad ((y : L; z : H; z_1 : H; z_2 : L)[z_1 \mapsto H],\ y = 1) \preceq ((y : L; z : H; z_1 : H; z_2 : L),\ (y = 1)[z/z_1]).$$

Next instantiate $\mathfrak{P}$ with $\mathfrak{P}^2_\mathrm{d}$. The side condition specializes to $y = 1 \wedge y \neq 1 \Rightarrow ...$, which vacuously holds. □

The typing rule for output imposes the constraint $l \sqsubseteq \mathfrak{P}^\circ(c) \sqsubseteq l'$. Here $\mathfrak{P}^\circ(c) \sqsubseteq l'$ takes care of the possibility for the output to be blocked by the environment (in line with the use of synchronous communication the treatment of output is "symmetric" to that of input; hence the possiblity of blocked output is also considered). In more detail, the presence/absence of the output can leak information if the subsequent computation is not kept confidential. This kind of leakage is in a sense analogous to the leakage created by looping. On the other hand, $l \sqsubseteq \mathfrak{P}^\circ(c)$ takes care of the possibility that a previously blocked communication can be revealed through the indirect blockage (absence) of the current output. Next, the constraint $\mathfrak{P}_\mathsf{S}[X] \sqsubseteq \mathfrak{P}^\circ(c)$ is concerned with the implicit flows from conditionals having variables in X to the *presence* of the output. Finally, the seemingly involved constraint $\mathfrak{P}[(c.j \mapsto \mathfrak{P}_\mathsf{S}(a_j))_j]_\mathsf{S} \preceq (\mathfrak{P}'_S \uplus \mathfrak{P}^\bullet_\mathsf{S}, \mathfrak{P}'_\mathsf{V} \wedge \bigwedge_j a_j \in \mathfrak{P}^\bullet_\mathsf{V}(c.j))$ can be understood by comparing the output $c!\vec{a}$ to the assignment $c := \vec{a}$.

The typing rule for input uses constraints about the presence label $\mathfrak{P}^\circ(c)$ of the channel c in a way similar to the rule for output does. Its last constraint $\mathfrak{P}[(x_j \mapsto \mathfrak{P}^\bullet_\mathsf{S}(c.j) \sqcup \mathfrak{P}^\circ(c))_j]_\mathsf{S} \preceq \mathfrak{P}'[(\underline{v_j}/x_j)_j]_\mathsf{V}$, requires $\mathfrak{P}^\circ(c) \sqsubseteq P'_\mathsf{S}(x_j)$ for all $j \in \{1, 2, ..., |\vec{x}|\}$, because the presence of the input leads to the modification of the variable x_j. The remaining part of this constraint can be understood by comparing the input $c?\vec{x}$ to the assignment $\vec{x} := c$.

We remark on the typing rule for if, where the set $fv(b)$ is unioned into the "progress set", resulting in $fv(b) \cup Y_1 \cup Y_2$. This guarantees a noninterference condition where two systems advance in a manner close to "lock-step" execution, thereby facilitating the articulation of the security guarantees under scheduling. Similar treatment of the "termination effects" of if can be found in [BC02, Smi06].

5.5.2 The Typing of Systems

The typing judgements for systems are of the form

$$\mathcal{P} \vdash \{\Phi\}\, \Sigma\, \{\Psi\}.$$

Here $\mathcal{P}$ is a policy environment for the system Σ, and for each $i \in Pid(\Sigma)$, $\Phi(i)$ and $\Psi(i)$ are the pre-condition and post-condition, respectively, for the process with identifier i in Σ. We denote by $\mathbf{T}^\Sigma$ the mapping such that $\mathbf{D}_{\mathbf{T}^\Sigma} = Pid(\Sigma)$ and $\forall i \in \mathbf{D}_{\mathbf{T}^\Sigma} : \mathbf{T}^\Sigma(i) = \mathbf{tt}$. The typing rules follow patterns that are fairly straightforward. Recall that the purpose of $nip(\_)$ introduced in Definition 5.3 is to rule out undesirable policy-level interference.

Example 5.6 *The system Σ_{MD} of Example 5.1 can be typed using the policies given in Example 5.2. It is not difficult to verify that $nip(\mathcal{P}_{\mathrm{MD}}(1))$ and $nip(\mathcal{P}_{\mathrm{MD}}(2))$ hold, and that $\mathcal{P}_{\mathrm{MD}} \vdash \{\mathbf{T}^{\Sigma_{\mathrm{MD}}}\}\, \Sigma_{\mathrm{MD}}\, \{\mathbf{T}^{\Sigma_{\mathrm{MD}}}\}$ can be established. The reason that $nip(\mathcal{P}_{\mathrm{MD}}(2))$ holds is as follows. Pick two arbitrary states σ_1*

and σ_2 *for process* 2, $\sigma_1 \overset{\mathcal{P}_{\mathrm{MD}}(2)}{=} \sigma_2$ *means that* σ_1 *and* σ_2 *agrees on the values of variables that are low according to both policies in* $\mathcal{P}_{\mathrm{MD}}(2)$. *This implies* $\sigma_1(y) = \sigma_2(y)$, *which further implies that* $\sigma_1 \models \mathfrak{P}_{\mathrm{d}}^1 \Leftrightarrow \sigma_2 \models \mathfrak{P}_{\mathrm{d}}^1$, *and that* $\sigma_1 \models \mathfrak{P}_{\mathrm{d}}^2 \Leftrightarrow \sigma_2 \models \mathfrak{P}_{\mathrm{d}}^2$. *It can be verified that the other condition required about* $\mathcal{P}_{\mathrm{MD}}^{\mathrm{ch}}$ *is also satisfied.* □

5.5.3 Theoretical Properties

The safety of our type system is demonstrated by a subject reduction [Pie02] result — well-typedness is preserved under transition. In the statement of Theorem 5.7 below, $\sigma \models \Phi$ represents $\forall i \in \mathbf{D}_\Phi : \sigma \models \Phi(i)$ and all un-quantified symbols are implicitly universally quantified. When an input is performed, the existence of channel policies describing the values received are relied on to ensure the satisfaction of the pre-condition Φ' of the derived system Σ', by the resulting state σ'. It is not difficult to see that the subject reduction result also indicates that the Hoare logic component of the type system gives a partial correctness interpretation of well-typed systems.

Theorem 5.7 (Subject Reduction) *If* $\mathcal{P} \vdash \{\Phi\}\,\Sigma\,\{\Psi\}$, $\langle \Sigma; \sigma \rangle \xrightarrow{\alpha}_\eta \langle \Sigma'; \sigma' \rangle$, $\sigma \models \Phi$, $\mathfrak{P} \in \mathcal{P}$ *and* $\sigma \models \mathfrak{P}$, *then*

1. $\exists \mathfrak{P}^\bullet : \alpha \models_! \mathfrak{P}^\bullet$, *and*
2. *if* $\exists \mathfrak{P}^\bullet : \alpha \models_? \mathfrak{P}^\bullet$, *then* $\exists \mathfrak{P}' \in \mathcal{P}, \Phi' : \mathcal{P} \vdash \{\Phi'\}\,\Sigma'\,\{\Psi\} \wedge \sigma' \models \Phi' \wedge \sigma' \models \mathfrak{P}'$.

The key result of this chapter, that well-typedness guarantees CA-security, is formally stated in Theorem 5.8. This soundness result means that our motivating example is noninterfering (Example 5.7).

Theorem 5.8 (Soundness) *For all systems* Σ *with policy environments* $\mathcal{P}$, *if* $\mathcal{P} \vdash \{\mathbf{T}^\Sigma\}\,\Sigma\,\{\mathbf{T}^\Sigma\}$, *then* $Sec_{\mathrm{com}}(\Sigma, \mathcal{P})$.

Example 5.7 *Going back to the multiplexer example, by Theorem 5.8, the well-typedness of the system* Σ_{MD} *of Example 5.1 guarantees* $Sec_{\mathrm{com}}(\Sigma_{\mathrm{MD}}, \mathcal{P}_{\mathrm{MD}})$. □

The proof of Theorem 5.8 is sketched below, where the role played by the compositionality result will also be elucidated (Theorem 5.6). All details of this proof can be found in the Coq development.

We define high/low processes with the help of the type system, following the approach initiated in [VIS96, SV98]. To take care of content-dependent policies, the memory state σ and policy $\mathfrak{P}$ satisfied by σ are made explicit in the defined predicates.

Table 5.7: The Low Equivalence Relation on Singleton Systems

$$(\mathrm{LO_{EQ}})\quad \frac{\exists \phi : lo^{\mathcal{K}}_{\phi}(\langle S;\sigma_1\rangle,\mathfrak{P}) \wedge lo^{\mathcal{K}}_{\phi}(\langle S;\sigma_2\rangle,\mathfrak{P}) \quad \sigma_1 \overset{\{\mathfrak{P}\}}{=} \sigma_2 \quad nip(\mathcal{K})}{(\langle i : S;\, \sigma_1\rangle,\mathfrak{P}) \overset{[i\mapsto\mathcal{K}]}{\simeq} (\langle i : S;\, \sigma_2\rangle,\mathfrak{P})}$$

$$(\mathrm{HI_{EQ}})\quad \frac{hi^{\mathcal{K}}_{\phi_1}(\langle S_1;\sigma_1\rangle,\mathfrak{P}) \wedge hi^{\mathcal{K}}_{\phi_2}(\langle S_2;\sigma_2\rangle,\mathfrak{P}) \quad \sigma_1 \overset{\{\mathfrak{P}\}}{=} \sigma_2 \quad nip(\mathcal{K})}{(\langle i : S_1;\, \sigma_1\rangle,\mathfrak{P}) \overset{[i\mapsto\mathcal{K}]}{\simeq} (\langle i : S_2;\, \sigma_2\rangle,\mathfrak{P})}$$

Definition 5.9 A process S with set $\mathcal{K}$ of variable policies is *high at state σ, under precondition ϕ and with policy $\mathfrak{P}$*, written $hi^{\mathcal{K}}_{\phi}(\langle S;\sigma\rangle,\mathfrak{P})$, if

$$\sigma \models \mathfrak{P} \ \wedge\ \sigma \models \phi \ \wedge \\ \exists X, l, X', l', \psi : X, l \vdash_{\mathcal{K}} \{\phi\}\, S\, \{\psi\} : X', l' \ \wedge\ (l = H \vee \exists x \in X : \mathfrak{P}_{\mathsf{S}}(x) = H)$$

Definition 5.10 A process S with set $\mathcal{K}$ of variable policies is *low at state σ, under precondition ϕ and with policy $\mathfrak{P}$*, written $lo^{\mathcal{K}}_{\phi}(\langle S;\sigma\rangle,\mathfrak{P})$, if

$$\sigma \models \mathfrak{P} \ \wedge\ \sigma \models \phi \ \wedge \\ (\exists X, l, X', l', \psi : X, l \vdash_{\mathcal{K}} \{\phi\}\, S\, \{\psi\} : X', l') \ \wedge \\ (\forall X, l, X', l', \psi : X, l \vdash_{\mathcal{K}} \{\phi\}\, S\, \{\psi\} : X', l' \Rightarrow (l = L \wedge \forall x \in X : \mathfrak{P}_{\mathsf{S}}(x) = L))$$

We then define a low-equivalence relation $\overset{[i\mapsto\mathcal{K}]}{\simeq}$ concerned with systems of the form $i : S$ in Table 5.7. For two singleton systems with low processes to be related, the processes they execute need to be the same, and the preconditions used should be identical. The policies used on both sides of $\overset{[i\mapsto\mathcal{K}]}{\simeq}$ are always the same (say $\mathfrak{P}$), and the memory states used always need to be low-equivalent under $\mathfrak{P}$. The two rules ($\mathrm{LO_{EQ}}$) and ($\mathrm{HI_{EQ}}$) correspond to the two constructors `LEQ_LO` and `LEQ_HI` of the parameterized relation `low_eq` in the Coq formalization in Appendix B.5.

We next build up to the following results:

- $\overset{[i\mapsto\mathcal{K}]}{\simeq}$ is a CA-bisimulation, and
- if $[i \mapsto \mathcal{K}] \vdash \{[i \mapsto \mathbf{tt}]\}\, i : S\, \{[i \mapsto \mathbf{tt}]\}$ holds, and $\langle i : S;\sigma_1\rangle \underset{[i\mapsto\mathcal{K}]}{\overset{\mathfrak{P}}{=}} \langle i : S;\sigma_2\rangle$, then $(\langle i : S;\, \sigma_1\rangle,\mathfrak{P}) \overset{[i\mapsto\mathcal{K}]}{\simeq} (\langle i : S;\, \sigma_2\rangle,\mathfrak{P})$ holds.

In other words, if $[i \mapsto \mathcal{K}] \vdash \{[i \mapsto \mathbf{tt}]\}\, i : S\, \{[i \mapsto \mathbf{tt}]\}$ can be established, then $Sec_{\mathrm{com}}(i : S, [i \mapsto \mathcal{K}])$ holds. Suppose a sysmtem $\Sigma_\star$ has the form $\Sigma_1||\Sigma_2$ or $\Sigma' \setminus \Omega$, and the policy environment $\mathcal{P}_\star$. By the typing rule for systems, and the compositionality theorem (Theorem 5.6), we can then inductively show that $\mathcal{P}_\star \vdash \{\mathbf{T}^{\Sigma_\star}\}\, \Sigma_\star\, \{\mathbf{T}^{\Sigma_\star}\}$ implies $Sec_{\mathrm{com}}(\Sigma_\star, \mathcal{P}_\star)$.

Definition 5.11 *UpdNext*(S) represents the set of variables that the process S attempts to update *immediately*. Formally, *UpdNext*$(S) = \emptyset$ except for the following cases.

$$\begin{array}{l} \mathit{UpdNext}(x := a) = \{x\} \\ \mathit{UpdNext}(c?\vec{x}) = \{x_1, ..., x_k\} \text{ where } \vec{x} = (x_1, ..., x_k) \\ \mathit{UpdNext}(S_1; S_2) = \mathit{UpdNext}(S_1) \end{array}$$

Lemma 5.12 *If* $x \in \mathbf{Var}_i \setminus \mathit{UpdNext}(S)$, *and* $\vdash_i \langle S; \sigma\rangle \xrightarrow{\alpha} \langle S'; \sigma'\rangle$, *then we have* $\sigma(x) = \sigma'(x)$.

The Coq formalization of Lemma 5.12 is `Lemma n_upd_same` in Appendix B.5.

The main part of the proof that $\overset{[i\mapsto\mathcal{K}]}{\simeq}$ is a CA-bisimulation is performed by a case analysis according to the rules ($\mathrm{LO_{EQ}}$) and ($\mathrm{HI_{EQ}}$) in Table 5.7. The reasoning for the case ($\mathrm{HI_{EQ}}$) is aided by the following lemma (`Lemma high_step` in Appendix B.5), whose statement uses the *UpdNext*$(\_)$ just defined.

Lemma 5.13 *If* $hi^{\mathcal{K}}_{\phi}(\langle S; \sigma\rangle, \mathfrak{P})$, *and* $\vdash_i \langle S; \sigma\rangle \xrightarrow{\alpha} \langle S'; \sigma'\rangle$, *then*

- $\forall c, \rho, \vec{v} : \alpha = c\rho\vec{v} \Rightarrow \mathfrak{P}^{\circ}(c) = H$,
- $\exists \mathfrak{P}^{\bullet} : \alpha \models_{!} \mathfrak{P}^{\bullet}$, *and*
- *if* $\exists \mathfrak{P}^{\bullet} : \alpha \models_{?} \mathfrak{P}^{\bullet}$, *then*

$$\begin{array}{l} \exists \mathfrak{P}' : (\forall x \in \mathbf{Var}_i : (x \in \mathit{UpdNext}(S) \Rightarrow \mathfrak{P}'_{\mathsf{S}}(x) = H) \wedge \\ \qquad\qquad\qquad\qquad (x \notin \mathit{UpdNext}(S) \Rightarrow \mathfrak{P}_{\mathsf{S}}(x) \sqsubseteq \mathfrak{P}'_{\mathsf{S}}(x))) \wedge \\ \qquad \exists \psi : hi^{\mathcal{K}}_{\psi}(\langle S'; \sigma'\rangle, \mathfrak{P}'). \end{array}$$

5.6 Deterministic Schedulers

Up till now, the execution of parallel processes is resolved in a fully nondeterministic manner: at any point in time, any process can execute. Correspondingly, in the previously formulated bisimulations for security, any process can be picked to fire the simulating transition. In practice, the execution is controlled by a scheduler. The security of a system executing freely, does not necessarily imply its security under scheduling. This is actually in line with the fact that information flow security is not usually preserved under refinement — one type of refinement is that of reducing nondeterminacy, and reducing the non-determinacy of a fully-nondeterministic scheduler does not preserve the security of the scheduler-system composite. Starting from this section we study the security of a system under the control of a deterministic scheduler.

We formalize deterministic schedulers Δ for systems Σ as Mealy automata (e.g., [Bab00]) $(Q, q_0, \mathbf{St}_{\Sigma}, 2^{Pid}, \delta, o)$ where Q is its finite set of states, q_0 the

Table 5.8: Transition Rules for Systems under Scheduling

$$\frac{\langle \Sigma;\sigma\rangle \xrightarrow{\alpha}_\eta \langle \Sigma';\sigma'\rangle}{\langle \Sigma;\sigma;q\rangle \xrightarrow{\alpha}_\eta \langle \Sigma';\sigma';\delta(q,\sigma)\rangle} \text{ if } o(q,\sigma)=\{\eta\}$$

$$\frac{\neg(\exists \alpha : \langle \Sigma;\sigma\rangle \xrightarrow{\alpha}_\eta)}{\langle \Sigma;\sigma;q\rangle \xrightarrow{\square}_\eta \langle \Sigma;\sigma;\delta(q,\sigma)\rangle} \text{ if } o(q,\sigma)=\{\eta\}$$

initial state, $\mathbf{St}_\Sigma$ (the memory states of Σ) is the alphabet, 2^{Pid} is the alphabet used for output, $\delta : Q \times \mathbf{St}_\Sigma \to Q$ is the transition function and $o : Q \times \mathbf{St}_\Sigma \to 2^{Pid}$ is the output function. For each state q and memory σ, $o(q,\sigma)$ is a set of one or two process identifiers, signaling the process(es) scheduled for the next step.

The semantics at the system level is adapted to execute the process(es) picked by the scheduler, and the transition rules are given in Table 5.8. The extended configurations $\widehat{C} = \langle \Sigma;\sigma;q\rangle$ now contain the current state q of the scheduler.

It is desirable to constrain the observational power of the scheduler to the parts of the states that are low with respect to all policies. This is achieved with the concept of H-obliviousness. The term is borrowed from [MZ08], but the intuition also underlies other scheduler formalizations such as that in [SS00].

Definition 5.14 (H-Oblivious Schedulers) For a deterministic scheduler $\Delta = (Q, q_0, \mathbf{St}_\Sigma, 2^{Pid}, \delta, o)$, we have $Obl_{\mathcal{P}}(\Delta)$ if $q_0 \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_0$, where $\overset{\Delta}{\underset{\mathcal{P}}{\approx}}$ is the largest relation s.t. $q_1 \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2$ implies

$$\forall \sigma_1,\sigma_2 : \sigma_1 \overset{\{P \mid P \,\mathsf{E}\, \mathcal{P}\}}{=} \sigma_2 \quad \Rightarrow \quad o(q_1,\sigma_1) = o(q_2,\sigma_2) \wedge \delta(q_1,\sigma_1) \overset{\Delta}{\underset{\mathcal{P}}{\approx}} \delta(q_2,\sigma_2).$$

The general necessity of considering a scheduler has been illustrated in Section 2.3.3, with a concurrent program using shared variables. We now give a similar example that is tailored to the setting of the current chapter. This example also justifies the use of $fv(b) \cup Y_1 \cup Y_2$, rather than $Y_1 \cup Y_2$ as the "progress set" in the typing of `if` statements.

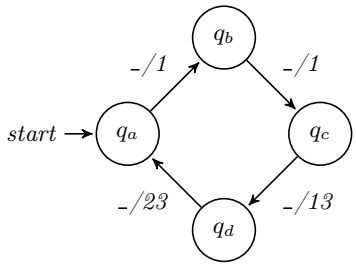
Example 5.8 *Consider the degenerate policy environment $\mathcal{P}_0$ and $\mathcal{P}_0^{\mathrm{ch}}$ such that*

$$\begin{aligned}
\mathcal{P}_0(1) &= \{(h : H, \mathbf{tt})\}\\
\mathcal{P}_0(2) &= \{\}\\
\mathcal{P}_0(3) &= \{(l : L, \mathbf{tt})\}\\
\mathcal{P}_0^{\mathrm{ch}} &= \{(c : L), (c.1 : L, c.1 : \mathbb{Z})\},
\end{aligned}$$

and the following system

$$\Sigma_\star = (1 : \mathsf{if}\ h = 1\ \mathsf{then\ skip; skip\ else\ skip}; c!1 \,||\, 2 : c!2 \,||\, 3 : c?l) \setminus \{c\}.$$

For all scheduled executions reaching a synchronization involving $c?l$, there exists a schedule *under a different evaluation of the conditional $h = 1$, reaching the same synchronization. However, consider a scheduler depicted as follows, where transitions do not depend on observation of the memory state (as signaled by the wildcard used to the left of "/" on each edge), and the lists of process identifiers chosen are specified to the right of the "/" on the edges.*



It turns out that the eventual content of l will again depend on the evaluation of $h = 1$ due to the race between the outputs. In fact this kind of leakage is well understood as "internal timing leaks" [SV98], where memory dependency is exposed via differences in execution times.

The system $\Sigma_\star$ does not type check — for its well-typedness, the presence level of c, as well as the level of l, needs to be lower-bounded by the level of h, which is not the case with $\mathcal{P}_0$ and $\mathcal{P}_0^{\mathrm{ch}}$. It is also not difficult to verify that $Sec_{\mathrm{com}}(\Sigma_\star, \mathcal{P}_0)$ does not hold, due to the strong lockstep requirement imposed by $\alpha \overset{\mathfrak{P}^\bullet}{\underset{\rho}{\sim}} \gamma$ in Figure 5.5. Indeed, we will show in the next section that CA-security implies security under a class of schedulers including the aforementioned one. □

It is not difficult to come up with sensible H-oblivious schedulers for the system Σ_{MD} considered in our motivating example. Simple examples of such schedulers do not base the scheduling decision on the memory at all, as in Example 5.8.

Remark 5.1 *Although the run-time environments of many systems exhibit probabilistic behaviors, the usefulness of deterministic scheduling cannot be underestimated. In many safety-critical systems, such as industrial control systems, avionics systems, etc., the exact scheduling needs to be* predictable *for their functionality and safety to be guaranteed. A differently formulated deterministic scheduler model is also used for concurrent program security in the recent work [MC12].*

5.7 Noninterference under Deterministic Scheduling

In this section, we formulate a noninterference property for systems executing under the control of deterministic schedulers. The property rejects system-scheduler composites with internal timing leaks. We then show that CA-security implies security under every such scheduler whose behavior is only influenced by its observation of the *public* memory (and its internal state).

In the following definition, we extend the notion of "low equivalence" ($\overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}}$) introduced in Section 5.4 to a relation $\overset{\mathfrak{P},\Delta}{\underset{\mathcal{P}}{=}}$ on our extended configurations. In fact, an alternative interpretation of this new relation is $\overset{\mathfrak{P},\Delta}{\underset{\mathcal{P}}{=}} = \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} \times \overset{\Delta}{\underset{\mathcal{P}}{\approx}}$.

Definition 5.15 (Low Eq. of Extended Configurations) $\langle \Sigma_1; \sigma_1; q_1 \rangle \overset{\mathfrak{P},\Delta}{\underset{\mathcal{P}}{=}} \langle \Sigma_2; \sigma_2; q_2 \rangle$ if and only if $\langle \Sigma_1; \sigma_1 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} \langle \Sigma_2; \sigma_2 \rangle$ and $q_1 \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2$.

We are in a position to state the security criterion for scheduled systems. In Definition 5.16, $- \overset{\Delta}{\underset{\mathcal{P}}{\approx}} -$ is the union of all bisimulations on scheduled systems characterized in Definition 5.17, where we omit all transition labels by abbreviating $\widehat{C} \xrightarrow{\alpha}_{\eta} \widehat{C}'$ as $\widehat{C} \longrightarrow \widehat{C}'$. Hence $\widehat{C} \longrightarrow \widehat{C}'$ represents a scheduled step from the extended configuration $\widehat{C}$ to the extended configuration $\widehat{C'}$, that executes a communication action, a τ, or is "blank" because no action from the scheduled process(es) can be performed.

Definition 5.16 (Security of Scheduled Systems) A system Σ is secure under the scheduling of Δ, denoted $Sec^{\Delta}(\Sigma, \mathcal{P})$, if and only if for all σ_1, σ_2, and $\mathfrak{P}$, if $\langle \Sigma; \sigma_1; q_0 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} \langle \Sigma; \sigma_2; q_0 \rangle$, then $(\langle \Sigma; \sigma_1; q_0 \rangle, \mathfrak{P}) \overset{\Delta}{\underset{\mathcal{P}}{\approx}} (\langle \Sigma; \sigma_2; q_0 \rangle, \mathfrak{P})$.

Definition 5.17 (Bisimulation for Scheduled Systems) A bisimulation $R^{\Delta}_{\mathcal{P}}$ for scheduled systems is a symmetric relation such that $(\widehat{C}_1, \mathfrak{P})\ R^{\Delta}_{\mathcal{P}}\ (\widehat{C}_2, \mathfrak{P})$ implies $\widehat{C}_1 \overset{\mathfrak{P},\Delta}{\underset{\mathcal{P}}{=}} \widehat{C}_2$ and the following:

$$\forall \widehat{C}_1' \ s.t.\ \widehat{C}_1 \longrightarrow \widehat{C}_1' : \exists \widehat{C}_2' : \widehat{C}_2 \longrightarrow \widehat{C}_2' \wedge \exists \mathfrak{P}' : (\widehat{C}_1', \mathfrak{P}')\ R^{\Delta}_{\mathcal{P}}\ (\widehat{C}_2', \mathfrak{P}').$$

Our final result (Theorem 5.19) is that the CA-security of a system guarantees its security under the control of any H-oblivious scheduler. However, the bisimulation for scheduled systems defined above dispensed with reliance upon certain channel policies for an input to be simulated; hence we need to impose the following condition of *input completeness* on the set $\mathcal{P}^{\mathrm{ch}}$ of channel policies for the link between the two noninterference conditions to be finally established.

Definition 5.18 (Input-Completeness) $\mathcal{P}^{\text{ch}}$ is input complete for Σ, denoted $IC(\mathcal{P}^{\text{ch}}, \Sigma)$, if the following holds:

$$\forall \sigma, \Sigma', \sigma', c, \vec{v} : \langle \Sigma; \sigma \rangle \rightarrow^* \langle \Sigma'; \sigma' \rangle \xrightarrow{c?\vec{v}} \Rightarrow \exists \mathfrak{P}^\bullet \in \mathcal{P}^{\text{ch}} : c?\vec{v} \models_? \mathfrak{P}^\bullet.$$

This condition essentially requires that each input (from the environment) that can be performed by a system Σ is satisfied by some content policy in $\mathcal{P}^{\text{ch}}$. Note that it is vacuously met by all systems that can only communicate internally (with τ-actions). This notion of input-completeness differs from the more common notions of input-completeness [TLHL11] and input-enabledness [EGP08] in the literature, in that the existing notions are a property of the system, while ours is a property of both the system and, more importantly, the set of channel policies in use.

Theorem 5.19 *For all systems Σ, policy environments $\mathcal{P}$ for variables, and schedulers Δ, if $Sec_{\text{com}}(\Sigma, \mathcal{P})$, $IC(\mathcal{P}^{\text{ch}}, \Sigma)$ and $Obl_{\mathcal{P}}(\Delta)$, then $Sec^{\Delta}(\Sigma, \mathcal{P})$.*

To build up to a proof of Theorem 5.19, we establish the following lemmas.

Lemma 5.20 *For all Σ, σ, Σ_1', σ_1', Σ_2', σ_2', c_1, ρ_1, $\vec{v}_1$, c_2, ρ_2, $\vec{v}_2$, and i, if $\langle \Sigma; \sigma \rangle \xrightarrow{c_1 \rho_1 \vec{v}_1}_i \langle \Sigma_1'; \sigma_1' \rangle$, then*

1. $\langle \Sigma; \sigma \rangle \xrightarrow{c_2 \rho_2 \vec{v}_2}_i \langle \Sigma_2'; \sigma_2' \rangle \Rightarrow c_1 = c_2 \wedge \rho_1 = \rho_2$, *and*
2. $\langle \Sigma; \sigma \rangle \not\xrightarrow{\tau}_i$.

This lemma simply formalizes the observation that there is no intra-process non-determinism with respect to the channel and polarity of actions (although there is with input data).

Lemma 5.21 *For all Σ_1, σ_1, Σ_1', σ_1', Σ_2, σ_2, α', Σ_2', σ_2', η and $\mathfrak{P}$, if we have $(\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P})$, $\langle \Sigma_1; \sigma_1 \rangle \overset{\square}{\twoheadrightarrow}_\eta \langle \Sigma_1'; \sigma_1' \rangle$, and $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2'; \sigma_2' \rangle$, then $\alpha' = \tau$ or $\mathfrak{P}^\circ(ch(\alpha')) = H$.*

Proof. This lemma can be shown using Lemma 5.20 and the symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$. Note that all possible cases for α' are $\alpha' = \tau$, $\mathfrak{P}^\circ(ch(\alpha')) = H$ and $\mathfrak{P}^\circ(ch(\alpha')) = L$. Hence we just need to show that $\mathfrak{P}^\circ(ch(\alpha')) = L$ is impossible. Assume per absurdum that $\mathfrak{P}^\circ(ch(\alpha')) = L$. By symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$ we have

$$(\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P}) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}).$$

By $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2'; \sigma_2' \rangle$, there exist $\mathfrak{P}_!^\bullet$, β' such that $\alpha' \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta'$, and for all $\mathfrak{P}_?^\bullet$, $\vec{v}'$ such that $\alpha' \overset{\mathfrak{P}_?^\bullet}{\underset{?}{\sim}} \beta'(\vec{v}')$, there exist Σ_1', σ_1', and $\mathfrak{P}'$ such that

$$\langle \Sigma_1; \sigma_1 \rangle \overset{\beta'(\vec{v}')}{\twoheadrightarrow}_\eta \langle \Sigma_1'; \sigma_1' \rangle \tag{5.4}$$

$$(\langle \Sigma_2'; \sigma_2' \rangle, \mathfrak{P}') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1'; \sigma_1' \rangle, \mathfrak{P}') \tag{5.5}$$

Since $\mathfrak{P}^\circ(ch(\alpha')) = L$, by $\alpha' \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta'$ we have the following cases:

1. $\alpha' = c!\vec{v}$ and $\beta' = c!\vec{v}''$ for some c, $\vec{v}$ and $\vec{v}''$ such that $v_j = v_j''$ whenever $\mathfrak{P}^\bullet(c.j) = L$,
2. $\alpha' = c?\vec{v}$ and $\beta' = c?[\,]$ for some c and $\vec{v}$.

In both cases, the universally quantified $\mathfrak{P}_?^\bullet$ and $\vec{v}'$ can be instantiated with $\mathfrak{P}_!^\bullet$ and $\vec{v}$, for $\alpha' \overset{\mathfrak{P}_?^\bullet}{\underset{?}{\sim}} \beta'(\vec{v}')$ to be satisfied. We thus obtain (5.4) un-guarded. In more detail, we have

$$\langle \Sigma_1; \sigma_1 \rangle \xrightarrow{\beta'(\vec{v})}_\eta \langle \Sigma_1'; \sigma_1' \rangle \tag{5.6}$$

However, we also have $\langle \Sigma_1; \sigma_1 \rangle \overset{\square}{\twoheadrightarrow}_\eta \langle \Sigma_1'; \sigma_1' \rangle$ from the pre-conditions. This is impossible due to Lemma 5.20, and this contradiction completes the proof. □

The proof of Theorem 5.19 can now be given as follows.

Proof of Theorem 5.19

We rephrase $Sec_{\text{com}}(\Sigma, \mathcal{P})$ as

$$\forall \sigma_1, \sigma_2, \mathfrak{P} \; s.t. \; \langle \Sigma; \sigma_1 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{\equiv}} \langle \Sigma; \sigma_2 \rangle : (\langle \Sigma; \sigma_1 \rangle, \mathfrak{P}) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma; \sigma_2 \rangle, \mathfrak{P}) \tag{5.7}$$

Given Δ such that $Obl_{\mathcal{P}}(\Delta)$ holds, we construct the following relation R_Δ,

$$\begin{aligned} R_\Delta = \; \{ & ((\langle \Sigma_1; \sigma_1; q_1 \rangle, \mathfrak{P}_1), (\langle \Sigma_2; \sigma_2; q_2 \rangle, \mathfrak{P}_2)) \mid \\ & q_1 \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2 \wedge \\ & (\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}_1) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P}_2) \wedge \\ & \exists \sigma : \langle \Sigma; \sigma \rangle \longrightarrow^* \langle \Sigma_1; \sigma_1 \rangle \wedge \exists \sigma : \langle \Sigma; \sigma \rangle \longrightarrow^* \langle \Sigma_2; \sigma_2 \rangle \}. \end{aligned}$$

Using (5.7), it is not difficult to show that for all σ_1, σ_2, and $\mathfrak{P}$ such that $\langle \Sigma; \sigma_1; q_0 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{\equiv}} \langle \Sigma; \sigma_2; q_0 \rangle$, the pair $((\langle \Sigma; \sigma_1; q_0 \rangle, \mathfrak{P}), (\langle \Sigma; \sigma_2; q_0 \rangle, \mathfrak{P}))$ is in R_Δ.

We next show that R_Δ is a bisimulation for scheduled systems. The symmetry of R_Δ is obvious by construction.

Take pair $((\langle \Sigma_1; \sigma_1; q_1 \rangle, \mathfrak{P}), (\langle \Sigma_2; \sigma_2; q_2 \rangle, \mathfrak{P}))$ from R_Δ. We have

$$q_1 \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2 \tag{5.8}$$

$$(\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P}) \tag{5.9}$$

$$\exists \sigma : \langle \Sigma; \sigma \rangle \longrightarrow^* \langle \Sigma_1; \sigma_1 \rangle \tag{5.10}$$

$$\exists \sigma : \langle \Sigma; \sigma \rangle \longrightarrow^* \langle \Sigma_2; \sigma_2 \rangle \tag{5.11}$$

We thus have $\langle \Sigma_1; \sigma_1 \rangle \overset{\mathfrak{P}}{\underset{\mathcal{P}}{=}} \langle \Sigma_2; \sigma_2 \rangle$. Hence $\langle \Sigma_1; \sigma_1; q_1 \rangle \overset{\mathfrak{P}, \Delta}{\underset{\mathcal{P}}{=}} \langle \Sigma_2; \sigma_2; q_2 \rangle$ holds.

Take some arbitrary Σ_1', σ_1' and q_1' such that $\langle \Sigma_1; \sigma_1; q_1 \rangle \longrightarrow \langle \Sigma_1'; \sigma_1'; q_1' \rangle$. By (5.9), we have $\sigma_1 \overset{\{\mathfrak{P}\}}{=} \sigma_2$, thus $\sigma_1 \overset{\{\mathfrak{P}' | \mathfrak{P}' \in \mathcal{P}\}}{=} \sigma_2$. By $Obl_{\mathcal{P}}(\Delta)$, we have $o(q_2, \sigma_2) = o(q_1, \sigma_1) = \{\eta\}$, and $q_1' \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2'$, where $q_2' = \delta(q_2, \sigma_2)$ (and we know that $q_1' = \delta(q_1, \sigma_1)$).

With a case analysis on whether this transition is a $\xrightarrow{\square}$, we show that there is always a one-step simulation.

1. $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ where $\alpha \neq \square$. We have

$$\langle \Sigma_1; \sigma_1 \rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1' \rangle \tag{5.12}$$

   By (5.9), there exist some $\mathfrak{P}_!^\bullet$ and β such that $\alpha \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta$, and for all $\mathfrak{P}_?^\bullet$, $\vec{v}$ such that $\alpha \overset{\mathfrak{P}_?^\bullet}{\underset{?}{\sim}} \beta(\vec{v})$, there exist Σ_2', σ_2', and $\mathfrak{P}'$ such that

$$\langle \Sigma_2; \sigma_2 \rangle \overset{\beta(\vec{v})}{\twoheadrightarrow}_\eta \langle \Sigma_2'; \sigma_2' \rangle \tag{5.13}$$

$$(\langle \Sigma_1'; \sigma_1' \rangle, \mathfrak{P}') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2'; \sigma_2' \rangle, \mathfrak{P}') \tag{5.14}$$

   (a) Suppose α is not an input. Then all $\mathfrak{P}_?^\bullet$ in $\mathcal{P}_\bullet^{\text{ch}}$, and all $\vec{v}$ will satisfy $\alpha \overset{\mathfrak{P}_?^\bullet}{\underset{?}{\sim}} \beta(\vec{v})$.

   (b) Suppose $\alpha = c?\vec{v}_1$ and $\mathfrak{P}^\circ(c) = H$. By $\alpha \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta$, $\beta = \epsilon$. By $IC(\mathcal{P}^{\text{ch}}, \Sigma)$ and 5.10, there exists some $\mathfrak{P}_2^\bullet$ such that for all $\vec{v}_2$, $\alpha \overset{\mathfrak{P}_2^\bullet}{\underset{?}{\sim}} \epsilon(\vec{v}_2)$.

   (c) Suppose $\alpha = c?\vec{v}_1$ and $\mathfrak{P}^\circ(c) = L$. By $\alpha \overset{\mathfrak{P}_!^\bullet}{\underset{!}{\sim}} \beta$, $\beta = c?[\,]$. By $IC(\mathcal{P}^{\text{ch}}, \Sigma)$ and $\exists \sigma : \langle \Sigma; \sigma \rangle \longrightarrow^* \langle \Sigma_1; \sigma_1 \rangle$, there exists some $\mathfrak{P}_2^\bullet$, and $\vec{v}_2 = \vec{v}_1$, such that $\alpha \overset{\mathfrak{P}_2^\bullet}{\underset{?}{\sim}} c?[\,](\vec{v}_2)$.

   Hence in all cases we can instantiate the universally quantified $\mathfrak{P}_?^\bullet$ and $\vec{v}$, and obtain the transition (5.13) such that (5.14) holds. We next make a case split on whether $\beta = \epsilon$.

(a) $\beta \neq \epsilon$.

i. $\beta \neq \square$. We have $\langle \Sigma_2; \sigma_2; q_2 \rangle \longrightarrow_\eta \langle \Sigma_2'; \sigma_2'; q_2' \rangle$. Combining $q_1' \overset{\Delta}{\underset{\mathcal{P}}{\approx}} q_2'$ and (5.14), we have

$$(\langle \Sigma_1'; \sigma_1'; q_1' \rangle, \mathfrak{P}') \; R_\Delta \; (\langle \Sigma_2'; \sigma_2'; q_2' \rangle, \mathfrak{P}').$$

ii. $\beta = \square$. We have $\langle \Sigma_2; \sigma_2 \rangle \overset{\square}{\twoheadrightarrow}_\eta \langle \Sigma_2'; \sigma_2' \rangle$. We know that $(\forall \alpha' : \langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'} \Rightarrow \mathfrak{P}^\circ(ch(\alpha')) = H) \wedge \Sigma_2' = \Sigma_2 \wedge \sigma_2' = \sigma_2$. There are two possibilities concerning the transtion that can be performed from $\langle \Sigma_2; \sigma_2 \rangle$.
Suppose no transition can be performed from $\langle \Sigma_2; \sigma_2 \rangle$. In this case we have $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\square}_\eta \langle \Sigma_2; \sigma_2'; q_2' \rangle$. This transition qualifies as the simulation of $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ in R_Δ, by reasoning similar to the case 1(a)i.
Suppose a transition $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2''; \sigma_2'' \rangle$ exists for some Σ_2'', σ_2'' and α', where α' is a communication such that $\mathfrak{P}^\circ(ch(\alpha')) = H$. By the symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$, (5.14), $\Sigma_2' = \Sigma_2$ and $\sigma_2' = \sigma_2$, we have

$$(\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P}') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1'; \sigma_1' \rangle, \mathfrak{P}'). \tag{5.15}$$

Hence there exist α', β', $\mathfrak{P}^\bullet_{!2}$ such that $\alpha' \overset{\mathfrak{P}^\bullet_{!2}}{\underset{!}{\sim}} \beta'$, and for all $\vec{v}'$ and $\mathfrak{P}^\bullet_{?2}$ such that $\alpha' \overset{\mathfrak{P}^\bullet_{?2}}{\underset{?}{\sim}} \beta'(\vec{v}')$, there exist Σ_1'', σ_1'', and $\mathfrak{P}''$, such that

$$\langle \Sigma_1'; \sigma_1' \rangle \overset{\beta'(\vec{v}')}{\twoheadrightarrow}_\eta \langle \Sigma_1''; \sigma_1'' \rangle \tag{5.16}$$

$$(\langle \Sigma_2''; \sigma_2'' \rangle, \mathfrak{P}'') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1''; \sigma_1'' \rangle, \mathfrak{P}'') \tag{5.17}$$

Note that $\alpha' \overset{\mathfrak{P}^\bullet_{!2}}{\underset{!}{\sim}} \beta'$ indicates that $\beta' = \epsilon$. Hence the universally quantified $\mathfrak{P}^\bullet_{?2}$ can be instantiated by any policy in $\mathcal{P}^{\text{ch}}$ (which is assumed to be non-empty), and $\vec{v}'$ by any value vector of the appropriate length, in order for $\alpha' \overset{\mathfrak{P}^\bullet_{?2}}{\underset{?}{\sim}} \beta'(\vec{v}')$ to hold, thereby obtaining (5.16) and (5.17). By (5.16) and $\beta' = \epsilon$, we have $\Sigma_1'' = \Sigma_1'$ and $\sigma_1'' = \sigma_1'$. By the symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$, we have

$$(\langle \Sigma_1'; \sigma_1' \rangle, \mathfrak{P}'') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2''; \sigma_2'' \rangle, \mathfrak{P}''). \tag{5.18}$$

Building on the transition $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2''; \sigma_2'' \rangle$, we have

$$\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2''; \sigma_2''; q_2' \rangle.$$

It is not difficult to see that the transition above qualifies as a simulation of $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ in R_Δ, since the pair $((\langle \Sigma_1',\ \sigma_1' \rangle, \mathfrak{P}''), (\langle \Sigma_2'',\ \sigma_2'' \rangle, \mathfrak{P}''))$ is in R_Δ.

(b) $\beta = \epsilon$.

i. Suppose $\neg(\exists \alpha' : \langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta)$. Then it is not difficult to see that $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\Box}_\eta \langle \Sigma_2; \sigma_2; q_2' \rangle$ qualifies as a simulation of the transition $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\alpha}_\eta \langle \Sigma_1'; \sigma_1'; q_1' \rangle$.

ii. Suppose there exists the transition

$$\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_\eta \langle \Sigma_2''; \sigma_2'' \rangle, \tag{5.19}$$

for some α', Σ_2'' and σ_2''. We know from $\alpha \overset{\mathfrak{P}^\bullet_!}{\underset{!}{\sim}} \epsilon$ that α is a communication such that $\mathfrak{P}^\circ(ch(\alpha)) = H$. By Lemma 5.21, $\alpha' = \tau$ or $\mathfrak{P}^\circ(ch(\alpha')) = H$.
Suppose $\mathfrak{P}^\circ(ch(\alpha')) = H$. The proof is similar to the case 1(a)ii, where there is a high communication from $\langle \Sigma_2; \sigma_2 \rangle$.
Suppose $\alpha' = \tau$. By symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$, and (5.9), we have

$$(\langle \Sigma_2; \sigma_2 \rangle, \mathfrak{P}) \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}).$$

By (5.19), there exist $\mathfrak{P}^\bullet_{!2}$ and β' such that $\tau \overset{\mathfrak{P}^\bullet_{!2}}{\underset{!}{\sim}} \beta'$, and for all $\mathfrak{P}^\bullet_{?2}$ and $\vec{v}'$ such that $\tau \overset{\mathfrak{P}^\bullet_{?2}}{\underset{?}{\sim}} \beta'(\vec{v}')$, there exist Σ_1'', σ_1'' and $\mathfrak{P}'$, such that

$$\langle \Sigma_1; \sigma_1 \rangle \overset{\beta'(\vec{v}')}{\twoheadrightarrow}_\eta \langle \Sigma_1''; \sigma_1'' \rangle \tag{5.20}$$

$$(\langle \Sigma_2''; \sigma_2'' \rangle, \mathfrak{P}'') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_1''; \sigma_1'' \rangle, \mathfrak{P}'') \tag{5.21}$$

By $\tau \overset{\mathfrak{P}^\bullet_{!2}}{\underset{!}{\sim}} \beta'$, we have $\beta' = \tau$ or $\beta' = \Box$. The universally quantified $\mathfrak{P}^\bullet_{?2}$ can be instantiated with any policy in the non-empty $\mathcal{P}^{\text{ch}}$, and $\vec{v}'$ with any value vector of the appropriate length, for $\tau \overset{\mathfrak{P}^\bullet_{?2}}{\underset{?}{\sim}} \beta'(\vec{v}')$ to be satisfied. We thus obtain (5.20) and (5.21). By $\beta = \epsilon$, and $\alpha \overset{\mathfrak{P}^\bullet}{\underset{!}{\sim}} \beta$, α is a communication such that $\mathfrak{P}^\circ(ch(\alpha)) = H$. Hence by Lemma 5.20, and (5.20), β' cannot be τ. Hence $\beta' = \Box$, and $\Sigma_1'' = \Sigma_1$ and $\sigma_1'' = \sigma_1$.
By symmetry of $- \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} -$, and (5.21), we have

$$(\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}'') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2''; \sigma_2'' \rangle, \mathfrak{P}'').$$

By transition (5.12), there exist $\mathfrak{P}^{\bullet}_{!3}$ and β'' such that $\alpha \overset{\mathfrak{P}^{\bullet}_{!3}}{\underset{!}{\sim}} \beta''$, and for all $\mathfrak{P}^{\bullet}_{?3}$ and $\vec{v}''$ satisfying $\alpha \overset{\mathfrak{P}^{\bullet}_{?3}}{\underset{?}{\sim}} \beta''(\vec{v}'')$, there exist Σ_2''', σ_2''' and $\mathfrak{P}'''$ such that

$$\langle \Sigma_2''; \sigma_2'' \rangle \overset{\beta''(\vec{v}'')}{\twoheadrightarrow}_{\eta} \langle \Sigma_2'''; \sigma_2''' \rangle \tag{5.22}$$

$$(\langle \Sigma_1'; \sigma_1' \rangle, \mathfrak{P}''') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2'''; \sigma_2''' \rangle, \mathfrak{P}''') \tag{5.23}$$

Since α is a high presence communication, $\beta'' = \epsilon$, and $\Sigma_2''' = \Sigma_2''$ and $\sigma_2''' = \sigma_2''$. Hence the transition $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\alpha}_{\eta} \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ can be simulated by $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\tau}_{\eta} \langle \Sigma_2''; \sigma_2''; q_2' \rangle$ in R_Δ, resulting in $((\langle \Sigma_1'; \sigma_1'; q_1' \rangle, \mathfrak{P}'''), (\langle \Sigma_2''; \sigma_2''; q_2' \rangle, \mathfrak{P}''')) \in R_\Delta$.

2. $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\square}_{\eta} \langle \Sigma_1'; \sigma_1'; q_1' \rangle$. We have $\Sigma_1' = \Sigma_1$, $\sigma_1' = \sigma_1$, and

$$\neg(\exists \alpha' : \langle \Sigma_1; \sigma_1 \rangle \xrightarrow{\alpha'}). \tag{5.24}$$

Compared with case 1, we argue more briefly, since no new intuition is needed. A case split is made on the transition that can happen from $\langle \Sigma_2; \sigma_2 \rangle$, over the processes in η. By (5.24) we have $\langle \Sigma_1; \sigma_1 \rangle \overset{\square}{\twoheadrightarrow}_{\eta} \langle \Sigma_1'; \sigma_1' \rangle$. By Lemma 5.21, the only possible cases are the following.

(a) There is transition $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'}_{\eta} \langle \Sigma_2'; \sigma_2' \rangle$ for some Σ_2', σ_2' and α' such that $\mathfrak{P}^{\circ}(ch(\alpha')) = H$. It can be deduced that $(\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2'; \sigma_2' \rangle, \mathfrak{P}')$ for some $\mathfrak{P}'$. Hence the transition $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\square}_{\eta} \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ can be simulated by $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\alpha'}_{\eta} \langle \Sigma_2'; \sigma_2'; q_2' \rangle$ in R_Δ.

(b) There is transition $\langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\tau}_{\eta} \langle \Sigma_2'; \sigma_2' \rangle$ for some Σ_2' and σ_2'. We can still deduce $(\langle \Sigma_1; \sigma_1 \rangle, \mathfrak{P}') \overset{\text{com}}{\underset{\mathcal{P}}{\sim}} (\langle \Sigma_2'; \sigma_2' \rangle, \mathfrak{P}')$ for some $\mathfrak{P}'$. The simulation in R_Δ is by the transition $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\alpha'}_{\eta} \langle \Sigma_2'; \sigma_2'; q_2' \rangle$.

(c) $\neg(\exists \alpha' : \langle \Sigma_2; \sigma_2 \rangle \xrightarrow{\alpha'})$. The transition $\langle \Sigma_1; \sigma_1; q_1 \rangle \xrightarrow{\square}_{\eta} \langle \Sigma_1'; \sigma_1'; q_1' \rangle$ (note that $\Sigma_1' = \Sigma_1$ and $\sigma_1' = \sigma_1$) can be simulated by $\langle \Sigma_2; \sigma_2; q_2 \rangle \xrightarrow{\square}_{\eta} \langle \Sigma_2; \sigma_2; q_2' \rangle$, resulting in $((\langle \Sigma_1; \sigma_1; q_1' \rangle, \mathfrak{P}), (\langle \Sigma_2; \sigma_2; q_2' \rangle, \mathfrak{P})) \in R_\Delta$.

The cases above are exhaustive and the proof is complete. □

The Σ_{MD} of our motivating example only communicates internally. Hence continuing with Example 5.7, we can finally conclude $Sec^{\Delta}(\Sigma_{\text{MD}}, \mathcal{P}_{\text{MD}})$ for all schedulers Δ such that $Obl_{\mathcal{P}_{\text{MD}}}(\Delta)$ holds.

5.8 Developing Proofs in Coq

In this section, we elaborate on certain key elements of our Coq formalization of the theory presented in this Chapter. In doing so, we make the complete formalization given in Appendix B accessible to the reader.

The Coq Proof Assistant

We briefly introduce the reader to the Coq proof assistant [CPA]. This is an interactive theorem prover based on type theory. The ability to use dependent types facilitates the formalization of programming language theory [Chl13]. The Curry-Howard correspondence (e.g., [Pie02]) that bridges logical inference and type inference is relied on in making and verifying formal logical statements. We elaborate slightly on this last statement with an example. For logical inference, we have the Modus ponens rule below on the left. For type inference in functional programs, on the other hand, we have the rule for function applications on the right.

$$\frac{A \to B \quad A}{B} \qquad \frac{f : A \to B \quad a : A}{f(a) : B}$$

The usual reading of the second rule is: if the term (function) f has the arrow type $A \to B$, and the term a has type A, then the term $f(a)$ has type B. But an alternative reading can be: if f *is an evidence of* the claim "if A then B", and a *is an evidence of* A, then $f(a)$ *is an evidence of* B. The latter can thus be seen as an evidence-carrying version of the modus ponens rule. In fact, proving in Coq boils to a great extent down to applying "tactics" that help create "proof terms" (such as the $f(a)$ above).

Theory aside, understanding the formulations to be presented in this section corresponds mostly to understanding the types (such as the $A \to B$, A and B in the example above), thus to understanding logical formulas. Note that in Coq formalizations, when we have A_1, A_2, ..., A_n ($n \geq 2$), and B, it is customary to write $A_1 \to A_2 \to ... \to A_n \to B$, for $A_1 \wedge A_2 \wedge ... \wedge A_n \to B$. This can be understood intuitively by noting that $\to$ is right-associative, and thus $A_1 \to A_2 \to ... \to A_n \to B$ is equivalent to $A_1 \wedge A_2 \wedge ... \wedge A_n \to B$ when both are seen as logical formulas. In the terms of functional programming, $A_1 \to A_2 \to ... \to A_n \to B$ works by *currying* [Pie02].

Outline of the Proof

We define mappings with *explicit domains* as follows.

```
1 Section Mappings.
2
3 Variable A: Type.
4 Definition Dec (A:Type) := ∀ a1 a2: A, {a1=a2} + {a1<>a2}.
```

```
5  Variable dec: Dec A.
6
7  Record Mapping (A B:Type) (dec_A:Dec A) :=
8    mkMapping {
9        il: list A;
10       mp: A→ B
11   }.
12 ...
13 End Mappings.
14
15 Notation "## A" := (il _ _ _ A) (at level 10).
16 Notation "&& A" := (mp _ _ _ A) (at level 10).
```

We define a mapping as a record having two fields (lines 7-11): `il` is its domain represented by a list, and `mp` a function keeping record of which element in the domain is mapped to which element in the range. The `Dec A` on line 4 says that equality of type `A` should be decidable. An evidence of the decidability of equality on the domain is needed to create a mapping, since the comparison of domain elements is unavoidable when using it. As suggested by lines 15 and 16, notations are defined for referencing the `il` field and `mp` field, respectively, of a mapping `A`. The wildcards "`_`" are used for types that can be inferred by the Coq system.

The disjoint union between two mappings is then defined as follows.

```
1  Definition m_union {B: Type} (m1 m2: Mapping A B dec) :
2    option (Mapping A B dec) :=
3    if (common _ dec (il A B _ m1) (il A B _ m2)) then
4      None
5    else
6      Some (mkMapping A B dec
7                      ((il A B _ m1) ++ (il A B _ m2))
8                      (u_map (il A B _ m1) (mp A B _ m1) (mp A B _ m2))
9           ).
```

We make use of optional datatypes of Gallina (the specification language of Coq), and the disjoint union is `None` when the domains of both mappings have *common elements*. When the two mappings have disjoint domains, we then combine their domains (line 7) and their bodies (line 8). The function `u_map` essentially produces the function that maps all elements in `(il A B _m1)` according to `(mp A B _m1)`, and maps all elements elsewhere according to `(mp A B _m2)`.

The setting of Section 5.2 is one in which the process-local memories are of the same type as that of global memories. Somewhat different than this, in our Coq formalization, global memories are mappings from process identifiers to process-local memories. Process-local memories are in turn mappings from process-local variables to their values.

```
Definition Mem := Mapping id nat eq_id_dec.
Definition SysMem := Mapping pid Mem eq_pid_dec.
```

This approach is adopted because keeping the boundaries between different processes explicit makes subsequent formalization and proof tasks more convenient. On the other hand, for a concise presentation on paper, the homogeneity of local and global memories is more desirable. It is expected that this difference will not render the Coq formalization unfaithful to the technical development since both memory models correspond to the view of different processes having disjoint local memories.

We define actions that can be τ's, outputs, inputs, ϵ's or $\Box$'s, as follows.

```
Inductive action : Type :=
  | Act_LTau : action
  | Act_LOt : pch → list nat → action
  | Act_LIn : pch → list nat → action
  | Act_Eps : action                (* ϵ *)
  | Act_Suspd : action              (* □ *)
.
```

Related to the formalization of memories, the semantic rule for synchronous communication in systems is exhibited next.

```
1  Inductive sys_step : conf → list pid → action → conf → Prop :=
2    ...
3    | Sys_Step_Sync :
4        ∀ sys1 sys1' sys2 sys2' m1 m1' m2 m2'
5              i j c vl alpha1 alpha2 sysm,
6          sys1//m1 ===> [[i], alpha1] sys1'//m1' →
7          sys2//m2 ===> [[j], alpha2] sys2'//m2' →
8          (common _ eq_pid_dec (##m1) (##m2) = false) →
9          eq_sysmem sysm (u_sysmem m1 m2) →
10         (alpha1 = Act_LOt c vl ∧ alpha2 = Act_LIn c vl ∨
11          alpha1 = Act_LIn c vl ∧ alpha2 = Act_LOt c vl) →
12         (sys1 |P| sys2)//sysm ===> [[i;j], Act_LTau]
13         (sys1' |P| sys2')//( upd_sysmem (upd_sysmem sysm i m1') j m2')
14   ...
15   where "sys '//' m '===>' '[' pidlst ',' alpha ']' sys' '//' m'" :=
16         (sys_step (sys,m) pidlst alpha (sys', m')) : sem_scope.
```

Line 1 says that a transition step of the system is a judgement (represented by the type `Prop` of Gallina, for propositions) on two configurations, a list of process identifiers, and an action. A piece of notation has been introduced such that a transition of the form $\langle \Sigma; \sigma \rangle \xrightarrow{\alpha}_{\eta} \langle \Sigma'; \sigma' \rangle$ is represented as `sys//m ===> [pidlst, alpha] sys'//m'`. Here `sys` corresponds to Σ, `m` to σ, `pidlst` to η, `alpha` to α, `sys'` to Σ' and `m'` to σ'. In addition, `sys1 |P| sys2` represents the parallel composition of `sys1` and `sys2`.

We explicitly require that the domains of the memories used by the two systems in parallel (i.e., the sets of process identifiers used in the two systems) should be disjoint. This reflects the "background" assumption on the well-formedness of systems. It is also worth mentioning that we use the self-made binary relation `eq_sysmem`, rather than Coq's built-in equality `=`, to express that two system memories are the same (e.g., line 9 of the rule `Sys_Step_Sync`). In prose, the requirements of `eq_sysmem` are

1. two memories should have the same domain, and
2. they map each element *in the domain* to the same value.

We move on to discuss the setup for our information flow type system.

```
1  Definition Assertion    := Mem → Prop.
2  Definition AsnEnv       := Mapping pid Assertion eq_pid_dec.
3  Definition PVarVal      := Assertion.
4  Definition PVarSec      := Mapping id SecLev eq_id_dec.
5  Definition PVar         := prod PVarSec PVarVal.
6  Definition PVarEnv      := Mapping pid (Ensemble PVar) eq_pid_dec.
7  Definition PChP         := pch → SecLev.
8  Definition PChCVal      := ch → (Ensemble nat).
9  Definition PChCSec      := ch → SecLev.
10 Definition PChC         := prod PChCSec PChCVal.
11 Definition PChEnv       := sig (fun (CE: Ensemble PChC) =>
12                         ∃ PcC, PcC from CE).
```

Our Hoare logic assertions are of the type `Mem -> Prop`, i.e., they are functions for process-local memories to propositions. This follows the approach taken in [PCG+]. An assertion environment is a mapping from process identifiers to process-local assertions. Confidentiality policies for memories have the product type `prod PVarSec PVarVal`, where `PVarSec` is the type for their confidentiality components and `PVarVal` the type for their value components. The latter type is but a synonym of the type for assertions. We then formalize a policy environment ($\mathcal{P}$ in the technical development) as a mapping from process identifiers to sets of process-local variable policies (line 6). Concerning channels, the distinguished policy on "presence" has the arrow type from polyadic channels (`pch`) to confidentiality levels (`SecLev`). On the other hand, each content policy has the product type `prod PChCSec PChCVal`, where the component types `PChCSec` and `PChCVal` are self-explanatory. In our technical development, the set $\mathcal{P}^{\text{ch}}_{\bullet}$ of content policies for channels is required to be non-empty. This non-emptiness constraint is imposed using subset types [BC04] (or refinement types). In more detail, a channel policy environment is a set of content policies `Ensemble PChC` such that there exists a content policy in it (`exists PcC, PcC from CE`).

We define two global variables, `PcP` of type `PChP` and `PChCSet` of type `PChEnv`. This allows us to omit the distinguished presence policy and the set of content

policies for channels as the parameters of various definitions that follow.

```
Variable PcP : PChP.
Variable PChCSet : PChEnv.
```

Analogous to the two-level structure in process-local memories and system memories, for variable policies we introduce the system-level. A system-level variable policy (also called a composed policy hereafter) has the product type `prod CompPSec CompPVal`. Here `CompPSec` is the type of the confidentiality component, which is a mapping from process identifiers to process-local confidentiality policies. On the other hand, `CompPVal` is the type of the value component, which is a mapping from process identifiers to process-local assertions.

```
Definition CompPSec := Mapping pid PVarSec eq_pid_dec.
Definition CompPVal := AsnEnv.
Definition CompP    := prod CompPSec CompPVal.
```

Then the fact $\mathfrak{P} \Vdash \mathcal{P}$, that a variable policy is composed according to the policy environment $\mathcal{P}$, is expressed as:

```
Definition comp_from (CP: CompP) (PE: PVarEnv) :=
  (##(fst CP) = ##PE) ∧ (##(snd CP) = ##PE) ∧
  (∀ i, List.In i (##PE) →
        ∃ P, (P from (&&PE i) ∧ pair (&&(fst CP) i) (&&(snd CP) i) = P)).
```

For `comp_from CP PE` to hold, both components of the system-level variable policy `CP` should have the same domain as the policy environment `PE`. More importantly, for each process identifier `i` in the domain of `PE`, there should exist a variable policy `P` from the set of all variable policies for process `i`, such that the confidentiality component of `CP` maps `i` to the confidentiality component of `P`, and the value component of `CP` maps `i` to the value component of `P`.

For the type system at the process-level, the notation `K,i,X,l \- phi,S,psi -: X',l'` is introduced to represent typing judgements. Compared with the $X, l \vdash_{\mathcal{K}} \{\phi\}\, S\, \{\psi\} : X', l'$ used in the presentation on paper, the only small difference is that the process identifier `i` of `S` is made explicit.

We elaborate on the rule for input — this rule is the lengthiest in all the formalized typing rules.

```
1  Inductive well_typed (K: Ensemble PVar) (i: pid) :
2    (Ensemble id) → SecLev → Assertion → com → Assertion →
3    (Ensemble id) → SecLev → Prop :=
4    ...
5    | TP_IN : ∀ X l phi xl c l',
6                contained_in X (pid_ids i) →
7                ((Ordr l (PcP c)) ∧ (Ordr (PcP c) l') ∧
8                 (∀ P m, ##m = pid_ids i →
```

```
9                 P from K → (snd P m ∧ all_assn xl phi m) →
10                ((∀ l, sLift P X l → Ordr l (PcP c)) ∧
11                 ∀ PcC vl, PcC from (proj1_sig PChCSet) →
12                           |vl| = |xl| → vl_sat vl c PcC →
13                 ∃ Q, Q from K ∧
14                   (∀ x', x' ido P ∧ ~(List.In x' xl) →
15                        (Ordr (secL P x') (secL Q x'))) ∧
16                   (∀ j, 1<=j<=|xl| →
17                          (Ordr (lub (fst PcC (Chan c j)) (PcP c))
18                                (secL Q (xl x@ (j−1))))) ∧
19                   (val_sat_sub_n m Q xl (num vl))
20                )
21               )
22              ) →
23              K, i, X, l \− (all_assn xl phi), (c ?? xl), phi −:
24                            (Empty_set _), l'
25  ...
26 where " K ',' i ',' X ',' l '\-' phi ',' S ',' psi '-:' X' ',' l' "
27       := (well_typed K i X l phi S psi X' l') : hoare_spec_scope.
```

We re-display the typing rule for input in Table 5.6 below, for ease of comparison.

$$X, l \vdash_{\mathcal{K}} \{\forall \vec{x} : \phi\}\, c?\vec{x}\, \{\phi\} : \emptyset, l' \quad \text{if } l \sqsubseteq \mathfrak{P}^{\circ}(c) \sqsubseteq l' \text{ and}$$
$$\forall \mathfrak{P} \in \mathcal{K} : \mathfrak{P}_{\mathsf{V}} \wedge (\forall \vec{x} : \phi) \Rightarrow (\mathfrak{P}_{\mathsf{S}}[X] \sqsubseteq \mathfrak{P}^{\circ}(c) \wedge\ \forall \mathfrak{P}^{\bullet} \in \mathcal{P}^{\mathrm{ch}} : \forall \vec{v}\ s.t. \bigwedge_j v_j \in \mathfrak{P}^{\bullet}(c.j) :$$
$$\exists \mathfrak{P}' \in \mathcal{K} : \mathfrak{P}[(x_j \mapsto \mathfrak{P}^{\bullet}_{\mathsf{S}}(c.j) \sqcup \mathfrak{P}^{\circ}(c))_j]_{\mathsf{S}} \preceq \mathfrak{P}'[(\underline{v_j}/x_j)_j]_{\mathsf{V}})$$

The reason for the Coq formalization to be lengthier is twofold. First of all, a few "healthiness constraints" are made explicit (the grey parts in lines 6, 8 and 12). The first such constraint says that each variable in `X` is an identifier local to process `i`. The second such constraint says that only local memories having the set of identifiers of process `i` as their domains are considered as local states. The third such constraint says that only value vectors having the same number of components as $\vec{x}$ does are considered as proper input values. Second of all, the condensed formulation $\mathfrak{P}[(x_j \mapsto \mathfrak{P}^{\bullet}_{\mathsf{S}}(c.j) \sqcup \mathfrak{P}^{\circ}(c))_j]_{\mathsf{S}} \preceq \mathfrak{P}'[(v_j/x_j)_j]_{\mathsf{V}})$ is spelled out into the three separate constraints in lines 14 to 19.

We dispense with explaining all the user-defined notations needed to completely understand the rule `TP_IN`, but point out the following facts. Starting from line 14, the first constraint expresses $\forall x', (x' \in \mathbf{D}_{\mathfrak{P}_{\mathsf{S}}} \wedge x' \notin \{\vec{x}\}) \Rightarrow \mathfrak{P}_{\mathsf{S}}(x') \sqsubseteq \mathfrak{P}'_{\mathsf{S}}(x')$. The second constraint expresses $\forall j\ s.t.\ 1 \leq j \leq |\vec{x}| : \mathfrak{P}^{\bullet}_{\mathsf{S}}(c.j) \sqcup \mathfrak{P}^{\circ}(c) \sqsubseteq \mathfrak{P}'(x_j)$. The third constraint says that $\mathfrak{P}'[(\underline{v_j}/x_j)_j]_{\mathsf{V}}$ should hold on the process-local memory m assumed from the beginning, satisfying $\mathfrak{P}_{\mathsf{V}}$ and $\forall \vec{x} : \phi$ (line 9).

Unlike its presentation on paper, the formalized rule `TP_IN` uses process-local memories `m` explicitly. This stated-oriented approach to the formalization of Hoare logic has previously been taken in [PCG+]. The logic and theory

supported by the assertions are essentially "borrowed" from Gallina, and there is no need for the user to provide extra machinery in their interpretation.

The typing rules for systems are no more perplexing; hence we omit their explanation and direct our attention now to the formalization of CA-security.

The satisfaction of an assertion environment by a system memory ($\sigma \models \Phi$ introduced in the beginning of Section 5.5.3) has the following, self-explanatory formalization.

```
Definition comp_sat (m: SysMem) (AE: AsnEnv) :=
  (##m = ##AE) ∧ (∀ i, List.In i (##m) → (&&AE i (&&m i))).
```

The low-equivalence relation on states in Figure 5.5 is then formulated as follows.

```
1  Definition state_eq (P: CompP) (PE: PVarEnv) sys1 m1 sys2 m2 :=
2    comp_from P PE ∧
3    get_pids sys1 = ##PE ∧ get_pids sys2 = ##PE ∧
4    ##(fst P) = ##(snd P) ∧
5    comp_sat m1 (snd P) ∧ comp_sat m2 (snd P) ∧
6    ∀ i, List.In i (##PE) →
7              ##(&&(fst P) i) = pid_ids i ∧
8              ##(&&m1 i) = pid_ids i ∧ ##(&&m2 i) = pid_ids i ∧
9              ∀ x, List.In x (pid_ids i) →
10                      (&&(&&(fst P) i) x = L) →
11                      &&(&&m1 i) x = &&(&&m2 i) x.
```

The parts in grey are again healthiness constraints on the domains of certain mappings, that are taken for granted in the pen-and-paper formulation. Since the value component of a composed policy essentially has the type `Asn_Env`, just like the pre-conditions and post-conditions in the typing rules for systems, the newly introduced `comp_sat` can be used to express the satisfaction of composed policies by memory states (line 5). The core of `state_eq` is the part saying that low variables have the same value in both memories, spanning lines 9-11. Due to the two-level structure used in our memories and policies, double "dereferencing" of system memories and composed policies are needed. Essentially, we focus on the process `i` first (line 6), and then on each variable `x` local to that process (line 9).

Action schemas have the following formalization, which is similar to that of actions. Note that it takes only a polyadic channel for an input action schema to be formed.

```
Inductive action_sk : Type :=
  | ActSk_LTau : action_sk
  | ActSk_LOt : pch → list nat → action_sk
  | ActSk_LIn : pch → action_sk
  | ActSk_Eps : action_sk
```

```
  | ActSk_Suspd : action_sk
.
```

Since actions and action schemas have different types, it becomes convenient to make separate formalizations of low equivalence between two actions, and low equivalence between an action and an action schema. The two versions (called `act_eq` and `act_sk_eq`, in `Sec.v`) parallel each other, but make use of different utility functions due to the difference in type. We elide the details of the two definitions here and Appendix B can be consulted for a clear view.

We are now in a position to exhibit the formulation of CA-bisimulation. The elements in the following snippet correspond to the pen-and-paper formulation exactly, although lengthier notations are used.

```
Program Definition ca_bisim (PE: PVarEnv) (R: relation (prod conf CompP))
:=
  symmetric _ R ∧
  ∀ C1 C2 CP,
    R (pair C1 CP) (pair C2 CP) →
    state_eq CP PE (fst C1) (snd C1) (fst C2) (snd C2) ∧
    ∀ alpha pidlst C1',
      sys_step C1 pidlst alpha C1' →
      ∃ PcCOt bta,
        PcCOt from PChCSet ∧ act_sk_eq alpha bta PcCOt Pol_Ot ∧
        ∀ PcCIn vl,
          PcCIn from PChCSet →
          act_eq alpha (fill bta vl) PcCIn Pol_In →
          ∃ C2' CP',
            may_step C2 pidlst (fill bta vl) C2' ∧
            R (pair C1' CP') (pair C2' CP').
```

The `PChCSet` used in the definition above has type `PChEnv`, which is declared to be a subset type, indicating that `PChCSet` is a set and that it is non-empty. To express that a policy `PcCOt` is a member of the *underlying set* of `PChCSet`, we would need to write `PcCOt from (proj1_sig PChCSet)` [BC04]. The keyword `Program` has been used to enable us to write the simpler `PcCOt from PChCSet` instead — the `Program` mechanism of Coq allows one to deal with subset types as their underlying types.

We now end our description with the formalization of CA-security. There is again a precise correspondence with the pen-and-paper formulation, which renders verbal explanation unnecessary.

```
Definition ca_sec sys PE :=
  ∀ sysm1 sysm2 CP,
    state_eq CP PE sys sysm1 sys sysm2 →
    ∃ R, ca_bisim PE R ∧
        R (pair (pair sys sysm1) CP) (pair (pair sys sysm2) CP).
```

Level of Automation

It is advocated in [Chl13] that proof automation should be pushed to the extreme. This is largely done via user-defined tactics — analogous to programming automated theorem provers inside Coq. On the other hand, several existing proof developments [BBJ13, MPP13, CHRS14] in Coq are performed mostly through manual application of mostly elementary tactics.

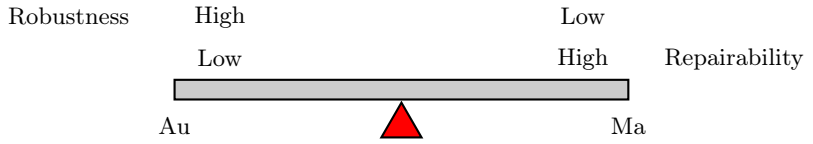


Figure 5.9: The Respective Merits of Automated and Manual Proofs

When developing proofs in a proof assistant, the workflow rarely goes linearly from formulation to proving. This is because small deficiencies in the formulation can often be spotted during the proof process. Hence cycling through the two tasks is often unavoidable. Taking into consideration the changes that are often needed in the formulation, and the possibility that existing proofs can be broken, we summarize the respective merits of crafting tactics for automation and manually applying elementary tactics in Figure 5.9.

In short, the automated style of proof tends to be more *robust* under formulational adjustments. On the other hand, once a highly automated proof is broken, it is more difficult to locate the problem and repair it. On the contrary, proofs composed of tactics involving less automation tend to be more easily broken, but more easily adapted/repaired at the same time.

Apart from formulational changes, another relevant factor is the "purpose" of proofs. One typical purpose is program verification (e.g., [BJSB11]), where automation is often desirable. Since the problem domain is relatively narrow and well-defined (verifying method contracts, etc.), extensive automation can be feasible. Another typical purpose is verifying the theoretical results of research papers (e.g., [BBJ13, MPP13]), where the level of automation cannot be pushed as high. This is because the results are typically "deeper" (in the extreme, consider the four-color theorem or the odd order theorem whose proofs have been completed in Coq [Gon07, GAA+13] within the last decade), which calls for human intuition to be exercised. In addition, the reasoning involved can be drastically different from paper to paper, which makes it difficult to reuse automation.

Benefits and Perils of Using a Proof Assistant

Apart from the obvious effect of obtaining higher correctness assurance, some convenience is in fact gained by using a proof assistant. Since the names of variables, as well as the effective hypotheses, are maintained automatically at any point of the proving process, naming clashes cease to be a worry, and there is no need to look back on the part of the proof already written to gather information on what is known (or to keep this in mind). Another advantage is related to changes of formulation, under which it is often difficult to manually re-certify the correctness of a modified proof, while re-certification accomplished in a proof assistant results immediately in great confidence in the new proof. This is especially (comparably) effective when the changes needed in the old proof are in fact not many, but it is difficult or/and time-consuming to see with the human eye that no more changes are required.

One of the perils in performing formalization in a proof assistant is simply the mis-understanding of the associativity between connectives. For instance $\forall \mathtt{x} : \mathtt{A} \to \mathtt{B} \wedge \forall \mathtt{y} : \mathtt{C} \to \mathtt{D}$ could be written where $(\forall \mathtt{x} : \mathtt{A} \to \mathtt{B}) \wedge (\forall \mathtt{y} : \mathtt{C} \to \mathtt{D})$ is actually intended. This is more likely to happen when the formulas involved are large. It might not be difficult to spot this issue when performing proofs related to this formulation: `inversion H` where `H` is of the form $\forall \mathtt{x} : \mathtt{A} \to \mathtt{B} \wedge \forall \mathtt{y} : \mathtt{C} \to \mathtt{D}$, will not work, whereas the parenthesized version can be successfully broken into the two conjuncts. Besides, certain details can render the development process time-consuming. Such details are often below the abstraction level of the formulations presented on paper, and have little to do with intuition, but are nevertheless indispensable for mechanized certification. Since deviation from the paper-formulation also adds to the difficulty in understanding, keeping such low-level machinery minimal in the formalization is desirable.

5.9 Related Work

Broberg and Sands [BS06, BS10] enforce information flow policies that depend on the states of "locks". A basic example is the policy that only allows a bidder to view others' bids after the auction has been completed (signaled, e.g., by the fact that all the bidders have submitted their bids). Here whether a bidder has submitted a bid can be captured by the state of a "lock". The approach in [BS10], in particular, allows parameterized locks to be used, with the typical purpose of expressing that an actor assumes a role. An encoding of the Decentralized Label Model [ML97] is performed in the label model of this latter work, where parameterized locks are used to encode the principal hierarchy.

Amtoft et al. [ADZ+12] employ a relational Hoare logic for the expression of content-dependent information flow policies. An example policy that can be expressed in their framework is "for the values of variable y to be equal at

program point ℓ', the values of x_1 should be equal at program point ℓ under condition ϕ_1, while the values of x_2 should be equal at program point ℓ under condition ϕ_2". Their language corresponds to the sequential While language augmented with for-loops and arrays.

Lourenço and Caires [LC13, LC15] develop dependent information flow types that also allow for content-dependency. In [LC13], they work with a core functional language extended with database primitives. The basic goal is to use security labels that apply not for whole relations (tables), but for individual records. Suppose Pt denotes *readership* of patients. Then the label $Pt(j)$ means that information can only flow to a particular patient, which corresponds to a record whose identifier is j. The subsequent work [LC15] generalizes this to support a wide class of dependencies of security labels on run-time data values, in a general purpose λ-calculus.

The early work [AR80] by Andrews and Reitman discusses a program logic (bearing resemblance to Hoare logic) that enforces information flow policies depending on the security classes and values of variables; however, their analysis is not justified by a semantic property of security.

Barthe et al [BFG+14] develop relational refinement types for a higher-order functional language, thereby gaining the ability to verify a broad range of safety/security properties, including termination-insensitive information flow security. In more detail, the combination of data type a and security class *public* can be expressed as *type eq* $a = x : a$ $\{\!|L\, x = R\, x|\!\}$, with the literal interpretation "x has type a, and is subject to the constraint that it has the same value in a related pair of executions". Their relational types take the flavor of the relation Hoare logic of [ADZ+12], and refinement types can be understood as a fragment of dependent types (such as those of [LC15]) amenable to automation; hence it is likely that their technique can apply to content-dependent information flow, although no example is provided in this direction.

Our CA-bisimulation does not universally quantify over states, but rather carries the states along with the executions. This has the flavor of the "flat bisimulation" considered in [Dam06] that further goes back to [BC02] in the setting of shared memory. "Flat bisimulations" do not give rise to a compositional notion of security when shared-memory is used, since memory modifications by other concurrent processes are not captured. Mantel et al recovers compositionality in the same setting by requiring the bisimulation relation to be "closed under globally consistent changes" [MSS11].

Known formalizations of information flow type systems include Barthe and Nieto's formalization [BN04] of Boudol and Castellani's work [BC02] in Isabelle [IPA], a type system for Java bytecode by Barthe et al [BPR07] formalized in Coq, and a series of other formalizations [GLMS14a, GLMS14b, GMS14] by Grewe et al, performed in Isabelle.

Chapter 6

Conclusion

We have demonstrated that noninterference properties and information flow analyses can support *fine-grained policies*, for *concurrent, communicating systems*.

First of all, the *presence* of communication actions and their *content* can be treated as separate dimensions. Moreover, all combinations of sensitivity levels in the two dimensions are practically meaningful *for integrity*. These combinations are interpreted using a notion of δ-security where δ is the strategy of the environment. When no mixed combination is in use, δ_{ALL}-security degenerates to the classical condition SBNDC, where δ_{ALL} is the most powerful (aggressive) environment. A compositionality result is also given about δ_{ALL}-security, to facilitate structural security analysis of concurrent systems.

Second of all, the security levels of variables and channels can be adjusted according to different *contents* of the memory and communications. A compositional notion of CA-security is proposed, taking into consideration both the presence and content of communications and content-dependent policies. It is shown that the CA-security of a system implies its security under deterministic schedulers whose behaviors can only be affected by their public observation. A security type system sound with respect to CA-security is devised, incorporating a Hoare logic component that provides approximative information about the local memory contents at different program points. The compositionality of CA-security, and the safety and soundness of the type system, are verified using the Coq proof assistant.

The benefits of the development on concurrent program security are two fold: the policies are more expressive/informative and the analysis can be made more permissive/precise. This is demonstrated by the security of a multiplexer pattern (a recurring pattern in the MILS security architecture) that separates traffics at two different protection levels. The presence integrity of the communi-

cations can be easily corrupted, while the *content integrity* of the traffics at the sources and sinks still enjoy a clear correspondence. When the pattern is decomposed into a multiplexer and a demultiplexer sharing a communication channel, the content-dependent policy of the shared channel gives out rich information on the correspondence between the protection level of the messages and their tagging. This enables the modular typing of the multiplexer and demultiplexer processes in our Hoare logic equipped security type system.

6.1 Discussion, and Future Work

Relativity of Two-run Security Our bisimulation-based security properties belong to the so-called class of "two-run security" (e.g., [vDHS15]). When the only security classes are H and L, each interpreted *relative to* the other, this two-run perspective is fairly sensible. However, in a principal-based label model (e.g., DLM [ML97]), two-run security mandates that the principals allowed as readers/writers in different labels are properly *related to* each other, without requiring directly that each reader/writer is indeed an (explicit or implicit) reader/writer. Take for example the simple assignment $x := y$, and a label model where each label is the set of allowed readers. Suppose the label of x is $\{A\}$, and the label of y is $\{A, B\}$. This assignment is typically considered secure. The intuition is the flow from y to x does not result in more principals than A and B (in the label of the source) to learn the information in y. The explanation in terms of two-run security is: $\{A\}$ indicates higher confidentiality than $\{A, B\}$ does — $\{A\}$ can be thought of as H and $\{A, B\}$ as L, and variation in the H-variable x certainly does not cause any variation in the L-variable y through the assignment (in fact the opposite is the case). However, two-run security does not really say that A is (resp. A and B are) the only allowed reader for x (resp. readers for y), since it is not broken by labeling x instead with $\{A, pl\}$ and labeling y instead with $\{A, B, pl\}$ where pl is an arbitrary list of principals. Likewise, two-run security does not really distinguish whether a label like this represents the set of *readers*, or the set of *non-writers*. This problem is spotted in [NNL15a]: with the use of principal-based labels, two-run security *only* demands the absence of ill-dependency — the same core shared between both confidentiality and integrity. This can lead to the thinking that confidentiality and integrity are the same, which is not always desirable.

Relation with Fault-Tolerance In fault tolerance, failures of a system can be divided in two classes: stopping failures and byzantine failures. A system admitting stopping failures can either stop functioning or function flawlessly, whereas a system admitting byzantine failures can function arbitrarily (e.g., [Lyn96]). Concerning the provision of data on channels, stopping failures thus correspond to either the suspension of data or the provision of correct data. On the other hand, byzantine failures correspond to the suspension or corruption of

data. In the sense above there is thus some connection between stopping failures and channels with LH-integrity, and between byzantine failures and channels with LL-integrity.

Refinement The preservation of security under refinement [DNH84, AH89] is an important issue for process calculi [Ros95, BFPR03b] and event-based systems [Man01, vdM12, VDMZ13]. We have only shown that our notion of δ-security is preserved under the restriction of environment strategies that is in a sense "horizontal refinement" [RG01, BPR04] of the environment. But the the problem of dealing with the refinement of the system specification is not addressed. However, this does not defeat the purpose of the security condition proposed in Chapter 4, and the relevant issues about it studied there, since the modeling of systems, in itself, can already uncover places where information leakage/corruption can happen (e.g., [KT09]). Moreover, when dealing with concurrent programming languages involving communication, the problem of distinguishing between the presence and content of communication recurs, and can be studied similarly. Nonetheless, identifying the conditions/operators under which δ-security is preserved under refinement is an important issue, which we leave for future work.

Nondeterminism Our language used in Chapter 5 does not have an operator for non-deterministic choice. Introducing nondeterministic choice, however, is not completely trivial, since the compositionality proof of CA-security leverages intra-thread determinacy on the use of channels and polarities. We leave further study on the impact of nondeterministic/probabilistic choice to future work.

Probabilistic Schedulers For multithreaded programs, the scheduling problem is an important and active area of research in its own [SS00, Sab03, MZ08, MS10, BRRS10, PHN13, VDMZ13]. It is interesting to investigate the use of probabilistic strategies and schedulers to express our top-level notion of content-dependent security. Analogous to the use of non-deterministic strategies in Chapter 4, we could use probabilistic strategies to capture the variations on the channels over which the environment is ready to interact with the system, as well as on the potential contents that the environment can potentially provide to the system. Going beyond the use of deterministic schedulers in Chapter 5, the use of probabilistic schedulers would enable us to frame our property as:

1. the resulting distributions over the potential public memories reached (resp. contents of communications performed) are the same after (resp. within) the next steps of the related pair of executions, and

2. the distributions over the channels with public presence levels over which communications can be performed are the same in the next steps of the related pair of executions.

Presence-Dependency Consider the following process where the subscripts are integrity levels. The input over c_{LH} is of high content integrity but with weaker guarantee in its presence; thus when it cannot happen in time, the other input over c_{HL} with opposite guarantees will be effective. The channel c'_{HL} has L for its content integrity since it is shared between two outputs, one of low content integrity. However, it is desirable to allow finer-grained typing on the content of the output channel. In the sense of *flow-sensitivity*, the output content is of low integrity if and only if it is performed from the second branch. But a more directly intuitive viewpoint is that of *presence-dependency* — the content of the output is of low integrity if and only if the input over c_{LH} has been *absent*.

$$
\begin{array}{l}
\&_2(c_{LH}?x_1, c_{HL}?x_2). \\
\quad \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{some}(y_1) : c'_{HL}!y_1 \\
\quad \mathsf{else\ case}\ x_2\ \mathsf{of}\ \mathsf{some}(y_2) : c'_{HL}!y_2 \\
\quad \mathsf{else}\ 0
\end{array}
$$

We leave it to future work to address this problem, in a language equipped with binders like those in the quality calculus.

Certified Type Checker The type system of Section 5.5 has been implemented from scratch in the OCaml language, rather than extracted from its specification in Coq. The main obstacles have been twofold:

1. Our specification in Coq is declarative, rather than computational, which has led to more conciseness but less amenability to program extraction.
2. Type checking needs to interface with an external solver, and it would be desirable to re-certify the computation performed by the solver in Coq.

Concerning the first issue, it is desirable to specify a type checking algorithm in the computational fragment of Coq and formally prove that it is sound with respect to the current specification. Concerning the second, an integration of Coq with SMT solving that supports the checking of external certificates is SMTCoq [AFG+11]. An alternative possibility is to implement the automation required by the type checker directly inside Coq, which is expected to be more difficult. We leave it to future work to explore these directions.

Asynchronous Communication Another line of future work is dealing with *asynchronous communication*, which is meaningful for distributed systems. The use of message queues (e.g., [HYC08]) or pattern matching (e.g., [DNFP98]) will be expected in the semantics. The treatment of presence and content of two well-defined aspects of communication will still be an enabler of finer-grained security definition and enforcement.

Probabilistic, Timing Channels In process algebraic-security, an existing line of research deals with probabilistic, timing channels (e.g., [Ald01, FGM00]) in process calculi with probabilistic choices and actions corresponding to the

ticks of clocks. In language-based security, recent research addresses low-level external timing channels such as those through cache behaviors [ZAM11]. The endeavors made in this thesis are, however, not along these directions.

Appendix A

Proofs for Chapter 4

Lemma A.1 *The following statements hold for all sequences π of abstract binders without holes.*

1. $b \xrightarrow{c!c'} b'$ *if and only if* $\langle b,\ \pi_{\triangle} c?[\,]\rangle \xrightarrow{c!c'} \langle b',\ \pi.c?c'_{\triangle}\rangle$
2. $b \xrightarrow{c?c'} b'$ *if and only if* $\langle b,\ \pi_{\triangle} c!c'\rangle \xrightarrow{c?c'} \langle b',\ \pi.c!c'_{\triangle}\rangle$

Proof. The proof is by a straightforward induction on the semantic derivations. □

Lemma A.2 *The following statements hold for all sequences π of abstract binders without holes.*

1. $P \xrightarrow{c!c'} P'$ *if and only if* $\langle P,\ \pi_{\triangle} c?[\,]\rangle \xrightarrow{c!c'} \langle P',\ \pi.c?c'_{\triangle}\rangle$
2. $P \xrightarrow{c?c'} P'$ *if and only if* $\langle P,\ \pi_{\triangle} c!c'\rangle \xrightarrow{c?c'} \langle P',\ \pi.c!c'_{\triangle}\rangle$
3. $P \xrightarrow{\tau} P'$ *if and only if* $\langle P,\ \pi\rangle \xrightarrow{\tau} \langle P',\ \pi\rangle$

Proof. The proof by a straightforward induction on the semantic derivations, and use of Lemma A.1. □

Lemma A.3 *If $\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\tau} \langle P',\ \pi'\rangle$, then $\pi' = \pi$, and for all π_1, there exists π'_1 such that $\delta \vdash \langle P,\ \pi_1\rangle \xrightarrow{\tau} \langle P',\ \pi'_1\rangle$.*

PROOF. The proof is straightforward by induction on the derivation of $\delta \vdash \langle P,\ \pi\rangle \xrightarrow{\tau} \langle P',\ \pi'\rangle$. □

LEMMA A.4 *If $\delta \vdash \langle P,\ \pi\rangle \stackrel{\tau^n}{\Longrightarrow} \langle P',\ \pi'\rangle$, then $\pi' = \pi$, and for all π_1, there exists π_1' such that $\delta \vdash \langle P,\ \pi_1\rangle \stackrel{\tau^n}{\Longrightarrow} \langle P',\ \pi_1'\rangle$.*

PROOF. The proof is by induction on n.

- If $n = 0$ then $P' = P$. Hence there exists $\pi_1' = \pi_1$ such that $\delta \vdash \langle P,\ \pi_1\rangle \stackrel{\epsilon}{\Longrightarrow} \langle P,\ \pi_1'\rangle$ holds.
- We prove the case $n = k$ assuming the statement holds with $n = k - 1$. We have $\delta \vdash \langle P,\ \pi_1\rangle \xrightarrow{\tau} \langle P'',\ \pi''\rangle$ and $\delta \vdash \langle P'',\ \pi''\rangle \stackrel{\tau^{k-1}}{\Longrightarrow} \langle P',\ \pi_1'\rangle$. Pick arbitrary π_1''. By IH, $\pi' = \pi''$ and there exists π_1''', such that $\delta \vdash \langle P'',\ \pi_1''\rangle \stackrel{\tau^{k-1}}{\Longrightarrow} \langle P',\ \pi_1'''\rangle$. Still by IH, we have $\pi_1''' = \pi_1''$. By Lemma A.3, we have $\delta \vdash \langle P,\ \pi_1''\rangle \xrightarrow{\tau} \langle P'',\ \pi_1''\rangle$. Therefore we have $\delta \vdash \langle P,\ \pi_1''\rangle \stackrel{\tau^k}{\Longrightarrow} \langle P',\ \pi_1'''\rangle$.

This completes the proof. □

We next restate Lemma 4.1 and give its proof.

LEMMA 4.1 *For all processes P, P', actions α_1, α_2, ..., and α_n such that there is at most one $i \in \{1, ..., n\}$ for which $\alpha_i \neq \tau$, and $\forall i \in \{1, ..., n\} : \alpha_i \neq \Box$, histories π such that $\pi = [\pi]_\triangle$, and π_0 such that $\forall i \in \{1, ..., n\} : \alpha_i \neq \tau \Rightarrow \pi_0 = \pi.\widetilde{\alpha_i}$, the following are equivalent:*

1. *$P \stackrel{\alpha_1 \cdots \alpha_n}{\Longrightarrow} P'$, and*
2. *$\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle \wedge \exists \pi_0' : \delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \stackrel{\alpha_1 \cdots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$.*

PROOF.

1 ⇒ 2 We proceed with an induction on the length n of the transition sequence $P \stackrel{\alpha_1 \cdots \alpha_n}{\Longrightarrow} P'$.

- If $n = 0$, then $P = P'$. We have $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle$ and there exists $\pi_0' = \pi_0$ such that $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \stackrel{\epsilon}{\Longrightarrow} \langle P,\ \pi_0'\rangle$.
- We next show 1 ⇒ 2 with $n = k$ ($k \geq 1$) assuming that it holds with $n = k - 1$. We have $P \xrightarrow{\alpha_1} P''$ and $P'' \stackrel{\alpha_2 \cdots \alpha_n}{\Longrightarrow} P'$, where the latter transition sequence has length $k-1$. By the induction hypothesis, there exists some π_0' such that $\delta_{\mathrm{ALL}} \vdash \langle P'',\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P'',\ \pi_0\rangle$ and $\delta_{\mathrm{ALL}} \vdash \langle P'',\ \pi_0\rangle \stackrel{\alpha_2 \cdots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$. Hence we also have $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle$.

* $\alpha_1 = \tau$. By Lemma A.2, we have $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \xrightarrow{\alpha_1} \langle P'',\ \pi_0\rangle$. We therefore have $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \overset{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$.
* $\alpha_1 \neq \tau$. By Lemma A.2, there exists π_0'' such that $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \xrightarrow{\alpha_1} \langle P'',\ \pi_0''\rangle$. We know from the precondition that $\forall j \geq 2 : a_j = \tau$. By Lemma A.4, we have $\delta_{\mathrm{ALL}} \vdash \langle P'',\ \pi_0''\rangle \overset{\alpha_2 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0''\rangle$. We therefore have $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \overset{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0''\rangle$.

$2 \Rightarrow 1$ We proceed with an induction on the length n of the transition sequence $\delta_{\mathrm{ALL}} \vdash \langle P,\ \pi_0\rangle \overset{\alpha_1 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$.

- If $n = 0$, then $\langle P,\ \pi_0\rangle = \langle P',\ \pi_0'\rangle$. Hence $P' = P$ and we do have $P \overset{\epsilon}{\Longrightarrow} P$.
- We next show $2 \Rightarrow 1$ with $n = k$ $(k \geq 1)$ assuming that it holds with $n = k - 1$. There exists π_0' such that $\langle P,\ \pi_0\rangle \xrightarrow{\alpha_1} \langle P'',\ \pi_0''\rangle$, and $\langle P'',\ \pi_0''\rangle \overset{\alpha_2 \ldots \alpha_n}{\Longrightarrow} \langle P',\ \pi_0'\rangle$, where the latter transition sequence has length $k-1$. From $\langle P,\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P,\ \pi_0\rangle$ we know $\langle P'',\ \pi\rangle \xrightarrow{\mathrm{env}} \langle P'',\ \pi_0''\rangle$. By the induction hypothesis, we have $P'' \overset{\alpha_2 \ldots \alpha_n}{\Longrightarrow} P'$. By Lemma A.2, we have $P \xrightarrow{\alpha_1} P''$. Hence we have $P \overset{\alpha_1 \ldots \alpha_n}{\Longrightarrow} P'$.

This completes the proof. □

Lemma A.5 *If for all $c \in \mathbf{Ch}$, neither π_1 nor π_2 contains any $c?[\,]$, then*

$$\pi_1 \; W_{\mathcal{E}} \; \pi_2 \iff \pi_2 \; W_{\mathcal{E}} \; \pi_1.$$

Proof. Trivial. □

Theorem 4.18 *Suppose $\{\vec{c'}\} \cap \mathbf{Dt} = \emptyset$, $Sec_{\delta_{\mathrm{ALL}}}(P_1)$, and $Sec_{\delta_{\mathrm{ALL}}}(P_2)$. Then we have $Sec_{\delta_{\mathrm{ALL}}}((\nu\vec{c'})(P_1|P_2))$, provided*

$$\begin{aligned}\forall i \in \{1,2\}, c_{LH} \in \mathbf{Ch} : (c_{LH}, \rho_1) \in \mathsf{rch}(P_i) \wedge (c_{LH}, \rho_2) \in \mathsf{rch}(P_{3-i}) \\ \Rightarrow \rho_1 \neq \rho_2 \wedge \mathsf{det}(P_i, c_{LH}) \wedge c_{LH} \in \{\vec{c'}\}.\end{aligned}$$

Proof. Let $\widetilde{\pi}$ be the order-preserving sequence of all actions in π with all the polarities ρ changed to $\tilde{\rho}$. For convenience we rename the P_1 and P_2 in the precondition of Theorem 4.18 into P_1° and P_2°, and the vector $\vec{c'}$ into $\vec{c^\circ}$.

Construct the binary relation R as:

$$\begin{aligned}R = \{(\langle P_1,\ \pi_1\rangle, \langle P_2,\ \pi_2\rangle) \mid \; & \exists P_{11}, P_{12}, P_{21}, P_{22}, \pi_{11}, \pi_{12}, \pi_{21}, \pi_{22} : \\ & \psi(P_1, P_2, P_{11}, P_{12}, P_{21}, P_{22}, \pi_1, \pi_2, \pi_{11}, \pi_{12}, \pi_{21}, \pi_{22})\},\end{aligned}$$

where $\psi(P_1, P_2, P_{11}, P_{12}, P_{21}, P_{22}, \pi_1, \pi_2, \pi_{11}, \pi_{12}, \pi_{21}, \pi_{22})$ is the conjunction of the following clauses:

$$\forall i \in \{1,2\} : P_i \equiv (\nu\vec{c^\circ})(P_{1i}|P_{2i}) \tag{A.1}$$

$$\forall j, i \in \{1,2\} : \exists \pi' : \delta_{\mathrm{ALL}} \vdash \langle P_j^\circ,\ \triangle\rangle \longrightarrow^* \langle P_{ji},\ \pi'\rangle \tag{A.2}$$

$$\forall j \in \{1,2\} : \langle P_{j1},\ \pi_{j1}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{j2},\ \pi_{j2}\rangle \tag{A.3}$$

$$knl_{\mathcal{E}}(\pi_1, \pi_2) \tag{A.4}$$

$$\forall j \in \{1,2\} : knl_{\mathcal{E}}(\pi_{j1}, \pi_{j2}) \tag{A.5}$$

$$\begin{aligned}&\forall c \; s.t. \; \mathcal{E}^\circ(c) = L \wedge \mathcal{E}^\bullet(c) = H : \forall i \in \{1,2\} : \\ &((\exists \rho : (c,\rho) \in \mathsf{rch}(P_1) \wedge (c,\tilde{\rho}) \notin \mathsf{rch}(P_2)) \Rightarrow \pi_i \downarrow c = \pi_{1i} \downarrow c \wedge \pi_{2i} \downarrow c = \epsilon) \wedge \\ &((\exists \rho : (c,\rho) \in \mathsf{rch}(P_2) \wedge (c,\tilde{\rho}) \notin \mathsf{rch}(P_1)) \Rightarrow \pi_i \downarrow c = \pi_{2i} \downarrow c \wedge \pi_{1i} \downarrow c = \epsilon) \wedge \\ &((\exists \rho : (c,\rho) \in \mathsf{rch}(P_1) \wedge (c,\tilde{\rho}) \in \mathsf{rch}(P_2)) \Rightarrow \pi_{2i} \downarrow c = \widetilde{\pi_{1i} \downarrow c} \wedge \pi_i \downarrow c = \epsilon)\end{aligned} \tag{A.6}$$

We show that R qualifies as a δ_{ALL}-bisimulation. Then $Sec_{\delta_{\mathrm{ALL}}}((\nu\vec{c^\circ})(P_1^\circ|P_2^\circ))$ will follow, since it holds that

$$\psi((\nu\vec{c^\circ})(P_1^\circ|P_2^\circ), (\nu\vec{c^\circ})(P_1^\circ|P_2^\circ), P_1^\circ, P_1^\circ, P_2^\circ, P_2^\circ, \triangle, \triangle, \triangle, \triangle, \triangle, \triangle),$$

and we thus have $\langle(\nu\vec{c^\circ})(P_1^\circ|P_2^\circ),\ \triangle\rangle \; R \; \langle(\nu\vec{c^\circ})(P_1^\circ|P_2^\circ),\ \triangle\rangle$.

Take arbitrary $(\langle P_1,\ \pi_1\rangle, \langle P_2,\ \pi_2\rangle) \in R$, α, P_1', π_1' and π_{20} such that there exists π_{10},

$$\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_1\rangle \xrightarrow{\mathrm{env}} \langle P_1,\ \pi_{10}\rangle$$

$$\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\alpha} \langle P_1',\ \pi_1'\rangle \tag{A.7}$$

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_2\rangle \xrightarrow{\mathrm{env}} \langle P_2,\ \pi_{20}\rangle \tag{A.8}$$

$$\pi_1' \; W_{\mathcal{E}} \; \pi_{20} \tag{A.9}$$

We distinguish between several cases for the transition (A.7) above.

1. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{c_1\rho c'} \langle P_1',\ \pi_1'\rangle$, for some c_1, ρ, c', where $P_1' \equiv (\nu \vec{c^\circ})(P_{11}'|P_{21})$ for some P_{11}'. That is to say, (A.7) is built on top of a local transition from P_{11}. Ite must be the case that $\pi_1' = \pi_1.c_1\widetilde{\rho}c'_{\triangle}$, and there exists the following transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_{11},\ \pi_{11}\rangle \xrightarrow{\mathrm{env},c_1\rho c'} \langle P_{11}',\ \pi_{11}.c\widetilde{\rho}c'_{\triangle}\rangle \tag{A.10}$$

   (a) $\mathcal{E}^\circ(c) = H$. By (A.4) and (A.9), $\pi_{20} = \pi_2.c_1\widetilde{\rho}c''$ for some c'' (which can be $[\,]$). By (A.5), it holds that $\pi_{11}.c_1\widetilde{\rho}c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.c_1\widetilde{\rho}c''$. By (A.3) and (A.10), there exists

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.c_1\widetilde{\rho}c''\rangle \overset{\widehat{c_1\rho c'''}}{\Longrightarrow} \langle P_{12}',\ \pi_{12}.c_1\widetilde{\rho}c'''_{\triangle}\rangle \tag{A.11}$$

   for some c''' such that

$$\langle P_{11}',\ \pi_{11}.c_1\widetilde{\rho}c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{12}',\ \pi_{12}.c_1\widetilde{\rho}c'''_{\triangle}\rangle,$$

   and

$$knl_{\mathcal{E}}(\pi_{11}.c_1\widetilde{\rho}c'_{\triangle}, \pi_{12}.c_1\widetilde{\rho}c'''_{\triangle}).$$

   It is not difficult to verify:

$$\begin{aligned}&\psi(P_1', (\nu\vec{c^\circ})(P_{12}'|P_{22}), P_{11}', P_{12}', P_{21}, P_{22},\\ &\quad \pi_1.c_1\widetilde{\rho}c'_{\triangle}, \pi_2.c_1\widetilde{\rho}c'''_{\triangle}, \pi_{11}.c_1\widetilde{\rho}c'_{\triangle}, \pi_{12}.c_1\widetilde{\rho}c'''_{\triangle}, \pi_{21}, \pi_{22}).\end{aligned}$$

   By (A.1), transition (A.7), and $\alpha = c_1\rho c'$, we have $c_1 \notin \{\vec{c^\circ}\}$. Since $c''' \in \mathbf{Dt}$ by (A.11), we have $c''' \notin \{\vec{c^\circ}\}$. Therefore transition (A.7) can be simulated by the following transition in R:

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \overset{\widehat{c_1\rho c'''}}{\Longrightarrow} \langle (\nu\vec{c^\circ})(P_{12}'|P_{22}),\ \pi_2.c_1\widetilde{\rho}c'''_{\triangle}\rangle.$$

   (b) $\mathcal{E}^\circ(c) = L$. In this case $\pi_{20} = \pi_2.\alpha'$ where $\alpha' = c_2\widetilde{\rho'}c''$ for some c_2, ρ' and c'' (which can be $[\,]$), or $\alpha' = \Box$. By (A.9), $\mathcal{E}^\circ(c_2) = L$.

      i. Suppose $\langle P_2,\ \pi_{20}\rangle$ can perform a communication (output or input). Then either $\langle P_{12},\ \pi_{20}\rangle$ or $\langle P_{22},\ \pi_{20}\rangle$ can perform the communication.

         A. $\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{20}\rangle \xrightarrow{c_2\rho' c'''} \langle P_{12}',\ \pi_2.c_2\widetilde{\rho'}c'''_{\triangle}\rangle$ for some P_{12}' and c'''. Using (A.2), (A.5), (A.6) and (A.9) we can deduce

$$\pi_{11}.c_1\widetilde{\rho}c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.c_2\widetilde{\rho'}c''.$$

         By (A.3), the transition (A.10) can be simulated locally by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.c_2\rho' c''\rangle \xrightarrow{c_2\rho' c'_2} \langle P_{12}'',\ \pi_{12}.c_2\widetilde{\rho'}c'_{2\triangle}\rangle \tag{A.12}$$

for some c'_2 and P''_{12}, resulting in

$$\langle P'_{11},\ \pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P''_{12},\ \pi_{12}.c_2\widetilde{\rho'}c'_{2\triangle}\rangle,$$

and

$$knl_{\mathcal{E}}(\pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}, \pi_{12}.c_2\widetilde{\rho'}c'_{2\triangle}).$$

Using (A.2), (A.4), (A.6), and the precondition of this theorem, we can deduce $knl_{\mathcal{E}}(\pi_1.c_1\widetilde{\rho}c'_{2\triangle}, \pi_2.c_2\widetilde{\rho'}c'_{2\triangle})$. On this basis, it is not difficult to verify:

$$\begin{aligned}&\psi(P'_1, (\nu\vec{c^{\diamond}})(P''_{12}|P_{22}), P'_{11}, P''_{12}, P_{21}, P_{22},\\ &\quad \pi_1.c_1\widetilde{\rho}c'_{2\triangle}, \pi_2.c_2\widetilde{\rho'}c'_{2\triangle}, \pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}, \pi_{12}.c_2\widetilde{\rho'}c'_{2\triangle}, \pi_{21}, \pi_{22}).\end{aligned}$$

Since $\alpha' = c_2\widetilde{\rho'}c''$, $\langle P_2,\ \pi_2.\alpha'\rangle$ can perform output/input over c_2. By (A.1) we have $c_2 \notin \{\vec{c^{\diamond}}\}$. In addition, since $c'_2 \in \mathbf{Dt}$ by (A.12), we have $c'_2 \notin \{\vec{c^{\diamond}}\}$. Hence the transition (A.7) can be simulated by the following transition in R:

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \overset{c_2\rho'c'_2}{\longrightarrow} \langle (\nu\vec{c^{\diamond}})(P''_{12}|P_{22}),\ \pi_2.c_2\widetilde{\rho'}c'_{2\triangle}\rangle.$$

B. $\delta_{\mathrm{ALL}} \vdash \langle P_{22},\ \pi_{20}\rangle \overset{c_2\rho'c'''}{\longrightarrow} \langle P'_{22},\ \pi_2.c_2\widetilde{\rho'}c'''_{\triangle}\rangle$ for some P'_{22} and c'''. We have the transition

$$\delta_{\mathrm{ALL}} \vdash \langle P_{22},\ \pi_{22}\rangle \overset{\mathrm{env},c_2\rho'c'''}{\longrightarrow} \langle P'_{22},\ \pi_{22}.c_2\widetilde{\rho'}c'''_{\triangle}\rangle. \tag{A.13}$$

Using (A.2), (A.5), (A.6) and (A.9) we can deduce

$$\pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}\ W_{\mathcal{E}}\ \pi_{12}.\Box, \tag{A.14}$$

and

$$\pi_{22}.c_2\widetilde{\rho'}c'''_{\triangle}\ W_{\mathcal{E}}\ \pi_{21}.\Box, \tag{A.15}$$

since $c''' \neq [\,]$.
By (A.3) and (A.14), the transition (A.10) can be simulated by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.\Box\rangle \overset{\Box}{\longrightarrow} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle,$$

with

$$\langle P'_{11},\ \pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle,$$

and

$$knl_{\mathcal{E}}(\pi_{11}.c_1\widetilde{\rho}c'_{2\triangle}, \pi_{12}.\Box_{\triangle}).$$

By (A.3), the symmetry of $\underset{\delta_{\mathrm{ALL}}}{\sim}$, and (A.15), the transition (A.13) can be simulated by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{21},\ \pi_{21}.\Box\rangle \overset{\Box}{\longrightarrow} \langle P_{21},\ \pi_{21}.\Box_{\triangle}\rangle,$$

with

$$\langle P_{21},\ \pi_{21}.\Box_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P'_{22},\ \pi_{22}.c_2\widetilde{\rho'}c'''_{\triangle}\rangle,$$

and

$$knl_{\mathcal{E}}(\pi_{21}.\Box_{\triangle}, \pi_{22}.c_2\widetilde{\rho'}c'''_{\triangle}).$$

Using (A.2), (A.4), (A.6), and the precondition of this theorem, we can deduce $knl_{\mathcal{E}}(\pi_1.c_1\widetilde{\rho}c'_{\triangle}, \pi_2.c_2\widetilde{\rho'}c'''_{\triangle})$. On this basis, it is not difficult to verify:

$$\psi(P'_1, (\nu\vec{c^{\diamond}})(P_{12}|P'_{22}), P'_{11}, P_{12}, P_{21}, P'_{22},\\ \pi_1.c_1\widetilde{\rho}c'_{\triangle}, \pi_2.c_2\widetilde{\rho'}c'''_{\triangle}, \pi_{11}.c_1\widetilde{\rho}c'_{\triangle}, \pi_{12}.\Box_{\triangle}, \pi_{21}.\Box_{\triangle}, \pi_{22}.c_2\widetilde{\rho'}c'''_{\triangle}).$$

Since $\alpha' = c_2\widetilde{\rho'}c''$, $\langle P_2,\ \pi_2.\alpha'\rangle$ can perform output/input over c_2. By (A.1) we have $c_2 \notin \{\vec{c^{\diamond}}\}$. By (A.13) we have $c''' \in \mathbf{Dt}$; hence $c''' \notin \{\vec{c^{\diamond}}\}$. Hence the transition (A.7) can be simulated by the following transition in R:

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{c_2\rho'c'''} \langle(\nu\vec{c^{\diamond}})(P_{12}|P'_{22}),\ \pi_2.c_2\widetilde{\rho'}c'''_{\triangle}\rangle.$$

ii. Suppose $\langle P_2,\ \pi_{20}\rangle$ cannot perform any communication (output/input). We have $\pi_{11}.c_1\widetilde{\rho}c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.\Box$. By (A.5), transition (A.10) can be simulated by

$$\langle P_{12},\ \pi_{12}.\Box\rangle \xrightarrow{\Box} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle,$$

with $\pi_{11}.c_1\rho c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.\Box_{\triangle}$, and

$$\langle P'_{11},\ \pi_{11}.c_1\rho c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle.$$

On this basis, we can deduce that $knl_{\mathcal{E}}(\pi_1.c_1\rho c'_{\triangle}, \pi_2.\Box_{\triangle})$, and

$$\psi(P'_1, P_2, P'_{11}, P_{12}, P_{21}, P_{22},\\ \pi_1.c\widetilde{\rho}c'_{\triangle}, \pi_2.\Box_{\triangle}, \pi_{11}.c_1\widetilde{\rho}c'_{\triangle}, \pi_{12}.\Box_{\triangle}, \pi_{21}, \pi_{22}).$$

Hence the transition (A.7) can be simulated by the following transition in R:

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xrightarrow{\Box} \langle P_2,\ \pi_2.\Box_{\triangle}\rangle.$$

2. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{c_1\rho c'} \langle P'_1,\ \pi'_1\rangle$, for some c_1, ρ, c', where $P'_1 \equiv (\nu\vec{c^{\diamond}})(P_{11}|P'_{21})$ for some P'_{21}. That is to say, (A.7) is built on top of a local transition from P_{21}. The reasoning needed is analogous to that of case 1.

3. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\tau} \langle P_1',\ \pi_1'\rangle$, for some c_1, ρ, c', where $P_1' \equiv (\nu\vec{c^{\circ}})(P_{11}'|P_{21})$ for some P_{11}'. That is to say, (A.7) is built on top of a local τ-step from P_{11}. Since $\Box \in \delta_{\mathrm{ALL}}(\pi_{11})$, We also have

$$\langle P_{11},\ \pi_{11}\rangle \xrightarrow{\mathrm{env},\tau} \langle P_{11}',\ \pi_{11}.\Box\rangle. \tag{A.16}$$

From (A.5) we have $\pi_{11}.\Box\ W_{\mathcal{E}}\ \pi_{12}.\Box$. Thus by (A.3), the transition (A.16) can be simulated locally by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.\Box\rangle \stackrel{\widehat{\tau}}{\Longrightarrow} \langle P_{12}',\ \pi_{12}.\Box\rangle$$

for some P_{12}', with $\langle P_{11}',\ \pi_{11}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{12}',\ \pi_{12}\rangle$ and $knl_{\mathcal{E}}(\pi_{11}, \pi_{12})$.

It is not difficult to verify:

$$\psi(P_1', (\nu\vec{c^{\circ}})(P_{12}'|P_{22}), P_{11}', P_{12}', P_{21}, P_{22}, \pi_1, \pi_2, \pi_{11}, \pi_{12}, \pi_{21}, \pi_{22}).$$

Hence the transition (A.7) can be simulated by the following transition in R:

$$\langle P_2,\ \pi_{20}\rangle \stackrel{\widehat{\tau}}{\Longrightarrow} \langle (\nu\vec{c^{\circ}})(P_{12}'|P_{22}),\ \pi_{20}\rangle.$$

4. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\tau} \langle P_1',\ \pi_1'\rangle$, for some c_1, ρ, c', where $P_1' \equiv (\nu\vec{c^{\circ}})(P_{11}|P_{21}')$ for some P_{21}'. That is to say, (A.7) is built on top of a local τ-step from P_{21}. The reasoning is similar to that of case 3.

5. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\tau} \langle P_1',\ \pi_1'\rangle$, for some c_1, ρ, c', where $P_1' \equiv (\nu\vec{c^{\circ}})(P_{11}'|P_{21}')$ for some P_{11}' and P_{21}', i.e., the transition is produced by communication between P_{11} and P_{12}.

   Without loss of generality, suppose

$$\delta_{\mathrm{ALL}} \vdash \langle P_{11},\ {}_{\triangle}c?[]\rangle \xrightarrow{c!c'} \langle P_{11}',\ c?c'_{\triangle}\rangle \tag{A.17}$$

$$\delta_{\mathrm{ALL}} \vdash \langle P_{21},\ {}_{\triangle}c!c'\rangle \xrightarrow{c?c'} \langle P_{21}',\ c!c'_{\triangle}\rangle \tag{A.18}$$

   Thus there exists the following transitions:

$$\delta_{\mathrm{ALL}} \vdash \langle P_{11},\ \pi_{11}\rangle \xrightarrow{\mathrm{env},c!c'} \langle P_{11}',\ \pi_{11}.c?c'_{\triangle}\rangle \tag{A.19}$$

$$\delta_{\mathrm{ALL}} \vdash \langle P_{21},\ \pi_{21}\rangle \xrightarrow{\mathrm{env},c?c'} \langle P_{21}',\ \pi_{21}.c!c'_{\triangle}\rangle \tag{A.20}$$

   We distinguish between two cases: $\mathcal{E}^{\circ}(c) = H$ and $\mathcal{E}^{\circ}(c) = L$.

   (a) $\mathcal{E}^{\circ}(c) = H$.
   By (A.5) we have $\pi_{11}.c?c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.c?[\,]$. We also have $\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}\rangle \xrightarrow{\mathrm{env}} \langle P_{12},\ \pi_{12}.c?[]\rangle$. By (A.3) and transition (A.19), there exists P_{12}' and c'' ($\mathcal{E}^{\bullet}(c) = H \Rightarrow c'' = c'$) such that

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.c?[]\rangle \stackrel{\widehat{c!c''}}{\Longrightarrow} \langle P_{12}',\ \pi_{12}.c?c''_{\triangle}\rangle,$$

with $\langle P'_{11},\ \pi_{11}.c?c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P'_{12},\ \pi_{12}.c?c''_{\triangle}\rangle$ and $\pi_{11}.c?c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.c?c''_{\triangle}$. By (A.5), we have $\pi_{21}.c!c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{22}.c!c''$. We also have $\delta_{\mathrm{ALL}} \vdash \langle P_{22},\ \pi_{22}\rangle \xrightarrow{\mathrm{env}} \langle P_{22},\ \pi_{22}.c!c''\rangle$. By (A.3) and transition (A.20), there exists P'_{22} such that

$$\delta_{\mathrm{ALL}} \vdash \langle P_{22},\ \pi_{22}.c!c''\rangle \xRightarrow{\widehat{c?c''}} \langle P'_{22},\ \pi_{22}.c!c''_{\triangle}\rangle,$$

with $\langle P'_{21},\ \pi_{21}.c!c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P'_{22},\ \pi_{22}.c!c''_{\triangle}\rangle$ and $\pi_{21}.c!c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{22}.c!c''_{\triangle}$. It is not difficult to verify:

$$\psi(P'_1, (\nu\vec{c^{\diamond}})(P'_{21}|P'_{22}), P'_{11}, P'_{12}, P'_{21}, P'_{22},\\ \pi_1, \pi_2, \pi_{11}.c?c'_{\triangle}, \pi_{12}.c?c''_{\triangle}, \pi_{21}.c!c''_{\triangle}, \pi_{22}.c!c''_{\triangle}).$$

Hence the transition (A.7) can be simulated by the following transition sequence in R:

$$\delta_{\mathrm{ALL}} \vdash \langle P_2,\ \pi_{20}\rangle \xRightarrow{\widehat{\tau}} \langle (\nu\vec{c^{\diamond}})(P'_{12}|P'_{22}),\ \pi_{20}\rangle.$$

(b) $\mathcal{E}^{\circ}(c) = L$.

i. $\mathcal{E}^{\bullet}(c) = L$. By (A.5), we have $\pi_{11}.c?c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.\Box$. We also have

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}\rangle \xrightarrow{\mathrm{env}} \langle P_{12},\ \pi_{12}.\Box\rangle.$$

By (A.3), transition (A.19) can be simulated by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{12},\ \pi_{12}.\Box\rangle \xrightarrow{\Box} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle,$$

with $\langle P'_{11},\ \pi_{11}.c?c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{12},\ \pi_{12}.\Box_{\triangle}\rangle$ and $\pi_{11}.c?c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{12}.\Box_{\triangle}$. For similar reasons transition (A.20) can be simulated by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{22},\ \pi_{22}.\Box\rangle \xrightarrow{\Box} \langle P_{22},\ \pi_{22}.\Box_{\triangle}\rangle,$$

with $\langle P'_{21},\ \pi_{21}.c!c'_{\triangle}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{22},\ \pi_{22}.\Box_{\triangle}\rangle$, and $\pi_{21}.c!c'_{\triangle}\ W_{\mathcal{E}}\ \pi_{22}.\Box_{\triangle}$. It is not difficult to verify:

$$\psi(P'_1, P_2, P'_{11}, P_{12}, P'_{21}, P_{22}, \pi_1, \pi_2, \pi_{11}.c?c'_{\triangle}, \pi_{12}.\Box_{\triangle}, \pi_{21}.c!c'_{\triangle}, \pi_{22}.\Box_{\triangle}).$$

Thus transition (A.7) can be simulated by 0-step (in τ^*) from $\langle P_2,\ \pi_{20}\rangle$.

ii. $\mathcal{E}^{\bullet}(c) = H$.
By (A.3), and the symmetry of $\underset{\delta_{\mathrm{ALL}}}{\sim}$, we have

$$\langle P_{12},\ \pi_{12}\rangle \underset{\delta_{\mathrm{ALL}}}{\sim} \langle P_{11},\ \pi_{11}\rangle.$$

By $\mathcal{E}^\circ(c) = L$ and (A.5), we have

$$\pi_{12}.\Box_\triangle \; W_\mathcal{E} \; \pi_{11}.c?[].$$

We also have the transition

$$\delta_{\text{ALL}} \vdash \langle P_{12},\ \pi_{12}\rangle \xrightarrow{\text{env},\Box} \langle P_{12},\ \pi_{12}.\Box_\triangle\rangle.$$

Hence there exists some c'' and P'_{11} such that

$$\delta_{\text{ALL}} \vdash \langle P_{11},\ \pi_{11}.c?[]\rangle \xrightarrow{c!c''} \langle P'_{11},\ \pi_{11}.c?c''_\triangle\rangle, \tag{A.21}$$

with $\langle P_{12},\ \pi_{12}.\Box_\triangle\rangle \underset{\delta_{\text{ALL}}}{\sim} \langle P'_{11},\ \pi_{11}.c?c''_\triangle\rangle$ and $\pi_{12}.\Box_\triangle \; W_\mathcal{E} \; \pi_{11}.c?c''_\triangle$. By the symmetry of $\underset{\delta_{\text{ALL}}}{\sim}$, we have

$$\langle P'_{11},\ \pi_{11}.c?c''_\triangle\rangle \underset{\delta_{\text{ALL}}}{\sim} \langle P_{12},\ \pi_{12}.\Box_\triangle\rangle.$$

By Lemma A.5, we have

$$\pi_{11}.c?c''_\triangle \; W_\mathcal{E} \; \pi_{12}.\Box_\triangle.$$

By $\mathsf{det}(P_1^\circ, c)$, (A.19), (A.21) and (A.2), we have $c' = c''$, which gives

$$\pi_{11}.c?c'_\triangle \; W_\mathcal{E} \; \pi_{12}.\Box. \tag{A.22}$$

We also have the following transition based on transition (A.21):

$$\delta_{\text{ALL}} \vdash \langle P_{11},\ \pi_{11}\rangle \xrightarrow{\text{env},c!c'} \langle P'_{11},\ \pi_{11}.c?c'_\triangle\rangle.$$

The transition (A.19) can thus be simulated by

$$\delta_{\text{ALL}} \vdash \langle P_{12},\ \pi_{12}.\Box\rangle \xrightarrow{\Box} \langle P_{12},\ \pi_{12}.\Box_\triangle\rangle.$$

From (A.17), (A.18) and (A.2), we know that $(c, !) \in \mathsf{rch}(P_1^\circ)$ and $(c, ?) \in \mathsf{rch}(P_2^\circ)$. Hence by (A.5), (A.6) and (A.22), we have

$$\pi_{21}.c!c'_\triangle \; W_\mathcal{E} \; \pi_{22}.\Box.$$

By (A.3), the transition (A.20) can be simulated by

$$\delta_{\text{ALL}} \vdash \langle P_{22},\ \pi_{22}.\Box\rangle \xrightarrow{\Box} \langle P_{22},\ \pi_{22}.\Box_\triangle\rangle,$$

with $\langle P'_{21},\ \pi_{21}.c!c'_\triangle\rangle \underset{\delta_{\text{ALL}}}{\sim} \langle P_{22},\ \pi_{22}.\Box_\triangle\rangle$, and $\pi_{21}.c!c'_\triangle \; W_\mathcal{E} \; \pi_{22}.\Box_\triangle$. It can now be verified that

$$\psi(P'_1, P_2, P'_{11}, P_{12}, P'_{21}, P_{22}, \pi_1, \pi_2, \pi_{11}.c?c'_\triangle, \pi_{12}.\Box_\triangle, \pi_{21}.c!c'_\triangle, \pi_{22}.\Box_\triangle).$$

Hence the transition (A.7) can be simulated by zero-step (in τ^*) from $\langle P_2,\ \pi_{20}\rangle$.

6. The transition (A.7) is $\delta_{\mathrm{ALL}} \vdash \langle P_1,\ \pi_{10}\rangle \xrightarrow{\Box} \langle P_1',\ \pi_1'\rangle$, where $P_1' \equiv P_1$.

   By (A.9) we know that $\pi_{20} = \pi_2.c_2\widetilde{\rho'}c''$ with some c_2 such that $\mathcal{E}^\circ(c_2) = L$, and some ρ' and c'', or $\pi_{20} = \pi_2.\Box$.

   We have

$$\delta_{\mathrm{ALL}} \vdash \langle P_{11},\ \pi_{11}\rangle \xrightarrow{\mathrm{env}} \langle P_{11},\ \pi_{11}.\Box\rangle$$
$$\delta_{\mathrm{ALL}} \vdash \langle P_{21},\ \pi_{21}\rangle \xrightarrow{\mathrm{env}} \langle P_{21},\ \pi_{21}.\Box\rangle$$

   As in Case (1b), we distinguish between a few cases about the action that can be performed from $\langle P_2,\ \pi_{20}\rangle$. The reasoning in each case is similar to that of Case (1b). The symmetry of $\underset{\delta_{\mathrm{ALL}}}{\sim}$ needs to be exploited and "local simulations" by

$$\delta_{\mathrm{ALL}} \vdash \langle P_{11},\ \pi_{11}.\Box\rangle \xrightarrow{\Box} \langle P_{11},\ \pi_{11}.\Box_\triangle\rangle,$$

   and

$$\delta_{\mathrm{ALL}} \vdash \langle P_{21},\ \pi_{21}.\Box\rangle \xrightarrow{\Box} \langle P_{21},\ \pi_{21}.\Box_\triangle\rangle$$

   will be used. We do not cover further details.

The discussion above completes the proof. □

Appendix B

Outline of Coq Proofs for Chapter 5

B.1 Library Language

```
Require Export Utils.

Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp.

Tactic Notation "aexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ANum" | Case_aux c "AId" | Case_aux c "APlus"
  | Case_aux c "AMinus" | Case_aux c "AMult" ].

Definition var_set : Type := Ensemble id.

Fixpoint fva (a : aexp) : var_set :=
  match a with
  | ANum n ⇒ Empty_set id
  | AId x ⇒ Singleton id x
  | APlus a1 a2 ⇒ Union id (fva a1) (fva a2)
  | AMinus a1 a2 ⇒ Union id (fva a1) (fva a2)
  | AMult a1 a2 ⇒ Union id (fva a1) (fva a2)
  end.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
```

```
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp.

Tactic Notation "bexp_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "BTrue" | Case_aux c "BFalse" | Case_aux c "BEq"
  | Case_aux c "BLe" | Case_aux c "BNot" | Case_aux c "BAnd" ].

Fixpoint fvb (b : bexp) : var_set :=
  match b with
  | BTrue ⇒ Empty_set id
  | BFalse ⇒ Empty_set id
  | BEq a1 a2 ⇒ Union id (fva a1) (fva a2)
  | BLe a1 a2 ⇒ Union id (fva a1) (fva a2)
  | BNot b ⇒ fvb b
  | BAnd b1 b2 ⇒ Union id (fvb b1) (fvb b2)
  end.

Definition block : Type := var_set.

Inductive pch : Type :=
| PChan : nat → pch.

Definition pch_def := PChan 0.

Definition pch_set : Type := Ensemble pch.

Inductive ch : Type :=
  | Chan : pch → nat → ch.

Inductive com : Type :=
  | CSkip : com
  | CNil : com
  | CAss : id → aexp → com
  | COut : pch → list aexp → com
  | CIn : pch → list id → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Tactic Notation "com_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "SKIP" | Case_aux c "STOP"
  | Case_aux c "::=" | Case_aux c "!" | Case_aux c "?"
  | Case_aux c ";;" | Case_aux c "IF" | Case_aux c "WHILE" ].

Notation "'SKIP'" :=
  (CSkip) (at level 60).
Notation "'NIL'" :=
  (CNil) (at level 60).
Notation "x '::=' a" :=
  (CAss x a) (at level 60).
Notation "c '!' al" :=
```

```
  (COut c al) (at level 60).
Notation "c '??' xl" :=
  (CIn c xl) (at level 60).
Notation "S1 ;; S2" :=
  (CSeq S1 S2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' S 'END'" :=
  (CWhile b S) (at level 80, right associativity).
Notation "'IIF' b 'THEN' S1 'ELSE' S2 'FII'" :=
  (CIf b S1 S2) (at level 80, right associativity).

Fixpoint get_vars (S: com) : var_set :=
  match S with
  | CSkip ⇒ Empty_set _
  | CNil ⇒ Empty_set _
  | CAss x a ⇒ Singleton _ x |U| (fva a)
  | COut c al ⇒ Union_n (map fva al)
  | CIn cl xl ⇒ Union_n (map (Singleton _) xl)
  | CSeq S1 S2 ⇒ (get_vars S1) |U| (get_vars S2)
  | CWhile b S' ⇒ (fvb b) |U| (get_vars S')
  | CIf b S1 S2 ⇒ (fvb b) |U| (get_vars S1 |U| get_vars S2)
  end.

Lemma nil_not_nil :
  ∀ (S: com), S = NIL ∨ S ≠ NIL.

Definition pid : Type := nat.

Definition var_thread_t : Type := id → pid.

Inductive Sys : Type :=
| SysProc : pid → com → Sys
| SysPar : Sys → Sys → Sys
| SysRes : Sys → Ensemble pch → Sys.

Notation "i ':' S" :=
  (SysProc i S) (at level 70).
Notation "sys1 '|P|' sys2" :=
  (SysPar sys1 sys2) (at level 80).
Notation "sys '\' Omg" :=
  (SysRes sys Omg) (at level 75).

Tactic Notation "sys_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "PROC" | Case_aux c "PAR" | Case_aux c "RES" ].

Fixpoint get_pchs (S: com) : pch_set :=
  match S with
  | CSkip ⇒ Empty_set _
  | CNil ⇒ Empty_set _
  | CAss x a ⇒ Empty_set _
  | COut c al ⇒ Singleton _ c
  | CIn c xl ⇒ Singleton _ c
  | CSeq S1 S2 ⇒ (get_pchs S1) |U| (get_pchs S2)
```

```
  | CWhile b S' ⇒ get_pchs S'
  | CIf b S1 S2 ⇒ (get_pchs S1) |U| (get_pchs S2)
  end.

Fixpoint get_pids (sys: Sys) :=
  match sys with
  | SysProc j _ ⇒ [j]
  | SysPar sys1 sys2 ⇒ get_pids sys1 ++ get_pids sys2
  | SysRes sys _ ⇒ get_pids sys
  end.

Lemma pid_in_par :
  ∀ sys1 sys2 i,
    (List.In i (get_pids sys1) ∨ List.In i (get_pids sys2)) →
    (List.In i (get_pids (sys1 |P| sys2))).

Hint Resolve pid_in_par.

Fixpoint in_list
          (A: Type) (dec: ∀ a1 a2: A, {a1=a2} + {a1≠a2})
          (e: A) (l: list A) : bool :=
  match l with
  | nil ⇒ false
  | h :: t ⇒ if (dec h e) then true else in_list _ dec e t
  end.

Lemma bool_prop :
  ∀ (A:Type) (dec: ∀ a1 a2: A, {a1=a2} + {a1≠a2}) (a:A) lst,
    List.In a lst → in_list _ dec a lst = true.

Lemma prop_bool :
  ∀ (A:Type) (a:A) (dec: ∀ a1 a2: A, {a1=a2} + {a1≠a2}) lst,
    ¬ List.In a lst → in_list _ dec a lst = false.

Lemma prop_bool' :
  ∀ (A:Type) (a:A) (dec: ∀ a1 a2: A, {a1=a2} + {a1≠a2}) lst,
    in_list _ dec a lst = true → List.In a lst.

Hint Resolve prop_bool'.

Theorem eq_pid_dec : ∀ i j : pid, {i = j} + {i ≠ j}.

Fixpoint get_com (sys: Sys) i :=
  match sys with
  | SysProc _ SS ⇒ SS
  | SysPar sys1 sys2 ⇒
    if (in_list _ eq_pid_dec i (get_pids sys1))
    then get_com sys1 i
    else get_com sys2 i
  | SysRes sys' Omg ⇒ get_com sys' i
  end.

Eval compute in (get_com (1:SKIP |P| 2:((Id 1)::=ANum 0)) 2).

Fixpoint common
          (A:Type) (dec: ∀ a1 a2: A, {a1=a2} + {a1≠a2}) (l1 l2 : list A) : bool
```

```
  :=
  match l1 with
    | nil ⇒ false
    | h :: t ⇒ (in_list _ dec h l2) || (common _ dec t l2)
  end.
Inductive wf : Sys → Prop :=
  | WF_Sys_One : ∀ i SS, wf (i:SS)
  | WF_Sys_Par :
      ∀ sys1 sys2,
        wf sys1 → wf sys2 →
        (common _ eq_pid_dec (get_pids sys1) (get_pids sys2) = false) →
        wf (sys1 |P| sys2)
  | WF_Sys_Res : ∀ sys' Omg, wf sys' → wf (sys'\Omg)
.
```

B.2 Library Semantics

```
Require Export Language.
Variable U: Type.

Section Mappings.
Variable A: Type.
Definition Dec (A : Type) := ∀ a1 a2: A, {a1=a2} + {a1≠a2}.
Variable dec : Dec A.
Definition Sub (il: list pid) := (sig (fun i:pid ⇒ List.In i il)).
Record Mapping (A B:Type) (dec_A : Dec A) :=
  mkMapping {
      il: list A;
      mp: A→B
  }.
Definition map_def B (b_def: B) := mkMapping A B dec [] (fun _ ⇒ b_def).
Definition Mem := Mapping id nat eq_id_dec.
Definition m_empty : Mem := mkMapping id nat eq_id_dec [] (fun (x:id) ⇒ 0).
Definition u_map {B: Type} (il1 : list A) (m1 m2: A→B) :=
  fun (a : A) ⇒
    if (in_list _ dec a il1) then (m1 a) else (m2 a).
Lemma u_map_same :
  ∀ (B: Type) (il1 : list A) (m : A→ B),
    u_map il1 m m = m.
Definition m_union {B: Type} (m1 m2: Mapping A B dec) :
  option (Mapping A B dec) :=
  if (common _ dec (il A B _ m1) (il A B _ m2)) then
    None
```

```
  else
    Some (mkMapping A B dec
                    ((il A B _ m1) ++ (il A B _ m2))
                    (u_map (il A B _ m1) (mp A B _ m2))
         ).
Definition get_val {B: Type} (m: Mapping A B dec) (x:A) : option B :=
  if (in_list _ dec x (il _ _ dec m))
  then Some ((mp _ _ dec m) x)
  else None.
Definition upd_val {B: Type} m (a:A) (v:B) : (Mapping A B dec) :=
  if (in_list _ dec a (il _ _ dec m))
  then (mkMapping A B dec
                  (il _ _ dec m)
                  (fun (a1 : A) ⇒ if (dec a a1) then v else (mp _ _ dec m a1))
       )
  else m.
Theorem update_eq : ∀ {B: Type} (val: B) a m,
  List.In a (il A B _ m) →
  get_val (upd_val m a val) a = Some val.
Lemma neq_a : ∀ {B: Type} a1 a2 (b1 b2: B), a1 ≠ a2 →
                (if dec a1 a2 then b1 else b2) = b2.
Theorem update_neq : ∀ {B: Type} a2 a1 (val: B) m,
  a2 ≠ a1 →
  get_val (upd_val m a2 val) a1 = get_val m a1.
Theorem update_dom :
  ∀ {B:Type} m a (v: B), (il _ _ dec (upd_val m a v)) = (il _ _ dec m).
End Mappings.
Notation "## A" := (il _ _ _ A) (at level 10).
Notation "&& A" := (mp _ _ _ A) (at level 10).
Definition val_of {B:Type} (def:B) (opt_a : option B) :=
  match opt_a with
  | Some a ⇒ a
  | None ⇒ def
  end.
Lemma sub_mp_l :
  ∀ A (dec: Dec A) B b_def (m1 m2: Mapping A B dec) i,
    common _ dec (##m1) (##m2) = false →
    List.In i (##m1) →
    (&&(val_of (map_def A dec B b_def) (m_union _ _ m1 m2))) i = (&&m1) i.
Hint Resolve sub_mp_l.
Lemma in_not_in :
  ∀ (A:Type) (dec: Dec A) l1 l2 (i:A),
    common _ dec l1 l2 = false → List.In i l2 → ¬List.In i l1.
Lemma sub_mp_r :
```

```
  ∀ A (dec: Dec A) B b_def (m1 m2: Mapping A B dec) i,
    common _ dec (##m1) (##m2) = false →
    List.In i (##m2) →
    (&&(val_of (map_def A dec B b_def) (m_union _ _ m1 m2))) i = (&&m2) i.

Hint Resolve sub_mp_r.

Lemma union_dom :
  ∀ (A B1 B2: Type)
        (decA: Dec A)
        (map1 map1' md1: Mapping A B1 decA)
        (map2 map2' md2: Mapping A B2 decA),
    ##map1 = ##map2 → ##map1' = ##map2' →
    (common _ decA (##map1) (##map1') = false) →
    ##(val_of md1 (m_union _ decA map1 map1')) =
    ##(val_of md2 (m_union _ decA map2 map2')).

Hint Resolve union_dom.

Definition option_bop (A B: Type) (a1 a2: option A) (op: A→A→B) :=
match a1 with
  | Some v1 ⇒ match a2 with
                | Some v2 ⇒ Some (op v1 v2)
                | None ⇒ None
              end
  | None ⇒ None
end.

Fixpoint aeval (m : Mem) (a : aexp) : option nat :=
  match a with
  | ANum n ⇒ Some n
  | AId x ⇒ get_val id eq_id_dec m x
  | APlus a1 a2 ⇒ option_bop nat nat (aeval m a1) (aeval m a2)
                                     (fun n1 n2 ⇒ n1 + n2)
  | AMinus a1 a2 ⇒ option_bop nat nat (aeval m a1) (aeval m a2)
                                     (fun n1 n2 ⇒ n1 - n2)
  | AMult a1 a2 ⇒ option_bop nat nat (aeval m a1) (aeval m a2)
                                     (fun n1 n2 ⇒ n1 × n2)
  end.

Fixpoint beval (m : Mem) (b : bexp) : option bool :=
  match b with
  | BTrue ⇒ Some true
  | BFalse ⇒ Some false
  | BEq a1 a2 ⇒ option_bop nat bool (aeval m a1) (aeval m a2) beq_nat
  | BLe a1 a2 ⇒ option_bop nat bool (aeval m a1) (aeval m a2) ble_nat
  | BNot b1 ⇒ match (beval m b1) with
                | Some b ⇒ Some (negb b)
                | None ⇒ None
              end
  | BAnd b1 b2 ⇒ option_bop bool bool (beval m b1) (beval m b2) andb
  end.
```

```
Fixpoint update_n m (xl: list id) (vl: list nat) : Mem :=
  if beq_nat (length xl) (length vl) then
    match xl with
      | [] ⇒ m
      | h :: t ⇒ upd_val id eq_id_dec (update_n m t (tl vl)) h (hd 0 vl)
    end
  else
    m.

Lemma beq_nat_from_eq : ∀ m n, m = n → beq_nat m n = true.

Hint Resolve beq_nat_from_eq.

Lemma update_n_dom :
  ∀ m xl vl, (length xl = length vl) → ##(update_n m xl vl) = ##m.

Definition lst_distinct (l : list _) : Prop :=
    (∀ i j, 0 ≤ i < length l → 0 ≤ j < length l →
                  (nth i l (Id 0) = nth j l (Id 0)) → i = j).

Lemma hd_is_0th :
  ∀ (A:Type) (lst: list A) (def: A), length lst > 0 → (hd def lst) = (nth 0 lst def).

Hint Resolve hd_is_0th.

Lemma skip_head :
  ∀ i h l', (nth (i+1) (h :: l') (Id 0)) = nth i l' (Id 0).

Lemma distinct_cover :
  ∀ l' h, lst_distinct (h :: l') → lst_distinct l'.

Lemma distinct_neq :
  ∀ l' h, lst_distinct (h::l') →
                  (∀ i, 0 ≤ i < length l' → h ≠ nth i l' (Id 0)).

Lemma update_n_x :
  ∀ xl,
    lst_distinct xl →
    (∀ m i vl,
      length vl = length xl →
      0 ≤ i < length xl →
      List.In (nth i xl (Id 0)) (il id nat eq_id_dec m) →
      get_val id eq_id_dec (update_n m xl vl) (nth i xl (Id 0)) = Some (nth i vl 0)
    ).

Lemma update_n_x_same :
  ∀ m x xl vl,
    length vl = length xl → ¬ List.In x xl →
    get_val id eq_id_dec (update_n m xl vl) x = get_val id eq_id_dec m x.

Inductive action : Type :=
  | Act_LTau : action
  | Act_LOt : pch → list nat → action
  | Act_LIn : pch → list nat → action
  | Act_Eps : action
  | Act_Suspd : action
```

```
.

Definition get_pch alpha : pch :=
  match alpha with
  | Act_LTau ⇒ pch_def
  | Act_LOt c vl ⇒ c
  | Act_LIn c vl ⇒ c
  | _ ⇒ pch_def
  end.

Inductive polarity : Type :=
  | Pol_Ot : polarity
  | Pol_In : polarity
  | Pol_Bot : polarity
  .

Definition get_rho alpha :=
match alpha with
  | Act_LTau ⇒ Pol_Bot
  | Act_LOt _ _ ⇒ Pol_Ot
  | Act_LIn _ _ ⇒ Pol_In
  | _ ⇒ Pol_Bot
end.

Definition get_vl alpha :=
match alpha with
  | Act_LTau ⇒ []
  | Act_LOt _ vl ⇒ vl
  | Act_LIn _ vl ⇒ vl
  | _ ⇒ []
end.

Inductive all_some {A:Type} : list (option A) → Prop :=
  | All_Some_Nil : all_some []
  | All_Some_Ext : ∀ h t, h ≠ None → all_some t → all_some (h::t)
.

Variable pid_ids : pid → list id.

Fixpoint get_ids sys :=
  match sys with
  | SysProc j _ ⇒ pid_ids j
  | SysPar sys1 sys2 ⇒ get_ids sys1 ++ get_ids sys2
  | SysRes sys _ ⇒ get_ids sys
  end.

Print f_compose.

Reserved Notation "S '/' m '==>' '[' i ',' alpha ']' S' '/' m' "
                  (at level 40, m at level 39, i at level 39,
                   alpha at level 39, S' at level 39).

Inductive com_step : pid → (prod com Mem) → action →
                     (prod com Mem) → Prop :=
  | S_Skip : ∀ i m, ##m = pid_ids i →
```

```
                (SKIP) / m ==> [i,Act_LTau] (NIL) / m
  | S_Ass : ∀ i m x a val, ##m = pid_ids i →
                List.In x (##m) → (aeval m a = Some val) →
                ((x::=a)/m ==> [i,Act_LTau] (NIL)/upd_val id eq_id_dec m x val)
  | S_Out : ∀ i m c (al: list aexp), ##m = pid_ids i →
                all_some (map (aeval m) al) →
                (c!al)/m ==>
                      [i, (Act_LOt c (map (f_compose (val_of 0) (aeval m)) al))]
                      (NIL)/m
  | S_In : ∀ i m c (xl: list id) (vl: list nat), ##m = pid_ids i →
                lst_distinct xl → (∀ x, List.In x xl → List.In x (##m)) →
                length vl = length xl →
                (c??xl)/m ==> [i,(Act_LIn c vl)] (NIL)/ update_n m xl vl
  | S_SeqHalfway : ∀ i m S1 m' S1' S2 alpha,
      S1 / m ==> [i,alpha] S1' / m' → S1'≠ NIL →
      (S1 ;; S2) / m ==> [i,alpha] (S1' ;; S2) / m'
  | S_SeqFinished : ∀ i m S1 m' S2 alpha,
      S1 / m ==> [i,alpha] (NIL) / m' →
      (S1 ;; S2) / m ==> [i,alpha] (S2) / m'
  | S_IfTrue : ∀ i b m S1 S2, ##m = pid_ids i →
      (beval m b = Some true) →
      (IIF b THEN S1 ELSE S2 FII) / m ==> [i,Act_LTau] (S1) / m
  | S_IfFalse : ∀ i b m S1 S2, ##m = pid_ids i →
      (beval m b = Some false) →
      (IIF b THEN S1 ELSE S2 FII) / m ==> [i,Act_LTau] (S2) / m
  | S_WhileTrue : ∀ i b m S, ##m = pid_ids i →
      (beval m b = Some true) →
      (WHILE b DO S END) / m ==> [i,Act_LTau] (S ;; WHILE b DO S END) / m
  | S_WhileFalse : ∀ i b m S, ##m = pid_ids i →
      (beval m b = Some false) →
      (WHILE b DO S END) / m ==> [i,Act_LTau] (NIL) / m
  where " S '/' m '==>' '[' i ',' alpha ']' S' '/' m' " :=
  (com_step (i) (S,m) alpha (S',m')) : sem_scope.

Open Scope sem_scope.

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "S_Skip" | Case_aux c "S_Ass"
  | Case_aux c "S_Out" | Case_aux c "S_In"
  | Case_aux c "S_SeqHalfway" | Case_aux c "S_SeqFinished"
  | Case_aux c "S_IfTrue" | Case_aux c "S_IfFalse"
  | Case_aux c "S_WhileTrue" | Case_aux c "S_WhileFalse"].

Lemma inv_step_dom :
  ∀ i S S' m m' alpha,
    S/m ==> [i, alpha] S'/m' → (##m = pid_ids i ∧ ##m' = pid_ids i).

Hint Resolve inv_step_dom.

Definition act_from c rho vl :=
```

```
  match rho with
  | Pol_Ot ⇒ Act_LOt c vl
  | Pol_In ⇒ Act_LIn c vl
  | _ ⇒ Act_LTau
  end.

Lemma inv_step_act :
  ∀ i S S' m m' alpha,
    S/m ==> [i, alpha] S'/m' →
    ((∃ c rho vl, (rho = Pol_Ot ∨ rho = Pol_In) ∧ alpha = act_from c rho vl) ∨
     alpha = Act_LTau).

Lemma step_det :
  ∀ i S S' S'' m m' m'' alpha1 alpha2,
    S/m ==> [i, alpha1] S'/m' →
    S/m ==> [i, alpha2] S''/m'' →
    (get_pch alpha1 = get_pch alpha2 ∧ get_rho alpha1 = get_rho alpha2).

Definition SysMem := Mapping pid Mem eq_pid_dec.

Definition sysmem_def : SysMem := map_def pid eq_pid_dec Mem m_empty.

Definition u_sysmem (m1 m2: SysMem) := val_of sysmem_def (m_union _ _ m1 m2).

Definition modif_sysmem sysm i (m' : Mem) :=
  upd_val pid eq_pid_dec sysm i m'.

Definition upd_sysmem sysm i (sysm': SysMem) :=
  upd_val pid eq_pid_dec sysm i (&&sysm' i).

Definition upd_sm_lst sysm pidlst (sysm': SysMem) :=
  match pidlst with
    | [i] ⇒ upd_sysmem sysm i (sysm')
    | [i;j] ⇒ upd_sysmem (upd_sysmem sysm i (sysm')) j sysm'
    | _ ⇒ sysm
  end.

Definition eq_sysmem (m1 m2: SysMem) :=
  (##m1 = ##m2) ∧ (∀ i, List.In i (##m1) → (&&m1) i = (&&m2) i).

Lemma eq_sysmem_refl :
  ∀ m, eq_sysmem m m.

Lemma eq_sysmem_sym :
  ∀ m1 m2, eq_sysmem m1 m2 → eq_sysmem m2 m1.

Lemma eq_sysmem_trans :
  ∀ m1 m2 m3, eq_sysmem m1 m2 → eq_sysmem m2 m3 → eq_sysmem m1 m3.

Hint Resolve eq_sysmem_refl eq_sysmem_sym eq_sysmem_trans: eq_sysmem_eqv.

Definition conf : Type := prod Sys SysMem.

Reserved Notation "sys '//' m '==>' '[' pidlst ',' alpha ']' sys' '//' m' "
                  (at level 50, m at level 49, pidlst at level 49,
                   alpha at level 49, sys' at level 49).

Inductive sys_step : conf → list pid → action → conf → Prop :=
```

```
  | Sys_Step_One :
      ∀ i S m S' m' alpha sysm,
        S/m ==> [i, alpha] S'/m' →
        ##sysm = [i] → (&&sysm) i = m →
        (i:S)//sysm ===> [[i], alpha] (i:S')//(modif_sysmem sysm i m')
  | Sys_Step_Left :
      ∀ i sys1 m1 sys1' m1' sys2 m2 alpha sysm,
        sys1//m1 ===> [[i], alpha] sys1'//m1' →
        ##m2 = get_pids sys2 →
        (common _ eq_pid_dec (##m1) (##m2) = false) →
        eq_sysmem sysm (u_sysmem m1 m2) →
        (sys1 |P| sys2)//sysm ===> [[i], alpha]
                              (sys1' |P| sys2)//(upd_sysmem sysm i m1')
  | Sys_Step_Right :
      ∀ i sys1 m1 sys2 m2 sys2' m2' alpha sysm,
        sys2//m2 ===> [[i], alpha] sys2'//m2' →
        ##m1 = get_pids sys1 →
        (common _ eq_pid_dec (##m1) (##m2) = false) →
        eq_sysmem sysm (u_sysmem m1 m2) →
        (sys1 |P| sys2)//sysm ===> [[i], alpha]
                              (sys1 |P| sys2')//(upd_sysmem sysm i m2')
  | Sys_Step_Sync :
      ∀ sys1 sys1' sys2 sys2' m1 m1' m2 m2'
            i j c vl alpha1 alpha2 sysm,
        sys1//m1 ===> [[i], alpha1] sys1'//m1' →
        sys2//m2 ===> [[j], alpha2] sys2'//m2' →
        (common _ eq_pid_dec (##m1) (##m2) = false) →
        eq_sysmem sysm (u_sysmem m1 m2) →
        (alpha1 = Act_LOt c vl ∧ alpha2 = Act_LIn c vl ∨
         alpha1 = Act_LIn c vl ∧ alpha2 = Act_LOt c vl) →
        (sys1 |P| sys2)//sysm ===> [[i;j], Act_LTau]
        (sys1' |P| sys2')//(upd_sysmem (upd_sysmem sysm i m1') j m2')
  | Sys_Step_Res :
      ∀ sys sys' C m m' pidlst alpha sysm,
        sys//m ===> [pidlst, alpha] sys'//m' →
        eq_sysmem sysm m →
        (alpha = Act_LTau ∨
         (alpha ≠ Act_LTau ∧ ¬ Ensembles.In _ C (get_pch alpha))) →
        (sys\C)//sysm ===> [pidlst, alpha]
              (sys'\C)//(upd_sm_lst sysm pidlst m')
  where "sys '//' m '===>' '[' pidlst ',' alpha ']' sys' '//' m'" :=
        (sys_step (sys,m) pidlst alpha (sys',m')) : sem_scope.

Tactic Notation "step_cases_sys" tactic(first) ident(c) :=
  first;
  [ Case_aux c "Sys_Step_One" |
    Case_aux c "Sys_Step_Left" | Case_aux c "Sys_Step_Right" |
    Case_aux c "Sys_Step_Sync" | Case_aux c "Sys_Step_Res" ].
```

```
Lemma com_step_inv_act :
  ∀ S m S' m' alpha i,
    S/m ==> [i, alpha] S'/m' →
    (alpha = Act_LTau ∨
     (∃ c vl, alpha = Act_LOt c vl) ∨
     (∃ c vl, alpha = Act_LIn c vl)).

Lemma sys_step_inv_act :
  ∀ sys m sys' m' alpha pidlst,
    sys//m ===> [pidlst, alpha] sys'//m' →
    (alpha = Act_LTau ∨
     (∃ c vl, alpha = Act_LOt c vl) ∨
     (∃ c vl, alpha = Act_LIn c vl)).

Hint Resolve sys_step_inv_act.

Inductive sys_multi_step : conf → conf → Prop :=
  | Sys_step_refl : ∀ C, sys_multi_step C C
  | Sys_multi_step :
      ∀ sys sys' sys'' m m' m'' pidlst alpha,
        sys//m ===> [pidlst, alpha] sys''//m'' →
        sys_multi_step (sys'', m'') (sys', m') →
        sys_multi_step (sys, m) (sys', m').

Lemma sys_step_inv_pid :
  ∀ sys sys' (sysm sysm': SysMem) pidlst alpha,
    sys//sysm ===> [pidlst, alpha] sys'//sysm' →
    get_pids sys' = get_pids sys.

Lemma sys_step_in_pidlst :
  ∀ sys sys' sysm sysm' pidlst alpha,
    sys//sysm ===> [pidlst, alpha] sys'//sysm' →
    (∀ i, List.In i pidlst → List.In i (get_pids sys)).

Hint Resolve sys_step_inv_pid.

Lemma upd_sm_lst_dom :
  ∀ sysm pidlst m,
    ##(upd_sm_lst sysm pidlst m) = ## sysm.

Lemma sys_step_inv_mem :
  ∀ sys sys' (sysm sysm': SysMem) pidlst alpha,
    sys//sysm ===> [pidlst, alpha] sys'//sysm' →
    (##sysm = get_pids sys ∧ ##sysm' = get_pids sys').

Hint Resolve sys_step_inv_mem.

Lemma get_com_sub :
  ∀ sys1 sys2 i,
    (common _ eq_pid_dec (get_pids sys1) (get_pids sys2) = false) →
    (
      (List.In i (get_pids sys1) → get_com (sys1 |P| sys2) i = get_com sys1 i) ∧
      (List.In i (get_pids sys2) → get_com (sys1 |P| sys2) i = get_com sys2 i)
    ).
```

```
Lemma sys_proc_trans :
  ∀ sys sysm sys' sysm' alpha i,
    sys//sysm ===> [[i],alpha] sys'//sysm' →
    ∃ S S' m'',
      List.In i (get_pids sys) ∧
      S = get_com sys i ∧
      S/((&&sysm) i) ==> [i,alpha] S'/m''.

Lemma sys_step_det:
  ∀ sys1 sys2 m1 m2 i c vl alpha' m2' sys2',
    sys1 // m1 ===> [[i], Act_LOt c vl] sys2 // m2 →
    sys1 // m1 ===> [[i], alpha'] sys2' // m2' →
    get_pch alpha' = c.

Ltac inv_sub_cl :=
  match goal with
  | [id: (Ensembles.In _ (_ |U| _) _) ⊢ _] ⇒ (inversion id; subst; clear id)
  end.
```

B.3 Library Types

```
Require Export Semantics.

Inductive SecLev :=
| H : SecLev
| L : SecLev
.

Inductive Ordr : SecLev → SecLev → Prop :=
| LL : Ordr L L
| LH : Ordr L H
| HH : Ordr H H
.

Hint Resolve LL LH HH.

Definition lub l1 l2 :=
  match l1 with
  | H ⇒ H
  | L ⇒ l2
  end.

Hint Unfold lub.

Lemma L_le : ∀ l, Ordr L l.

Lemma le_H : ∀ l, Ordr l H.

Lemma ordr_refl : ∀ l, Ordr l l.

Lemma ordr_trans : ∀ l1 l2 l3, Ordr l1 l2 → Ordr l2 l3 → Ordr l1 l3.

Lemma ordr_lub_ll :
  ∀ l1 l2 l3, Ordr (lub l1 l2) l3 → Ordr l1 l3.
```

```
Lemma ordr_lub_lr :
  ∀ l1 l2 l3, Ordr (lub l1 l2) l3 → Ordr l2 l3.

Hint Resolve L_le le_H ordr_refl ordr_trans ordr_lub_ll ordr_lub_lr.

Lemma self_lub : ∀ l, l = lub l l.

Lemma le_lub_l : ∀ l1 l2, Ordr l1 (lub l1 l2).

Lemma le_lub_r : ∀ l1 l2, Ordr l2 (lub l1 l2).

Lemma both_le_lub_le :
  ∀ l1 l2 l3, Ordr l1 l3 → Ordr l2 l3 → Ordr (lub l1 l2) l3.

Hint Resolve le_lub_l le_lub_r self_lub both_le_lub_le.

Lemma lub_inc :
  ∀ l1 l2 l_sm l_lg,
    Ordr l_sm l_lg → Ordr (lub l1 l_sm) (lub l2 l_sm) →
    Ordr (lub l1 l_lg) (lub l2 l_lg).

Lemma lub_with_H : ∀ l, lub l H = H.

Hint Resolve lub_inc.

Definition Assertion := Mem → Prop.

Definition PVarVal := Assertion.
Definition PVarSec := Mapping id SecLev eq_id_dec.

Definition pvs_def := mkMapping id SecLev eq_id_dec [] (fun x ⇒ L).

Definition m'_def dom := mkMapping id nat eq_id_dec dom (fun x ⇒ 0).

Definition PVar := prod PVarSec Assertion.

Definition same_ele (A: Type) (l1 l2: list A) :=
  ∀ (a: A), (List.In a l1) ↔ (List.In a l2).

Definition PVarEnv := Mapping pid (Ensemble PVar) eq_pid_dec.

Definition empty_pe :=
  mkMapping pid (Ensemble PVar) eq_pid_dec ([]) (fun i ⇒ Empty_set _).

Theorem eq_ch_dec : ∀ c1 c2 : ch, {c1 = c2} + {c1 ≠ c2}.

Definition PChP := pch → SecLev.

Definition PChCVal := ch → (Ensemble nat).
Definition PChCSec := ch → SecLev.
Definition PChC := prod PChCSec PChCVal.

Notation " a 'from' A" := (Ensembles.In _ A a) (at level 20).
Definition PChEnv := sig (fun (CE: Ensemble PChC) ⇒ ∃ PcC, PcC from CE).

Variable PcP : PChP.
Variable PChCSet : PChEnv.

Theorem pcp_dec : ∀ c, {PcP c = H} + {PcP c = L}.

Definition bassn b : Assertion :=
  fun m ⇒ (beval m b = Some true).

Definition assn_sub x a phi : Assertion :=
  fun (m : Mem) ⇒
```

```
    phi (upd_val id eq_id_dec m x (val_of 0 (aeval m a))).
Definition assn_sub_n xl al phi : Assertion :=
  fun (m : Mem) ⇒ phi (update_n m xl (map (f_compose (val_of 0) (aeval m)) al)).

Notation "phi & x \-> a" := (assn_sub x a phi) (at level 10).

Definition all_assn xl phi : Assertion :=
  fun m ⇒
    (∀ (vl:list nat),
       ((assn_sub_n xl (map (fun v ⇒ ANum v) vl) phi) m))
.

Definition assert_implies (phi psi : Assertion) : Prop :=
  ∀ m, phi m → psi m.

Hint Unfold assert_implies.

Notation "phi -» psi" :=
  (assert_implies phi psi) (at level 80) : hoare_spec_scope.
Open Scope hoare_spec_scope.
Notation "phi «-» psi" :=
  (phi -» psi ∧ psi -» phi) (at level 80) : hoare_spec_scope.

Notation "lst '@' i" := (nth (i) lst 0) (at level 20).

Notation "lst 'a@' i" := (nth (i) lst (ANum 0)) (at level 20).
Notation "lst 'x@' i" := (nth (i) lst (Id 0)) (at level 20).

Notation "'|' lst '|'" := (length lst) (at level 20).

Definition val_sat_sub (m: Mem) (P: PVar) x a :=
  snd P (upd_val _ _ m x (val_of 0 (aeval m a))).

Definition val_sat_sub_n (m: Mem) (P: PVar) xl al :=
  snd P (update_n m xl (map (f_compose (val_of 0) (aeval m)) al)).

Definition vl_sat vl c (P: PChC) :=
  ∀ i, 1≤i≤|vl| → ((vl @ (i - 1)) from (snd P (Chan c i))).

Program Definition val_sat_c (alpha: action) (rho: polarity) (P: PChC) :=
  Ensembles.In _ PChCSet P ∧
  (rho = get_rho alpha → vl_sat (get_vl alpha) (get_pch alpha) P).

Definition num vl := map ANum vl.

Notation " x 'ido' P " := (List.In x (il id SecLev eq_id_dec (fst P)))
  (at level 20).

Definition secL (P: PVar) x := (mp id SecLev eq_id_dec (fst P) x).

Inductive sLift (P: PVar) X : SecLev → Prop :=
  | Lift_L : (∀ x, x from X → (secL P x) = L) → sLift P X L
  | Lift_H : ∀ x, x from X ∧ (secL P x) = H → sLift P X H
.

Definition and_assn phi b := (fun m ⇒ (phi m ∧ (bassn b m))).
Definition and_nssn phi b := (fun m ⇒ (phi m ∧ ~(bassn b m))).

Hint Unfold and_assn and_nssn.
```

```
Definition eval al j m := (val_of 0 (aeval m (al a@ (j-1)))).
Definition evxl xl j m := (val_of 0 (aeval m (AId (xl x@ (j-1))))).

Reserved Notation
  " K ',' i ',' X ',' l '\-' phi ',' S ',' psi '-:' X' ',' l' "
  (at level 60, i at level 59, X at level 59, l at level 59,
                phi at level 59, S at level 59, psi at level 59,
                X' at level 59, l' at level 59).

Definition contained_in X xl := ∀ (x:id), x from X → List.In x xl.

Inductive well_typed (K: Ensemble PVar) (i: pid) :
  (Ensemble id) → SecLev → Assertion → com → Assertion → (Ensemble id) → SecLev
→ Prop :=
  | TP_NIL : ∀ X l phi,
               contained_in X (pid_ids i) →
               K, i, X, l \- phi, (NIL), phi -: (Empty_set _), L
  | TP_SK : ∀ X l phi,
              contained_in X (pid_ids i) →
              K, i, X, l \- phi, (SKIP), phi -: (Empty_set _), L
  | TP_AS : ∀ X l phi x a,
              contained_in X (pid_ids i) →
              (∀ (P: PVar) m, ##m = pid_ids i →
                 (P from K) → snd P m → (phi & x \-> a) m →
                (∃ Q, Q from K ∧
                  (∀ l', sLift P (X |U| fva a) l' → Ordr l' (secL Q x)) ∧
                  (Ordr l (secL Q x)) ∧
                  (∀ x', x' ido P → x'≠x → (Ordr (secL P x') (secL Q x'))) ∧
                  val_sat_sub m Q x a
                )
              )->
              K, i, X, l \- (phi & x \-> a), (x::=a), (phi) -:
                                                  (Empty_set _), L
  | TP_SEQ : ∀ X l Y1 l1 Y2 l2 S1 S2 phi phi' psi,
               K, i, X, l \- phi, S1, phi' -: Y1, l1 →
               K, i, (X |U| Y1), (lub l l1) \- phi', S2, psi -: Y2, l2 →
               K, i, X, l \- phi, (S1 ;; S2), psi -: (Y1 |U| Y2), (lub l1 l2)
  | TP_IF : ∀ X l Y1 l1 Y2 l2 b S1 S2 phi psi,
              K, i, (X |U| fvb b), l \- (and_assn phi b), S1, psi -: Y1, l1 →
              K, i, (X |U| fvb b), l \- (and_nssn phi b), S2, psi -: Y2, l2 →
              K, i, X, l \- phi, (IIF b THEN S1 ELSE S2 FII), psi -:
                                          (Y1 |U| Y2 |U| fvb b), (lub l1 l2)
  | TP_WH : ∀ X Y l phi b S,
              K, i, Y, l \- (and_assn phi b), S, phi -: Y, l →
              Included _ (X |U| fvb b) Y →
              K, i, X, l \- phi, (WHILE b DO S END), (and_nssn phi b) -: Y, l
  | TP_OT : ∀ X l phi c al l',
              contained_in X (pid_ids i) →
              ((Ordr l (PcP c)) ∧ (Ordr (PcP c) l') ∧
               (∀ P m, ##m = pid_ids i →
```

```
                  P from K → (snd P m ∧ phi m) →
                  (∀ l, sLift P X l → Ordr l (PcP c)) ∧
                  (∃ Q PcC,
                     Q from K ∧ PcC from (proj1_sig PChCSet) ∧
                     (∀ x, x ido P → (Ordr (secL P x) (secL Q x))) ∧
                     (∀ j, 1≤j≤|al| →
                                (∀ l, sLift P (fva (al a@ (j-1))) l →
                                                  Ordr l (fst PcC (Chan c j)))) ∧
                     (snd Q m ∧
                      ∀ j, 1≤j≤|al| →
                                                  (eval al j m) from (snd PcC (Chan c
j)))
                   )
                  )
                 ) →
                 K, i, X, l \- phi, (c ! al), phi -: (Empty_set _), l'
  | TP_IN : ∀ X l phi xl c l',
                 contained_in X (pid_ids i) →
                 ((Ordr l (PcP c)) ∧ (Ordr (PcP c) l') ∧
                  (∀ P m, ##m = pid_ids i →
                    P from K → (snd P m ∧ all_assn xl phi m) →
                    ((∀ l, sLift P X l → Ordr l (PcP c)) ∧
                     ∀ PcC vl,
                        PcC from (proj1_sig PChCSet) →
                        |vl|=|xl| → vl_sat vl c PcC →
                        ∃ Q, Q from K ∧
                           (∀ x', x' ido P ∧ ~(List.In x' xl) →
                                       (Ordr (secL P x') (secL Q x'))) ∧
                           (∀ j, 1≤j≤|xl| →
                                       (Ordr (lub (fst PcC (Chan c j)) (PcP c))
                                             (secL Q (xl x@ (j-1))))) ∧
                           (val_sat_sub_n m Q xl (num vl))
                    )
                  )
                 ) →
                 K, i, X, l \- (all_assn xl phi), (c ?? xl), phi -: (Empty_set _),
l'
  | TP_ST : ∀ X Y l1 l2 X' Y' l1' l2' phi psi phi' psi' S,
                 contained_in Y (pid_ids i) →
                 K, i, X', l1' \- phi', S, psi' -: Y', l2' →
                 (phi -» phi') → (psi' -» psi) →
                 (Included _ X X') → (Included _ Y' Y) →
                 (Ordr l1 l1') → (Ordr l2' l2) →
                 K, i, X, l1 \- phi, S, psi -: Y, l2
  where " K ',' i ',' X ',' l '\-' phi ',' S ',' psi '-:' X' ',' l' "
        := (well_typed K i X l phi S psi X' l') : hoare_spec_scope.

Hint Constructors well_typed.
```

```
Tactic Notation "well_typed_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TP_NIL" | Case_aux c "TP_SK" | Case_aux c "TP_AS" |
    Case_aux c "TP_SEQ" | Case_aux c "TP_IF" | Case_aux c "TP_WH" |
    Case_aux c "TP_OT" | Case_aux c "TP_IN" | Case_aux c "TP_ST"
  ].
Definition m_dom m := (il id nat eq_id_dec m).
Definition m_val m x := (mp id nat eq_id_dec m x).
Definition eq_mdom m1 m2 : Prop :=
  ∀ (x: id), List.In x (m_dom m1) ↔ List.In x (m_dom m2).
Lemma nil_impl:
  ∀ K i X l1 Y l2 phi psi,
       K, i, X, l1 \- phi, (NIL), psi -: Y, l2 → (phi -» psi).
Lemma skip_impl:
  ∀ K i X l1 Y l2 phi psi,
    K, i, X, l1 \- phi, (SKIP), psi -: Y, l2 → (phi -» psi).
Lemma tp_nil:
  ∀ K i X l Y l' phi psi,
    phi -» psi →
    contained_in X (pid_ids i) →
    contained_in Y (pid_ids i) →
    K, i, X, l \- phi, (NIL), psi -: Y, l'.
Hint Resolve nil_impl tp_nil.
Lemma tp_ot_H_prc :
  ∀ c al K i X l phi psi X' l',
  PcP c = H →
  K, i, X, l \- phi, (c ! al), psi -: X', l' →
  K, i, X, H \- phi, (c ! al), psi -: X', l'.
Lemma tp_in_H_prc :
  ∀ c xl K i X l phi psi X' l',
    PcP c = H →
    K, i, X, l \- phi, (c ?? xl), psi -: X', l' →
    K, i, X, H \- phi, (c ?? xl), psi -: X', l'.
Lemma incl_X_Y: ∀ X Y b, Included id (X |U| fvb b) Y → Included id (Y |U| fvb b) Y.
Hint Resolve incl_X_Y.
Lemma eval_num_list : ∀ m vl, (map (f_compose (val_of 0) (aeval m)) (num vl)) = vl.
Hint Resolve eval_num_list.
Ltac pccfrom :=
  match goal with
    | [H : ∃ (_:PChC), _ from _ ⊢ _] ⇒
          inversion H as [ePcC H_from]; ∃ ePcC; simpl; try split; try assumption
  end.
Ltac rwaeval :=
  match goal with
```

```
    | [H : aeval ?m ?a = _ ⊢ _] ⇒
      match goal with
          [H' : context[val_of 0 (aeval m a)] ⊢ _] ⇒ rewrite H in H'
      end
  end.

Lemma contained_in_empty:
  ∀ X', contained_in (Empty_set id) X'.

Lemma contained_in_incl:
  ∀ X1 X2 Y, contained_in X1 X2 → Ensembles.Included _ Y X1 → contained_in Y
X2.

Hint Resolve contained_in_empty.

Lemma block_contained :
  ∀ K i X l X' l' phi S psi,
    K, i, X, l \- phi, S, psi -: X', l' →
    (contained_in X (pid_ids i) ∧ contained_in X' (pid_ids i)).

Lemma all_assn_all_vl :
∀ xl psi m vl, all_assn xl psi m → psi (update_n m xl vl).

Lemma subject_reduction_proc :
  ∀ K X l1 Y l2 phi S psi m S' m' i alpha P,
    K, i, X, l1 \- phi, S, psi -: Y, l2 →
    S/m ==>[i,alpha] S'/m' →
    phi m → P from K → snd P m →
    ((∃ PcC, val_sat_c alpha Pol_Ot PcC) ∧
     ((∃ PcC, val_sat_c alpha Pol_In PcC) →
      ∃ Q phi', K, i, X, l1 \- phi', S', psi -: Y, l2 ∧
                        phi' m' ∧ Q from K ∧ snd Q m')).

Definition AsnEnv := Mapping pid Assertion eq_pid_dec.
Definition asn_empty : AsnEnv :=
  mkMapping pid Assertion eq_pid_dec [] (fun (i:pid) ⇒ (fun m ⇒ False)).

Definition CompPSec := Mapping pid PVarSec eq_pid_dec.
Definition CompPVal := AsnEnv.
Definition CompP := prod CompPSec CompPVal.

Definition cps_empty : CompPSec :=
  mkMapping pid PVarSec eq_pid_dec [] (fun (i:pid) ⇒ pvs_def).

Definition u_cpp (CP1 CP2: CompP) :=
  (pair
     (val_of cps_empty (m_union _ _ (fst CP1) (fst CP2)))
     (val_of asn_empty (m_union _ _ (snd CP1) (snd CP2)))
  ).

Definition comp_sat (m: SysMem) (AE: AsnEnv) :=
  (##m = ##AE) ∧ (∀ i, List.In i (##m) → (&&AE i (&&m i))).

Definition comp_from (CP: CompP) (PE: PVarEnv) :=
  (##(fst CP) = ##PE) ∧ (##(snd CP) = ##PE) ∧
  (∀ i, List.In i (##PE) →
```

```
                ∃ P, (P from (&&PE i) ∧
                        pair (&&(fst CP) i) (&&(snd CP) i) = P)
  ).
Definition set_m_eq (m1 m2: Mem) (A: Ensemble PVar) : Prop :=
  eq_mdom m1 m2 ∧
  (∀ x, List.In x (m_dom m1) →
            (∀ P, P from A → secL P x = L) →
            m_val m1 x = m_val m2 x).
Definition set_act_eq c rho' vl1 vl2 A (rho: polarity) :Prop :=
  |vl1|=|vl2| ∧
  (∀ j,
     1≤j≤|vl1| → rho=rho' → (∀ (P: PChC), P from A → fst P (Chan c j) = L) →
       (vl1 @ (j-1)) = (vl2 @ (j-1))
  ).
Program Definition nip (K: Ensemble PVar) :=
  (∀ m1 m2, set_m_eq m1 m2 K
                → (∀ P, P from K → (snd P m1 ↔ snd P m2))) ∧
  (∀ c vl1 vl2,
    set_act_eq c Pol_Ot vl1 vl2 PChCSet Pol_Ot
    → (∀ PcC, PcC from PChCSet
                    → (val_sat_c (Act_LOt c vl1) Pol_Ot PcC ↔
                       val_sat_c (Act_LOt c vl2) Pol_Ot PcC)))
.
Definition pve_empty : PVarEnv :=
  mkMapping pid (Ensemble PVar) eq_pid_dec [] (fun (i:pid) ⇒ (Empty_set _)).
Reserved Notation
  " PVE '\-' Phi ',' sys ',' Psi "
  (at level 60, Phi at level 59, sys at level 59, Psi at level 59).
Inductive well_typed_sys : PVarEnv → AsnEnv → Sys → AsnEnv → Prop :=
  | TPSys_One :
      ∀ K Y l' S phi psi i,
        (∀ P, P from K → ##(fst P) = pid_ids i) →
        K, i, (Empty_set _), L \- phi, S, psi -: Y, l' → nip K →
        (mkMapping pid (Ensemble PVar) eq_pid_dec [i] (fun j ⇒ K))
          \- (mkMapping pid Assertion eq_pid_dec [i]
                        (fun j ⇒ if beq_nat j i then phi else (fun _ ⇒ True))),

              (i:S),
              (mkMapping pid Assertion eq_pid_dec [i]
                        (fun j ⇒ if beq_nat j i then psi else (fun _ ⇒ True)))
  | TPSys_Res :
      ∀ PVE Phi Psi Sys Omg,
        PVE \- Phi, Sys, Psi → PVE \- Phi, (Sys \ Omg), Psi
  | TPSys_Par :
      ∀ PVE1 Phi1 Sys1 Psi1 PVE2 Phi2 Sys2 Psi2,
        PVE1 \- Phi1, Sys1, Psi1 → PVE2 \- Phi2, Sys2, Psi2 →
```

```
        (common _ eq_pid_dec (get_pids Sys1) (get_pids Sys2) = false) →
        (val_of pve_empty (m_union pid eq_pid_dec PVE1 PVE2))
          \- (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2)),
               (Sys1 |P| Sys2),
               (val_of asn_empty (m_union pid eq_pid_dec Psi1 Psi2))
  where " PVE '\-' Phi ',' sys ',' Psi " := (well_typed_sys PVE Phi sys Psi).

Hint Constructors well_typed_sys.

Tactic Notation "well_typed_cases_sys" tactic(first) ident(c) :=
  first;
  [ Case_aux c "TPsys_One" | Case_aux c "TPsys_Res" | Case_aux c "TPsys_Par"
].

Lemma asnenv_dom :
  ∀ PVE sys Phi Psi,
    PVE \- Phi, sys, Psi → (##Phi = get_pids sys ∧ ##Psi = get_pids sys).

Lemma penv_dom :
  ∀ PVE sys Phi Psi, PVE \- Phi, sys, Psi → ##PVE = get_pids sys.

Hint Resolve asnenv_dom.
Hint Resolve penv_dom.

Lemma comp_from_one :
  ∀ i K CP,
    comp_from CP {| il := [i]; mp := fun _ : pid ⇒ K |} →
    (pair ((&& (fst CP)) i) ((&& (snd CP)) i)) from K.

Hint Resolve comp_from_one.

Lemma comp_sat_one_policy :
  ∀ i (CP: CompP) m,
    comp_sat {| il := [i]; mp := fun _ : pid ⇒ m |} (snd CP) →
    (&& (snd CP)) i m.

Hint Resolve comp_sat_one_policy.

Lemma comp_sat_split :
  ∀ (sysm1 sysm2: SysMem) (Phi1 Phi2: AsnEnv),
    ##sysm1 = ##Phi1 → ##sysm2 = ##Phi2 →
    (common _ eq_pid_dec (##sysm1) (##sysm2) = false) →
    comp_sat (u_sysmem sysm1 sysm2)
                    (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2)) →
    (comp_sat sysm1 Phi1 ∧ comp_sat sysm2 Phi2).

Lemma comp_sat_combine :
  ∀ (sysm1 sysm2: SysMem) (Phi1 Phi2: AsnEnv),
    ##sysm1 = ##Phi1 → ##sysm2 = ##Phi2 →
    (common _ eq_pid_dec (##sysm1) (##sysm2) = false) →
    comp_sat sysm1 Phi1 → comp_sat sysm2 Phi2 →
    comp_sat (u_sysmem sysm1 sysm2)
                    (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2)).

Hint Resolve comp_sat_split.

Definition restrict (CP: CompP) (pidl: list pid) : CompP :=
```

```
  (pair
     {| il:=pidl; mp:=&&(fst CP) |}
     {| il:=pidl; mp:=&&(snd CP) |}
  ).

Lemma comp_from_split :
  ∀ PVE1 PVE2 CP,
    (common _ eq_pid_dec (##PVE1) (##PVE2) = false) →
    comp_from CP (val_of pve_empty (m_union pid eq_pid_dec PVE1 PVE2)) →
    (comp_from (restrict CP (##PVE1)) PVE1 ∧
     comp_from (restrict CP (##PVE2)) PVE2).

Lemma comp_from_combine :
  ∀ CP1 CP2 PVE1 PVE2,
    (common _ eq_pid_dec (##PVE1) (##PVE2) = false) →
    comp_from CP1 PVE1 → comp_from CP2 PVE2 →
    comp_from (u_cpp CP1 CP2)
              (val_of pve_empty (m_union pid eq_pid_dec PVE1 PVE2)).

Hint Resolve comp_from_split comp_from_combine.

Lemma CP_dom_inv :
  ∀ (CP: CompP) (PVE1 PVE2: PVarEnv),
    (common _ eq_pid_dec (##PVE1) (##PVE2) = false) →
    comp_from CP (val_of pve_empty (m_union pid eq_pid_dec PVE1 PVE2)) →
    ##(snd CP) = (##PVE1 ++ ##PVE2).

Lemma trivial_br :
  ∀ (f: pid → bool) (CP: CompP),
        (fun (a : pid) ⇒ if (f a) then (&& (snd CP)) a else (&& (snd CP)) a) =
        (fun (a: pid) ⇒ (&& (snd CP)) a).

Hint Resolve CP_dom_inv.

Lemma val_sat_c_opp:
  ∀ c vl (PcC: PChC),
    val_sat_c (Act_LOt c vl) Pol_Ot PcC → val_sat_c (Act_LIn c vl) Pol_In PcC.

Hint Resolve val_sat_c_opp.

Lemma comp_sat_eq :
  ∀ m1 m2 Phi,
  eq_sysmem m1 m2 → comp_sat m1 Phi → comp_sat m2 Phi.

Hint Resolve comp_sat_eq.

Lemma comp_sat_one :
  ∀ sysm i m' (psi: Assertion) mp0,
    ##sysm = ([i]) → psi m' →
    mp0 i = psi →
    comp_sat (modif_sysmem sysm i m')
             {| il := [i]; mp := mp0 |}.

Hint Resolve comp_sat_one.

Lemma upd_sysmem_eq:
  ∀ sysm i m',
```

```
    List.In i (##sysm) →
    (&& m') i = (&& (upd_sysmem sysm i m')) i.

Hint Resolve upd_sysmem_eq.

Lemma upd_sysmem_neq:
  ∀ sysm i j m',
    i≠j → (&& sysm) i = (&& (upd_sysmem sysm j m')) i.

Hint Resolve upd_sysmem_neq.

Lemma comp_sat_combine_l :
  ∀ sysm m' Phi1 Phi2 Phi1' i,
    ##sysm = (##Phi1) ++ (##Phi2) →
    common _ eq_pid_dec (##Phi1) (##Phi2) = false →
    List.In i (##Phi1) →
    comp_sat sysm (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2)) →
    ##Phi1' = ##Phi1 →
    comp_sat m' Phi1' →
    (∀ j, List.In j (##m') → j≠i → (&&sysm) j = (&&m' j)) →
    comp_sat (upd_sysmem sysm i m')
             (val_of asn_empty (m_union pid eq_pid_dec Phi1' Phi2)).

Hint Resolve comp_sat_combine_l.

Lemma n_common_equiv :
  ∀ (A : Type) (dec: Dec A) (il1 il2: list A),
    common A dec il1 il2 = false ↔ (∀ a, (List.In a il1 ∧ List.In a il2) → False).

Lemma sym_n_common :
  ∀ (A : Type) (dec: Dec A) (il1 il2: list A),
    common A dec il1 il2 = false →
    common A dec il2 il1 = false.

Lemma comp_sat_combine_r :
  ∀ sysm m' Phi1 Phi2 Phi2' i,
    ##sysm = (##Phi1) ++ (##Phi2) →
    common _ eq_pid_dec (##Phi1) (##Phi2) = false →
    List.In i (##Phi2) →
    comp_sat sysm (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2)) →
    ##Phi2' = ##Phi2 →
    comp_sat m' Phi2' →
    (∀ j, List.In j (##m') → j≠i → (&&sysm) j = (&&m' j)) →
    comp_sat (upd_sysmem sysm i m')
             (val_of asn_empty (m_union pid eq_pid_dec Phi1 Phi2')).

Hint Resolve comp_sat_combine_r.

Lemma modif_pidlst:
  ∀ sys sysm sys' sysm' pidlst alpha,
  sys//sysm ===>[pidlst, alpha] sys'//sysm' →
  (∀ i, List.In i (get_pids sys) → ¬List.In i pidlst → (&&sysm) i = (&&sysm' i)).

Lemma pid_lst_len :
∀ sys sys' m m' pidlst alpha,
```

```
  sys // m ===> [pidlst, alpha] sys' // m' → (length pidlst = 1 ∨ length pidlst = 2).

Lemma upd_in_pidlst :
  ∀ sys sys' sysm pidlst m m' alpha i,
    List.In i pidlst →
    sys // m ===> [pidlst, alpha] sys' // m' →
    (∀ j, List.In j pidlst → List.In j (##sysm)) →
    (&& (upd_sm_lst sysm pidlst m')) i = (&& m') i.

Lemma comp_sat_upd :
  ∀ sys sys' m m' sysm Phi Phi' alpha pidlst,
   sys // m ===> [pidlst, alpha] sys' // m' →
   eq_sysmem sysm m →
   comp_sat sysm Phi →
   comp_sat m' Phi' →
   comp_sat (upd_sm_lst sysm pidlst m') Phi'.

Lemma sysm_proj_l:
  ∀ sysm m1 m2 j,
    common _ eq_pid_dec (##m1) (##m2) = false →
    eq_sysmem sysm (u_sysmem m1 m2) →
    List.In j (##m1) →
    (&& sysm) j = (&& m1) j.

Lemma sysm_proj_r:
  ∀ sysm m1 m2 j,
    common _ eq_pid_dec (##m1) (##m2) = false →
    eq_sysmem sysm (u_sysmem m1 m2) →
    List.In j (##m2) →
    (&& sysm) j = (&& m2) j.

Lemma comp_sat_same:
∀ (CP: CompP) (m1 m2 sysm: SysMem),
  common _ eq_pid_dec (##m1) (##m2) = false →
  eq_sysmem sysm (u_sysmem m1 m2) →
  comp_sat sysm (snd CP) →
  comp_sat (u_sysmem m1 m2) {| il := ##m1 ++ ##m2; mp := && (snd CP) |}.

Lemma sysm_dom_split:
  ∀ m1 m2 sysm,
    common pid eq_pid_dec (##m1) (##m2) = false →
    eq_sysmem sysm (u_sysmem m1 m2) →
    ##sysm = ##m1 ++ ##m2.

Hint Resolve sysm_dom_split.

Theorem subject_reduction_sys :
  ∀ PVE sys sys' (Phi Psi: AsnEnv) alpha pidlst
        (sysm sysm': SysMem) (CP: CompP),
    PVE \- Phi, sys, Psi →
    sys//sysm ===>[pidlst, alpha] sys'//sysm' →
    comp_sat sysm Phi →
    comp_from CP PVE →
```

```
    comp_sat sysm (snd CP) →
    ((∃ PcC, val_sat_c alpha Pol_Ot PcC) ∧
     ((∃ PcC, val_sat_c alpha Pol_In PcC) →
      ∃ (CQ: CompP) (Phi': AsnEnv),
        PVE \- Phi', sys', Psi ∧
        comp_sat sysm' Phi' ∧
        comp_from CQ PVE ∧
        comp_sat sysm' (snd CQ)
     )
    ).
```

B.4 Library Sec

```
Require Export Types.
Require Import Classical_Prop.

Definition state_eq (P:CompP) (PE:PVarEnv) sys1 m1 sys2 m2 :=
  comp_from P PE ∧
  get_pids sys1 = ##PE ∧ get_pids sys2 = ##PE ∧
  ##(fst P) = ##(snd P) ∧
  comp_sat m1 (snd P) ∧ comp_sat m2 (snd P) ∧
  ∀ i, List.In i (##PE) →
            ##(&&(fst P) i) = pid_ids i ∧
            ##(&&m1 i) = pid_ids i ∧ ##(&&m2 i) = pid_ids i ∧
            ∀ x, List.In x (pid_ids i) →
                      (&&(&&(fst P) i) x = L) →
                      &&(&&m1 i) x = &&(&&m2 i) x.

Definition eq_sysconf (C1 C2: conf) :=
  fst C1 = fst C2 ∧ eq_sysmem (snd C1) (snd C2).

Definition high_comm alpha :=
  ∃ c rho' vl, (rho' = Pol_Ot ∨ rho' = Pol_In) ∧ alpha = act_from c rho' vl ∧
                                PcP c = H.

Definition may_step C1 pidlst alpha C2 :=
  (sys_step C1 pidlst alpha C2) ∨
  (eq_sysconf C1 C2 ∧ alpha = Act_Eps) ∨
  (eq_sysconf C1 C2 ∧
   (∀ alpha' C2', sys_step C1 pidlst alpha' C2' → high_comm alpha') ∧
   alpha = Act_Suspd).

Inductive action_sk : Type :=
  | ActSk_LTau : action_sk
  | ActSk_LOt : pch → list nat → action_sk
  | ActSk_LIn : pch → action_sk
  | ActSk_Eps : action_sk
  | ActSk_Suspd : action_sk
.
```

```
Definition fill bta vl' : action :=
  match bta with
  | ActSk_LTau ⇒ Act_LTau
  | ActSk_LOt pc vl ⇒ Act_LOt pc vl
  | ActSk_LIn pc ⇒ Act_LIn pc vl'
  | ActSk_Eps ⇒ Act_Eps
  | ActSk_Suspd ⇒ Act_Suspd
  end.

Definition act_sk_from c rho vl :=
  match rho with
  | Pol_Ot ⇒ ActSk_LOt c vl
  | Pol_In ⇒ ActSk_LIn c
  | _ ⇒ ActSk_LTau
  end.

Definition get_rho' (bta: action_sk) :=
match bta with
  | ActSk_LTau ⇒ Pol_Bot
  | ActSk_LOt _ _ ⇒ Pol_Ot
  | ActSk_LIn _ ⇒ Pol_In
  | _ ⇒ Pol_Bot
end.

Definition get_pch' (bta: action_sk) : pch :=
  match bta with
  | ActSk_LTau ⇒ pch_def
  | ActSk_LOt c vl ⇒ c
  | ActSk_LIn c ⇒ c
  | _ ⇒ pch_def
  end.

Definition get_vl' (bta: action_sk) :=
match bta with
  | ActSk_LTau ⇒ []
  | ActSk_LOt _ vl ⇒ vl
  | ActSk_LIn _ ⇒ []
  | _ ⇒ []
end.

Program Definition val_sat_c' (bta: action_sk) (rho: polarity) (P: PChC) :=
  Ensembles.In _ PChCSet P ∧
  (rho = get_rho' bta → vl_sat (get_vl' bta) (get_pch' bta) P).

Definition act_sk_eq' c rho' vl1 vl2 (PcC: PChC) (rho: polarity) :Prop :=
  |vl1|=|vl2| ∧
  (∀ j,
     1≤j≤|vl1| → rho=rho' → fst PcC (Chan c j) = L →
       (vl1 @ (j-1)) = (vl2 @ (j-1))
  ).

Inductive act_eq : action → action → PChC → polarity → Prop :=
  | AE_HPrc : ∀ c rho rho' vl alpha PcC,
```

```
                    PcP c = H → (rho'=Pol_Ot ∨ rho'=Pol_In) →
                    alpha = act_from c rho' vl →
                    val_sat_c alpha rho PcC →
                    act_eq alpha Act_Eps PcC rho
  | AE_LPrc : ∀ c rho rho' vl1 vl2 alpha1 alpha2 PcC,
                    PcP c = L → (rho'=Pol_Ot ∨ rho'=Pol_In) →
                    alpha1 = act_from c rho' vl1 →
                    alpha2 = act_from c rho' vl2 →
                    val_sat_c alpha1 rho PcC →
                    val_sat_c alpha2 rho PcC →
                    set_act_eq c rho' vl1 vl2 (Singleton _ PcC) rho →
                    act_eq alpha1 alpha2 PcC rho
  | AE_TT : ∀ PcC rho,
                 act_eq Act_LTau Act_LTau PcC rho
  | AE_TS : ∀ PcC rho,
                 act_eq Act_LTau Act_Suspd PcC rho
  .

Inductive act_sk_eq : action → action_sk → PChC → polarity → Prop :=
  | ASE_HPrc : ∀ c rho rho' vl alpha PcC,
                    PcP c = H → (rho'=Pol_Ot ∨ rho'=Pol_In) →
                    alpha = act_from c rho' vl →
                    val_sat_c alpha rho PcC →
                    act_sk_eq alpha ActSk_Eps PcC rho
  | ASE_LPrc : ∀ c rho rho' vl1 vl2 alpha bta PcC,
                     PcP c = L → (rho'=Pol_Ot ∨ rho'=Pol_In) →
                     alpha = act_from c rho' vl1 →
                     bta = act_sk_from c rho' vl2 →
                     val_sat_c alpha rho PcC →
                     val_sat_c' bta rho PcC →
                     act_sk_eq' c rho' vl1 vl2 PcC rho →
                     act_sk_eq alpha bta PcC rho
  | ASE_TT : ∀ PcC rho,
                  act_sk_eq Act_LTau ActSk_LTau PcC rho
  | ASE_TS : ∀ PcC rho,
                  act_sk_eq Act_LTau ActSk_Suspd PcC rho
.

Lemma alpha_bta_tau:
  ∀ alpha bta PcCOt rho,
    act_sk_eq alpha bta PcCOt rho →
    alpha = Act_LTau →
    bta = ActSk_LTau ∨ bta = ActSk_Suspd.

Hint Resolve alpha_bta_tau.

Lemma alpha_bta_n_tau:
  ∀ alpha bta PcCOt,
    alpha ≠ Act_LTau →
    act_sk_eq alpha bta PcCOt Pol_Ot →
```

```
    ((∃ vl, bta = ActSk_LOt (get_pch alpha) vl) ∨
     bta = ActSk_LIn (get_pch alpha) ∨
     bta = ActSk_Eps).

Hint Resolve alpha_bta_n_tau.

Lemma bta_for_low_ot:
  ∀ c vl bta PcC,
    PcP c = L →
    act_sk_eq (Act_LOt c vl) bta PcC Pol_Ot →
    ∃ vl0, |vl| = |vl0| ∧ bta = ActSk_LOt c vl0.

Lemma bta_for_low_in:
  ∀ c vl bta PcC,
    PcP c = L →
    act_sk_eq (Act_LIn c vl) bta PcC Pol_Ot →
    bta = ActSk_LIn c.

Check (relation (prod conf CompP)).

Program Definition ca_bisim (PE: PVarEnv) (R: relation (prod conf CompP)) :=
  symmetric _ R ∧
  ∀ C1 C2 CP,
    R (pair C1 CP) (pair C2 CP) →
    state_eq CP PE (fst C1) (snd C1) (fst C2) (snd C2) ∧
    ∀ alpha pidlst C1',
      sys_step C1 pidlst alpha C1' →
      ∃ PcCOt bta,
        PcCOt from PChCSet ∧ act_sk_eq alpha bta PcCOt Pol_Ot ∧
        ∀ PcCIn vl,
          PcCIn from PChCSet →
          act_eq alpha (fill bta vl) PcCIn Pol_In →
          ∃ C2' CP',
            may_step C2 pidlst (fill bta vl) C2' ∧
            R (pair C1' CP') (pair C2' CP').

Definition ca_sec sys PE :=
  ∀ sysm1 sysm2 CP,
    state_eq CP PE sys sysm1 sys sysm2 →
    ∃ R, ca_bisim PE R ∧
                  R (pair (pair sys sysm1) CP) (pair (pair sys sysm2) CP).

Definition u_pe PE1 PE2 :=
  val_of empty_pe (m_union _ _ PE1 PE2).

Hint Unfold u_pe.

Inductive R_Res (R: relation (prod conf CompP)) : relation (prod conf CompP) :=
  | R_Res_intro: ∀ sys1 m1 sys2 m2 CP1 CP2 Omg m1' m2',
                   R (pair (pair sys1 m1) CP1) (pair (pair sys2 m2) CP2) →
                   eq_sysmem m1 m1' → eq_sysmem m2 m2' →
                   R_Res R (pair (pair (sys1\Omg) m1') CP1)
                                 (pair (pair (sys2\Omg) m2') CP2)
  .
```

```
Hint Constructors R_Res.

Lemma sym_res :
  ∀ (R: relation (prod conf CompP)), symmetric _ R → symmetric _ (R_Res R).

Hint Resolve sym_res.

Lemma act_sk_mismatch :
  ∀ vl,
  (fill ActSk_Suspd vl = Act_LTau ∨
   (∃ (c : pch) (vl0 : list nat), fill ActSk_Suspd vl = Act_LOt c vl0) ∨
   (∃ (c : pch) (vl0 : list nat), fill ActSk_Suspd vl = Act_LIn c vl0)) → False.

Hint Resolve act_sk_mismatch.

Lemma act_sk_mismatch' :
  ∀ vl,
    (fill ActSk_Eps vl = Act_LTau ∨
     (∃ (c : pch) (vl0 : list nat), fill ActSk_Eps vl = Act_LOt c vl0) ∨
     (∃ (c : pch) (vl0 : list nat), fill ActSk_Eps vl = Act_LIn c vl0)) → False.

Hint Resolve act_sk_mismatch'.

Lemma eq_sysmem_upd:
  ∀ m m' sysm i,
  eq_sysmem m sysm →
  eq_sysmem (upd_sysmem m i m') (upd_sysmem sysm i m').

Lemma relevant_mem :
  ∀ sys sys' m m' pidlst alpha,
  sys//m ===> [pidlst, alpha] sys'//m' →
  (∀ sysm, eq_sysmem m sysm →
                  ∃ sysm',
                    eq_sysmem m' sysm' ∧
                    sys//sysm ===> [pidlst, alpha] sys'//sysm').

Lemma upd_sysmem_self :
  ∀ (m: SysMem) i m',
    eq_sysmem m' m →
    upd_sysmem m i m' = m.

Lemma upd_sm_lst_self :
  ∀ (m: SysMem) lst m',
    (length lst = 1 ∨ length lst = 2) →
    eq_sysmem m' m →
    upd_sm_lst m lst m' = m.

Lemma upd_sysmem_eq_repl:
  ∀ m1 m2 m' i,
  eq_sysmem m1 m2 →
  eq_sysmem (upd_sysmem m1 i m') (upd_sysmem m2 i m').

Lemma upd_sysmem_ul :
  ∀ m1 m2 m' i,
  (common _ eq_pid_dec (##m1) (##m2) = false) →
  List.In i (##m1) →
```

```
  upd_sysmem (u_sysmem m1 m2) i m' = u_sysmem (upd_sysmem m1 i m') m2.

Lemma upd_sysmem_ur :
  ∀ m1 m2 m' i,
  (common _ eq_pid_dec (##m1) (##m2) = false) →
  List.In i (##m2) →
  upd_sysmem (u_sysmem m1 m2) i m' = u_sysmem m1 (upd_sysmem m2 i m').

Lemma eq_sysmem_eqv :
  ∀ m1 m2,
  m1 = m2 → eq_sysmem m1 m2.

Lemma sysmem_u_eq :
  ∀ m1 m1' m2 m2',
    (common _ eq_pid_dec (##m1) (##m2) = false) →
    eq_sysmem m1 m1' → eq_sysmem m2 m2' →
    eq_sysmem (u_sysmem m1 m2) (u_sysmem m1' m2').

Lemma st_eq_sysmem:
   ∀ sys1 sys2 m1 m2 m1' m2' Omg CP PE,
   state_eq CP PE sys1 m1 sys2 m2 → eq_sysmem m1 m1' → eq_sysmem m2 m2'
   → state_eq CP PE (sys1\Omg) m1' (sys2\Omg) m2'.

Lemma upd_sm_lst_in :
  ∀ m m1 pidlst i,
    (length pidlst = 1 ∨ length pidlst = 2) →
    List.In i pidlst →
    List.In i (## m) →
    (&& (upd_sm_lst m pidlst m1)) i = (&&m1 i).

Lemma upd_sm_lst_nin :
  ∀ m m1 pidlst i,
    ¬ List.In i pidlst →
    List.In i (##m)->
    (&& (upd_sm_lst m pidlst m1)) i = (&&m) i.

Lemma upd_sysmem_eqv :
  ∀ sys1 sys2 m1 m2 i alpha m1' m2',
    sys1//m1 ===> [[i], alpha] sys2//m2 →
    eq_sysmem m1 m1' →
    eq_sysmem m2' m2 →
    eq_sysmem m2 (upd_sysmem m1' i m2').

Lemma upd_sm_lst_eqv :
∀ sys sys' m1 m1' m' sysm' pidlst alpha,
  (length pidlst = 1 ∨ length pidlst = 2) →
  may_step (pair sys m1) pidlst alpha (pair sys' sysm') →
  eq_sysmem m1 m1' →
  eq_sysmem m' sysm' →
  eq_sysmem sysm' (upd_sm_lst m1' pidlst m').

Lemma compositionality_res :
  ∀ sys PE,
    ##PE = get_pids sys →
```

```
    ca_sec sys PE →
    ∀ Omg, ca_sec (sys\Omg) PE.

Hint Resolve compositionality_res.

Definition proj (sysm: SysMem) (pidl: list pid) : SysMem :=
  {| il:=pidl; mp:=&&sysm |}.

Lemma u_proj :
  ∀ sysm pidl1 pidl2,
    (common _ eq_pid_dec pidl1 pidl2 = false) →
    (pidl1 ++ pidl2 = (##sysm)) →
    u_sysmem (proj sysm pidl1) (proj sysm pidl2) = sysm.

Hint Resolve u_proj.

Lemma u_restrict:
  ∀ PE1 PE2 CP,
    common pid eq_pid_dec (##PE1) (##PE2) = false →
    comp_from CP (u_pe PE1 PE2) →
    val_of asn_empty
     (m_union pid eq_pid_dec
      (snd (restrict CP (##PE1))) (snd (restrict CP (##PE2))))
      = snd CP.

Hint Resolve u_restrict.

Lemma dom_combine :
∀ sysm CP PE1 PE2,
  comp_sat sysm (snd CP) →
  comp_from CP (u_pe PE1 PE2) →
  common pid eq_pid_dec (##PE1) (##PE2) = false →
  ##PE1 ++ ##PE2 = ##sysm.

Hint Resolve dom_combine.

Lemma state_eq_split:
  ∀ sys1 sys2 sysm1 sysm2 PE1 PE2 CP,
  ##PE1 = get_pids sys1 → ##PE2 = get_pids sys2 →
  common pid eq_pid_dec (##PE1) (##PE2) = false →
  state_eq CP (u_pe PE1 PE2) (sys1 |P| sys2) sysm1 (sys1 |P| sys2) sysm2 →
  (state_eq (restrict CP (##PE1)) PE1
    sys1 (proj sysm1 (##PE1)) sys1 (proj sysm2 (##PE1)) ∧
    state_eq (restrict CP (##PE2)) PE2
    sys2 (proj sysm1 (##PE2)) sys2 (proj sysm2 (##PE2))).

Hint Resolve state_eq_split.

Lemma st_eq_m_cp:
  ∀ PE1 PE2 CP sys1 sys2 sysm1 sysm2,
   state_eq CP (u_pe PE1 PE2) (sys1 |P| sys2) sysm1 (sys1 |P| sys2) sysm2
   → (##sysm1 = ##(fst CP) ∧ ##sysm1 = ##(snd CP)).

Lemma st_eq_m_sys:
  ∀ CP PE sys1 m1 sys2 m2,
    state_eq CP PE sys1 m1 sys2 m2 →
```

```
    (##m1 = get_pids sys1 ∧ ##m2 = get_pids sys2).

Lemma st_eq_m_pe:
  ∀ CP PE sys1 m1 sys2 m2,
    state_eq CP PE sys1 m1 sys2 m2 →
    (##m1 = ##PE ∧ ##m2 = ##PE).

Hint Resolve st_eq_m_cp st_eq_m_sys st_eq_m_pe : st_eq_inv_mem.

Definition oppo rho : polarity :=
  match rho with
  | Pol_Ot ⇒ Pol_In
  | Pol_In ⇒ Pol_Ot
  | Pol_Bot ⇒ Pol_Bot
  end
  .

Lemma u_sysmem_inv :
∀ m1 m2 m1' m2',
  u_sysmem m1 m2 = u_sysmem m1' m2' →
  ##m1 = ##m1' → ##m2 = ##m2' →
  (common _ eq_pid_dec (##m1) (##m2) = false) →
  (∀ i,
   (List.In i (##m1) → (&&m1) i = (&&m1') i) ∧
   (List.In i (##m2) → (&&m2) i = (&&m2') i)).
   SearchAbout "functional".

Lemma sysm_dom_combine :
  ∀ sys1 sys2 sysm1 sysm2,
    ##sysm1 = get_pids sys1 →
    ##sysm2 = get_pids sys2 →
    (common _ eq_pid_dec (##sysm1) (##sysm2) = false) →
    ##(u_sysmem sysm1 sysm2) = get_pids (sys1 |P| sys2).

Hint Resolve sysm_dom_combine.

Lemma mp_dom_inv :
  ∀ (A B:Type) (dec: Dec A) (m1 m2 def: Mapping A B dec) a,
    (common _ dec (##m1) (##m2) = false) →
    List.In a (##(val_of def (m_union _ dec m1 m2))) →
    (List.In a (##m1) ∨ List.In a (##m2)).

Inductive R_Par (R1 R2: relation (prod conf CompP)) : relation (prod conf CompP)
:=
  | R_Par_intro: ∀ sys11 sys21 sys12 sys22 m11 m21 m12 m22 CP1 CP2 ml mr,
                   R1 (pair (pair sys11 m11) CP1) (pair (pair sys21 m21) CP1) →
                   R2 (pair (pair sys12 m12) CP2) (pair (pair sys22 m22) CP2) →
                   eq_sysmem (u_sysmem m11 m12) ml →
                   eq_sysmem (u_sysmem m21 m22) mr →
                   R_Par R1 R2
                         (pair (pair (sys11 |P| sys12) ml) (u_cpp CP1 CP2))
                         (pair (pair (sys21 |P| sys22) mr) (u_cpp CP1 CP2)).

Lemma sym_par :
```

```
  ∀ R1 R2,
    symmetric _ R1 → symmetric _ R2 → symmetric (conf × CompP) (R_Par R1 R2).

Lemma state_eq_eqv:
  ∀ CP PE sys1 m1 m1' sys2 m2 m2',
  state_eq CP PE sys1 m1 sys2 m2 →
  eq_sysmem m1 m1' →
  eq_sysmem m2 m2' →
  state_eq CP PE sys1 m1' sys2 m2'.

Lemma state_eq_combine:
∀ CP1 CP2 PE1 PE2 sys11 sys21 sys12 sys22 m11 m21 m12 m22,
    common _ eq_pid_dec (##PE1) (##PE2) = false →
    state_eq CP1 PE1 sys11 m11 sys21 m21 →
    state_eq CP2 PE2 sys12 m12 sys22 m22 →
    state_eq (u_cpp CP1 CP2) (u_pe PE1 PE2)
             (sys11 |P| sys12) (u_sysmem m11 m12)
             (sys21 |P| sys22) (u_sysmem m21 m22).

Lemma eq_sysmem_proj_l:
  ∀ m1 m1' m2 m2',
    eq_sysmem (u_sysmem m1 m2) (u_sysmem m1' m2') →
    ##m1 = ##m1' → ##m2 = ##m2' →
    (common _ eq_pid_dec (##m1) (##m2) = false) →
    eq_sysmem m1 m1'.

Lemma eq_sysmem_proj_r:
  ∀ m1 m1' m2 m2',
    eq_sysmem (u_sysmem m1 m2) (u_sysmem m1' m2') →
    ##m1 = ##m1' → ##m2 = ##m2' →
    (common _ eq_pid_dec (##m1) (##m2) = false) →
    eq_sysmem m2 m2'.

Lemma eps_impl:
  ∀ sys1 m1 C2 i,
  may_step (pair sys1 m1) [i] Act_Eps C2 →
  eq_sysconf (pair sys1 m1) C2.

Lemma act_eq_oppo_eps :
  ∀ c vl PcC,
    act_sk_eq (Act_LOt c vl) ActSk_Eps PcC Pol_Ot →
    act_eq (Act_LIn c vl) Act_Eps PcC Pol_In.

Lemma act_eq_oppo_comm :
  ∀ c vl1 vl2 PcC,
    PcP c = L →
    act_sk_eq (Act_LOt c vl1) (ActSk_LOt c vl2) PcC Pol_Ot →
    act_eq (Act_LIn c vl1) (Act_LIn c vl2) PcC Pol_In.

Lemma no_step_eq_sysmem:
  ∀ sys1 sys2 i j sysm sysm',
    ¬ (∃ alpha' C2', sys_step (pair (sys1 |P| sys2) sysm) [i; j] alpha' C2') →
    eq_sysmem sysm sysm' →
```

```
    ¬ (∃ alpha' C2', sys_step (pair (sys1 |P| sys2) sysm') [i; j] alpha' C2').

Lemma compositionality_par :
  ∀ sys1 sys2 PE1 PE2,
    ##PE1 = get_pids sys1 → ##PE2 = get_pids sys2 →
    (common _ eq_pid_dec (##PE1) (##PE2) = false) →
    (ca_sec sys1 PE1 → ca_sec sys2 PE2 → ca_sec (sys1 |P| sys2) (u_pe PE1 PE2)).

Theorem compositionality :
  ∀ sys1 sys2 PE1 PE2,
    ##PE1 = get_pids sys1 → ##PE2 = get_pids sys2 →
    (common _ eq_pid_dec (##PE1) (##PE2) = false) →
    (
      (ca_sec sys1 PE1 → ∀ Omg, ca_sec (sys1\Omg) PE1) ∧
      (ca_sec sys1 PE1 → ca_sec sys2 PE2 →
       ca_sec (sys1 |P| sys2) (u_pe PE1 PE2))
    ).
```

B.5 Library Soundness

```
Require Import Sec.

Definition high K i phi S (m: Mem) (P: PVar) :=
  snd P m ∧ phi m ∧ P from K ∧
  ∃ X l X' l' psi,
    K, i, X, l \- phi, S, psi -: X', l' ∧
    (l = H ∨ ∃ x, x from X ∧ && (fst P) x = H) .

Definition low K i phi S (m: Mem) (P: PVar) :=
  snd P m ∧ phi m ∧ P from K ∧
  (∃ X l X' l' psi,
    K, i, X, l \- phi, S, psi -: X', l') ∧
  (∀ X l X' l' psi,
      K, i, X, l \- phi, S, psi -: X', l' →
      (l = L ∧ ∀ x, x from X → && (fst P) x = L)).

Inductive low_eq : PVarEnv → relation (prod conf CompP) :=
  | LEQ_LO :
        ∀ K i S P m1 m2 sysm1 sysm2 (CP: CompP) PE,
        (∃ phi, low K i phi S m1 P ∧ low K i phi S m2 P) →
        eq_sysmem sysm1 {|il:=[i]; mp:=(fun _ ⇒ m1)|} →
        eq_sysmem sysm2 {|il:=[i]; mp:=(fun _ ⇒ m2)|} →
        ##(fst CP) = [i] → &&(fst CP) i = fst P →
        ##(snd CP) = [i] → &&(snd CP) i = snd P →
        ##PE = [i] → &&PE i = K →
        state_eq CP PE (i:S) sysm1 (i:S) sysm2 →
        nip K →
        low_eq PE (pair (pair (i:S) sysm1) CP) (pair (pair (i:S) sysm2) CP)
  | LEQ_HI :
      ∀ K i S1 S2 P m1 m2 sysm1 sysm2 (CP: CompP) PE,
```

```
        (∃ phi1 phi2, high K i phi1 S1 m1 P ∧ high K i phi2 S2 m2 P) →
        eq_sysmem sysm1 {|il:=[i]; mp:=(fun _ ⇒ m1)|} →
        eq_sysmem sysm2 {|il:=[i]; mp:=(fun _ ⇒ m2)|} →
        ##(fst CP) = [i] → &&(fst CP) i = fst P →
        ##(snd CP) = [i] → &&(snd CP) i = snd P →
        ##PE = [i] → &&PE i = K →
        state_eq CP PE (i:S1) sysm1 (i:S2) sysm2 →
        nip K →
        low_eq PE (pair (pair (i:S1) sysm1) CP) (pair (pair (i:S2) sysm2) CP)
  .

Lemma dist_not_exists : ∀ (U:Type) (P : U → Prop),
  (∀ u, P u) → ¬ (∃ u, ¬ P u).

Lemma not_exists_dist :
  ∀ (U:Type) (P : U → Prop),
    ¬ (∃ u, ¬ P u) → (∀ u, P u).

Lemma not_forall_dist :
  ∀ (U:Type) (P : U → Prop),
    ¬ (∀ u, P u) → (∃ u, ¬ P u).

Lemma not_exists_dist' :
  ∀ (U:Type) (P: U → Prop),
    ¬ (∃ u, P u) → (∀ u, ¬ P u).

Lemma not_L_then_H :
  ∀ (l:SecLev), ¬ (l=L) → l=H.

Lemma not_low_then_high:
  ∀ K i X l (phi: Assertion) S psi X' l' m (P:PVar),
  (∀ x, x from X → List.In x (pid_ids i)) →
  P from K →
  snd P m →
  phi m →
  K, i, X, l \- phi, S, psi -: X', l' →
  ¬ low K i phi S m P →
  high K i phi S m P.

Lemma both_high_or_low :
  ∀ S (sysm1 sysm2: SysMem) (K:Ensemble PVar) i P (Phi: AsnEnv) PVE Psi,
    P from K →
    (&&PVE i) = K →
    PVE \- Phi, (i:S), Psi →
    ##sysm1 = [i] → ##sysm2 = [i] →
    (snd P (&&sysm1 i)) → (snd P (&&sysm2 i)) →
    (&&Phi i) (&&sysm1 i) → (&&Phi i) (&&sysm2 i) →
    (
      low K i (&&Phi i) S (&&sysm1 i) P ∧ low K i (&&Phi i) S (&&sysm2 i) P ∨
      high K i (&&Phi i) S (&&sysm1 i) P ∧ high K i (&&Phi i) S (&&sysm2 i) P
    ).

Lemma beq_nat_i_i : ∀ i, beq_nat i i = true.
```

```
Hint Rewrite beq_nat_i_i : trivial_eq.

Lemma both_high_or_low' :
  ∀ S (sysm1 sysm2: SysMem) (K:Ensemble PVar) i (P:PVar) phi psi X l X' l',
    (∀ P, P from K → ##(fst P) = pid_ids i) →
    nip K →
    P from K →
    K, i, X, l \- phi, S, psi -: X', l' →
    ##sysm1 = [i] → ##sysm2 = [i] →
    (snd P (&&sysm1 i)) → (snd P (&&sysm2 i)) →
    phi (&&sysm1 i) → phi (&&sysm2 i) →
    (
      low K i phi S (&&sysm1 i) P ∧ low K i phi S (&&sysm2 i) P ∨
      high K i phi S (&&sysm1 i) P ∧ high K i phi S (&&sysm2 i) P
    ).

Lemma state_eq_sym :
  ∀ CP PE sys1 sysm1 sys2 sysm2,
    state_eq CP PE sys1 sysm1 sys2 sysm2 →
    state_eq CP PE sys2 sysm2 sys1 sysm1.

Lemma sym_low_eq:
  ∀ PE, symmetric _ (low_eq PE).

Lemma high_nil :
  ∀ (K: Ensemble PVar) i (phi: Mem→Prop) (m: Mem) (P: PVar),
    snd P m → phi m → P from K → high K i phi (NIL) m P.

Lemma trivial_modif :
  ∀ sysm i m,
  ##sysm = [i] →
  eq_sysmem (modif_sysmem sysm i m)
      {| il := [i]; mp := fun _ : pid ⇒ m |}.

Lemma trivial_modif_simp :
  ∀ sysm i,
  ##sysm = [i] →
  eq_sysmem sysm (modif_sysmem sysm i ((&& sysm) i)) .

Lemma st_eq_alt_com :
  ∀ P PE sys1 m1 sys2 m2 sys1' sys2',
    state_eq P PE sys1 m1 sys2 m2 →
    get_pids sys1' = ##PE → get_pids sys2' = ##PE →
    state_eq P PE sys1' m1 sys2' m2.

Lemma assn_sat_sysm:
  ∀ sysm m i x a (phi: Assertion),
    eq_sysmem sysm {| il := [i]; mp := fun _ : pid ⇒ m |} →
    (phi & x \-> a) m →
    (phi & x \-> a) ((&& sysm) i).

Lemma aeval_some_dist :
  ∀ m a val,
  aeval m a = Some val →
```

```
  ∀ x, x from (fva a) → in_list _ eq_id_dec x (##m) = true.

Lemma in_dom_aeval_dist :
  ∀ m a,
    (∀ x, x from (fva a) → in_list _ eq_id_dec x (##m) = true) →
    ∃ val, aeval m a = Some val.

Lemma assn_sat_m_sysm:
  ∀ phi x a m sysm i val,
    eq_sysmem sysm {| il := [i]; mp := fun _ ⇒ m |} →
    aeval ((&& sysm) i) a = Some val →
    (phi & x \-> a) m →
    phi (upd_val id eq_id_dec ((&& sysm) i) x val).

Definition secL' (P: PVarSec) x := (mp id SecLev eq_id_dec P x).

Lemma update_mp_eq:
  ∀ (A B: Type) (dec: Dec A) (m: Mapping A B dec) a val,
    List.In a (il _ _ _ m) →
    (&& (upd_val _ dec m a val)) a = val.

Lemma update_mp_neq:
  ∀ (A B: Type) (dec: Dec A) (m: Mapping A B dec) a val a',
    List.In a (il _ _ _ m) → a ≠ a' →
    (&& (upd_val _ dec m a val)) a' = &&m a'.

Lemma eval_eq:
  ∀ m1 m2 a,
    (∀ x, x from (fva a) →
                List.In x (##m1) ∧ List.In x (##m2) ∧ &&m1 x = &&m2 x) →
    aeval m1 a = aeval m2 a.

Lemma all_fv_low:
  ∀ P X a,
    (∀ l', sLift P (X |U| fva a) l' → Ordr l' L) →
    (∀ x, x from (fva a) → secL P x = L).

Lemma all_fv_low_lift :
  ∀ P X, (∀ l, sLift P X l → Ordr l L) → (∀ x, x from X → secL P x = L).

Lemma ex_fv_high_lift :
  ∀ P X l, (∃ x, x from X ∧ secL P x = Types.H) →
                (∀ l', sLift P X l' → Ordr l' l) →
                l = Types.H.

Lemma set_m_eq_monotone:
  ∀ K1 K2 m1 m2,
    set_m_eq m1 m2 K1 → Included _ K1 K2 → set_m_eq m1 m2 K2.

Lemma eq_sysmem_upd:
  ∀ sysm i x val,
    ##sysm = [i] →
    eq_sysmem
      (modif_sysmem sysm i (upd_val id eq_id_dec ((&& sysm) i) x val))
      {| il := [i]; mp := fun _ : pid ⇒ upd_val id eq_id_dec ((&& sysm) i) x val |}.
```

```
Lemma from_comp_from :
  ∀ PE P i,
    ##PE = [i] →
    P from (&& PE) i →
    comp_from (pair {| il := [i]; mp := fun _ : pid ⇒ fst P |}
                    {| il := [i]; mp := fun _ : pid ⇒ snd P |}) PE.

Inductive well_alg_typed (K: Ensemble PVar) (i: pid) :
  (Ensemble id) → SecLev → Assertion → com → Assertion →
  (Ensemble id) → SecLev → Prop :=
  | ATP_NIL : ∀ X l phi psi X' l',
                contained_in X (pid_ids i) → contained_in X' (pid_ids i) →
                (phi -» psi) → well_alg_typed K i X l phi (NIL) psi X' l'
  | ATP_SK : ∀ X l phi psi X' l',
               contained_in X (pid_ids i) → contained_in X' (pid_ids i) →
               (phi -» psi) → well_alg_typed K i X l phi (SKIP) psi X' l'
  | ATP_AS : ∀ X l (phi psi psi': Assertion) x a X' l',
               contained_in X (pid_ids i) → contained_in X' (pid_ids i) →
               (∀ (P: PVar) m, ##m = pid_ids i →
                 (P from K) → snd P m → (psi & x \-> a) m →
                 (∃ Q, Q from K ∧
                   (∀ l', sLift P (X |U| fva a) l' → Ordr l' (secL Q x)) ∧
                   (Ordr l (secL Q x)) ∧
                   (∀ x', x' ido P → x'≠x →
                            (Ordr (secL P x') (secL Q x'))) ∧
                   val_sat_sub m Q x a
                 )
               )->
               (phi -» (psi & x \-> a)) →
               (psi -» psi') →
               well_alg_typed K i X l phi (x::=a) psi' X' l'
  | ATP_SEQ : ∀ l0 X l Y1 l1 X' l' Y2 l2 X'' l'' S1 S2 phi0 phi phi' phi'' psi,
                well_alg_typed K i X l phi S1 phi' Y1 l1 →
                well_alg_typed K i X' l' phi'' S2 psi Y2 l2 →
                Ordr l0 l → (phi0 -» phi) →
                Included _ (X |U| Y1) X' →
                Ordr (lub l l1) l' →
                (phi' -» phi'') →
                Included _ (Y1 |U| Y2) X'' →
                contained_in X'' (pid_ids i) →
                Ordr (lub l1 l2) l'' →
                well_alg_typed K i X l0 phi0 (S1;;S2) psi X'' l''
  | ATP_IF : ∀ X l X1 Y1 l1 X2 Y2 l2 X' l1' l2' l' b S1 S2
                 phi phi1 phi2 psi psi1 psi2,
               well_alg_typed K i X1 l1 phi1 S1 psi1 Y1 l1' →
               well_alg_typed K i X2 l2 phi2 S2 psi2 Y2 l2' →
               Included _ (X |U| fvb b) X1 →
               Included _ (X |U| fvb b) X2 →
```

```
                  ((and_assn phi b) -» phi1) →
                  ((and_nssn phi b) -» phi2) →
                  Ordr l l1 → Ordr l l2 →
                  (psi1 -» psi) → (psi2 -» psi) →
                  Included _ (Y1 |U| Y2 |U| fvb b) X' →
                  contained_in X' (pid_ids i) →
                  Ordr (lub l1' l2') l' →
                  well_alg_typed K i X l phi (IIF b THEN S1 ELSE S2 FII) psi X' l'
  | ATP_WH : ∀ X1 l1 X2 l2 Y l phi psi phi' b S,
                  Included _ (X1 |U| fvb b) Y →
                  Ordr l1 l →
                  (phi -» phi') →
                  well_alg_typed K i Y l (and_assn phi' b) S phi' Y l →
                  (and_nssn phi' b -» psi) →
                  Included _ Y X2 →
                  contained_in X2 (pid_ids i) →
                  Ordr l l2 →
                  well_alg_typed K i X1 l1 phi (WHILE b DO S END) psi X2 l2
  | ATP_OT : ∀ X l phi psi psi' c al X' l',
                  contained_in X (pid_ids i) → contained_in X' (pid_ids i) →
                 ((Ordr l (PcP c)) ∧ (Ordr (PcP c) l') ∧
                  (∀ P m, ##m = pid_ids i →
                   P from K → (snd P m ∧ psi m) →
                   (∀ l, sLift P X l → Ordr l (PcP c)) ∧
                   (∃ Q PcC,
                      Q from K ∧ PcC from (proj1_sig PChCSet) ∧
                      (∀ x, x ido P → (Ordr (secL P x) (secL Q x))) ∧
                      (∀ j, 1≤j≤|al| →
                                (∀ l, sLift P (fva (al a@ (j-1))) l →
                                                  Ordr l (fst PcC (Chan c j)))) ∧
                      (snd Q m ∧
                       ∀ j, 1≤j≤|al| →
                                                  (eval al j m) from (snd PcC (Chan c
j)))
                   )
                  )
                 ) →
                 (phi -» psi) → (psi -» psi') →
                 well_alg_typed K i X l phi (c!al) psi' X' l'
  | ATP_IN : ∀ X l (phi psi psi': Assertion) xl c X' l',
                  contained_in X (pid_ids i) → contained_in X' (pid_ids i) →
                 ((Ordr l (PcP c)) ∧ (Ordr (PcP c) l') ∧
                  (∀ P m, ##m = pid_ids i →
                   P from K → (snd P m ∧ all_assn xl psi m) →
                   ((∀ l, sLift P X l → Ordr l (PcP c)) ∧
                    ∀ PcC vl,
                      PcC from (proj1_sig PChCSet) → |vl|=|xl| →
```

```
                    vl_sat vl c PcC →
                    ∃ Q, Q from K ∧
                      (∀ x', x' ido P ∧ ~(List.In x' xl) →
                                  (Ordr (secL P x') (secL Q x'))) ∧
                      (∀ j, 1≤j≤|xl| →
                                  (Ordr (lub (fst PcC (Chan c j)) (PcP c))
                                        (secL Q (xl x@ (j-1))))) ∧
                              (val_sat_sub_n m Q xl (num vl))
                  )
                 )
                ) →
                (phi -» all_assn xl psi) → (psi -» psi') →
                well_alg_typed K i X l phi (c??xl) psi' X' l'
.

Tactic Notation "well_alg_typed_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ATP_NIL" | Case_aux c "ATP_SK" | Case_aux c "ATP_AS" |
    Case_aux c "ATP_SEQ" | Case_aux c "ATP_IF" | Case_aux c "ATP_WH" |
    Case_aux c "ATP_OT" | Case_aux c "ATP_IN" ].

Lemma slft_monotone:
  ∀ P X a X',
    sLift P (X |∪| fva a) Types.H → Included _ X X' → sLift P (X' |∪| fva a) Types.H.

Hint Resolve slft_monotone.

Lemma assn_sub_trans:
  ∀ phi rho psi x a,
    (phi -» rho & x \-> a) →
    (rho -» psi) →
    (phi -» psi & x \-> a).

Hint Resolve assn_sub_trans.

Lemma wtp_awtp' :
  ∀ K i X l X' l' S (phi psi: Assertion),
    K, i, X, l \- phi, S, psi -: X', l' →
    (∀ X1 l1 X1' l1' phi1 psi1,
       Included _ X1 X → Included _ X' X1' →
       contained_in X1' (pid_ids i) →
       Ordr l1 l → Ordr l' l1' →
       (phi1 -» phi) → (psi -» psi1) →
       well_alg_typed K i X1 l1 phi1 S psi1 X1' l1').

Lemma wtp_awtp:
  ∀ K i X l X' l' S phi psi,
    K, i, X, l \- phi, S, psi -: X', l' →
    well_alg_typed K i X l phi S psi X' l'.

Lemma awtp_wtp:
  ∀ K i X l S X' l' phi psi,
    well_alg_typed K i X l phi S psi X' l' →
```

```
    K, i, X, l \- phi, S, psi -: X', l'.

Lemma upd_eq_val_eq :
  ∀ (sysm : SysMem) x val0 (val: nat) (i:pid),
    List.In x (##(&&sysm i)) →
    upd_val id eq_id_dec ((&& sysm) i) x val0 =
    upd_val id eq_id_dec ((&& sysm) i) x val →
    val0 = val.

Lemma a_eval_same :
  ∀ m1 m2 a,
    ##m1 = ##m2 →
    (∀ y, y from fva a → &&m1 y = &&m2 y) →
    aeval m1 a = aeval m2 a.

Lemma b_eval_some_dist :
  ∀ m b val,
  beval m b = Some val →
  ∀ x, x from (fvb b) → in_list _ eq_id_dec x (##m) = true.

Lemma in_dom_beval_dist :
  ∀ m b,
  (∀ x, x from (fvb b) → in_list _ eq_id_dec x (##m) = true) →
  (beval m b = Some true ∨ beval m b = Some false).

Lemma b_eval_same :
  ∀ m1 m2 b,
    ##m1 = ##m2 →
    (∀ y, y from fvb b → &&m1 y = &&m2 y) →
    beval m1 b = beval m2 b.

Lemma eval_from_dist :
  ∀ al m c PcC,
    PcC from proj1_sig PChCSet →
    (∀ j, 1 ≤ j ≤ |al| →
                  eval al j m from snd PcC (Chan c j)) →
    val_sat_c
      (Act_LOt c (map (f_compose (val_of 0) (aeval m)) al))
      Pol_Ot PcC.

Lemma all_some_dist:
  ∀ m al,
    all_some (map (aeval m) al) →
    ∀ j, 1≤j≤|al| → ∃ v, aeval m (al a@ (j-1)) = Some v.

Lemma some_all_some:
  ∀ m al,
    (∀ j, 1≤j≤|al| → ∃ v, aeval m (al a@ (j-1)) = Some v) →
    all_some (map (aeval m) al).

Lemma eval_from_dist_sk :
∀ al m c PcC,
    PcC from proj1_sig PChCSet →
    (∀ j, 1 ≤ j ≤ |al| →
```

```
                    eval al j m from snd PcC (Chan c j)) →
    val_sat_c'
      (ActSk_LOt c (map (f_compose (val_of 0) (aeval m)) al))
      Pol_Ot PcC.

Lemma val_sat_c_c'_12:
  ∀ c m1 m2 PcC al,
    PcC from (proj1_sig PChCSet) →
    (∀ j, 1 ≤ j ≤ |al | → eval al j m1 from snd PcC (Chan c j)) →
    (val_sat_c (Act_LOt c (map (f_compose (val_of 0) (aeval m1)) al)) Pol_Ot PcC →
     val_sat_c (Act_LOt c (map (f_compose (val_of 0) (aeval m2)) al)) Pol_Ot PcC
    ) →
    val_sat_c'
      (ActSk_LOt c (map (f_compose (val_of 0) (aeval m2)) al)) Pol_Ot PcC.

Lemma act_eq_len_vl :
  ∀ c vl vl' PcC,
    act_eq (Act_LIn c vl) (Act_LIn c vl') PcC Pol_In → |vl| = |vl'|.

Lemma trivial_map_vl :
  ∀ m vl, (map (f_compose (val_of 0) (aeval m)) (num vl)) = vl.

Lemma update_n_in:
  ∀ xl x j vl m,
    lst_distinct xl → |vl| = |xl| → 1 ≤ j ≤ |xl| → x = (xl x@ (j - 1)) →
    List.In x (##m) → (&& (update_n m xl vl)) x = (vl @ (j - 1)).

Lemma low_chan_eq_val:
  ∀ c vl vl' j PcC,
    1 ≤ j ≤ |vl| →
    act_eq (Act_LIn c vl) (Act_LIn c vl') PcC Pol_In →
    fst PcC (Chan c j) = L →
    vl @ (j - 1) = vl' @ (j - 1).

Lemma update_n_nin:
  ∀ m x xl vl,
    ¬ List.In x xl → List.In x (##m) → |vl| = |xl| →
    (&& (update_n m xl vl)) x = &&m x.

Inductive upd_next : com → id → Prop :=
  | UN_As : ∀ x a, upd_next (x::=a) x
  | UN_In : ∀ c xl x, List.In x xl → upd_next (c??xl) x
  | UN_Seq : ∀ S1 S2 x, upd_next S1 x → upd_next (S1;;S2) x
.

Lemma n_upd_same :
  ∀ S m S' m' i alpha,
  S / m ==> [i, alpha] S' / m' →
  (∀ x, List.In x (##m) → ¬upd_next S x → &&m x = &&m' x).

Definition high' K i X l phi S psi X' l' m P :=
  snd P m ∧ phi m ∧ P from K ∧
  K, i, X, l \- phi, S, psi -: X', l' ∧
  (l = Types.H ∨ ∃ x, x from X ∧ secL P x = Types.H).
```

```
Definition low' K i phi S psi X' l' m P :=
  snd P m ∧ phi m ∧ P from K ∧
  ∃ X l, (K, i, X, l \- phi, S, psi -: X', l') ∧
  (∀ X l, K, i, X, l \- phi, S, psi -: X', l' →
                    (l=Types.L ∧ ∀ x, x from X → secL P x = Types.L)).

Lemma upd_next_dec :
  ∀ x S, {upd_next S x} + {¬upd_next S x}.

Lemma tp_impl_contained:
  ∀ K i X l phi S psi X' l',
    K, i, X, l \- phi, S, psi -: X', l' →
    contained_in X (pid_ids i).

Lemma tp_impl_contained':
  ∀ K i X l phi S psi X' l',
    K, i, X, l \- phi, S, psi -: X', l' →
    contained_in X' (pid_ids i).

Hint Resolve tp_impl_contained tp_impl_contained'.

Lemma high_step :
  ∀ K i phi S psi m P alpha S' m' X l X' l',
    (∀ P, P from K → ##(fst P) = (pid_ids i)) →
    high' K i X l phi S psi X' l' m P →
    S / m ==> [i, alpha] S' / m' →
    (
      (∀ c rho vl,
        (rho = Pol_Ot ∨ rho = Pol_In) →
        alpha = act_from c rho vl →
        PcP c = Types.H) ∧
      (∃ PcC, PcC from (proj1_sig PChCSet) ∧ val_sat_c alpha Pol_Ot PcC) ∧
      (∀ PcC,
         PcC from (proj1_sig PChCSet) →
         val_sat_c alpha Pol_In PcC →
         (∃ P',
            P' from K ∧
            (∀ x, (List.In x (##m) ∧ upd_next S x → secL P' x = Types.H) ∧
                          (List.In x (##m) ∧ ¬upd_next S x →
                           Ordr (secL P x) (secL P' x))) ∧
            ∃ phi', high' K i X l phi' S' psi X' l' m' P'))
    ).

Lemma nupd_eq :
  ∀ S S' m m' i alpha x,
    S / m ==> [i, alpha] S' / m' →
    List.In x (##m) →
    ¬upd_next S x →
    &&m' x = &&m x.

Lemma askeq_high_comm_eps:
  ∀ c rho vl alpha PcC,
    PcC from proj1_sig PChCSet →
```

```
    val_sat_c alpha Pol_Ot PcC →
    (rho = Pol_Ot ∨ rho = Pol_In) →
    PcP c = Types.H →
    alpha = act_from c rho vl →
    act_sk_eq alpha ActSk_Eps PcC Pol_Ot.

Lemma high'_high :
  ∀ K i X l phi S psi X' l' m P,
    high' K i X l phi S psi X' l' m P → high K i phi S m P.

Lemma same_ch_pol_dist:
  ∀ alpha1 alpha2 c rho vl,
    get_pch alpha1 = get_pch alpha2 →
    get_rho alpha1 = get_rho alpha2 →
    rho = Pol_Ot ∨ rho = Pol_In →
    alpha1 = act_from c rho vl →
    ∃ vl0 , alpha2 = act_from c rho vl0.

Lemma leq_hi_sim :
∀ PE i P0 K S1 S2 sysm1 m1 sysm2 m2 (CP: CompP),
  K = (&& PE) i → (∀ (P: PVar), P from K → ##(fst P) = pid_ids i) →
  ##PE = [i] → nip K →
  ##(fst CP) = [i] → (&& (fst CP)) i = fst P0 →
  ##(snd CP) = [i] → (&& (snd CP)) i = snd P0 →
  (∃ phi1 phi2 , high K i phi1 S1 m1 P0 ∧ high K i phi2 S2 m2 P0) →
  eq_sysmem sysm1 {|il:=[i]; mp:=(fun _ ⇒ m1)|} →
  eq_sysmem sysm2 {|il:=[i]; mp:=(fun _ ⇒ m2)|} →
  state_eq CP PE (i : S1) sysm1 (i : S2) sysm2 →
  (∀ alpha pidlst C1',
     sys_step (pair (i:S1) sysm1) pidlst alpha C1' →
     ∃ PcCOt bta ,
       PcCOt from proj1_sig PChCSet ∧
       act_sk_eq alpha bta PcCOt Pol_Ot ∧
       (∀ (PcCIn : PChC) (vl : list nat),
          PcCIn from proj1_sig PChCSet →
          act_eq alpha (fill bta vl) PcCIn Pol_In →
          ∃ C2' CP' ,
            may_step (pair (i:S2) sysm2) pidlst (fill bta vl) C2' ∧
            low_eq PE (pair C1' CP') (pair C2' CP'))).

Lemma high'_nil_nil :
  ∀ m1' m2' (P': PVar) X' l' K i (psi: Assertion),
    contained_in X' (pid_ids i) →
    P' from K → (snd P') m1' → (snd P') m2' → psi m1' → psi m2' →
    (
      high' K i X' H psi (NIL) psi X' l' m1' P' ∧
      high' K i X' H psi (NIL) psi X' l' m2' P'
    ).

Lemma get_trivial_upd :
∀ sysm i m,
```

```
  (##sysm)=[i] →
  (&& (modif_sysmem sysm i m)) i = m.

Hint Resolve get_trivial_upd.

Lemma tp_seq_tp_first :
  ∀ S1 S2 K i X l phi psi X' l',
    well_typed K i X l phi (S1;;S2) psi X' l' →
    ∃ psi' X'' l'' X''' l''',
      well_typed K i X l phi S1 psi' X'' l'' ∧
      well_typed K i X''' l''' psi' S2 psi X' l' ∧
      Included _ (X |U| X'') X''' ∧ Included _ X'' X' ∧
      Ordr (lub l l'') l''' ∧ Ordr l'' l'.

Lemma high'_fst_high'_seq :
  ∀ K i S' S2 m' X l X' l' X'' l'' X''' l''' phi psi psi' P',
      high' K i X l phi S' psi' X'' l'' m' P' →
      K, i, X''', l''' \- psi', S2, psi -: X', l' →
      Included _ (X |U| X'') X''' → Ordr (lub l l'') l''' →
      Included _ X'' X' → Ordr l'' l' →
      high' K i X l phi (S';; S2) psi X' l' m' P'.

Lemma high'_nil_high'_S :
  ∀ S2 K i X l phi psi psi' X' l' X'' l'' X''' l''' m' P',
    high' K i X l phi (NIL) psi' X'' l'' m' P' →
    K, i, X''', l''' \- psi', S2, psi -: X', l' →
    Included _ X X''' → Ordr l l''' →
    high' K i X l phi S2 psi X' l' m' P'.

Lemma act_eq_vl_sat_eps :
∀ c vl vl0 PcCIn,
act_eq (Act_LIn c vl) (fill ActSk_Eps vl0) PcCIn Pol_In → vl_sat vl c PcCIn.

Definition may_step_simp C1 pidlst alpha C2 :=
  (sys_step C1 pidlst alpha C2) ∨
  (eq_sysconf C1 C2 ∧ alpha = Act_Eps).

Lemma leq_lo_sim:
∀ PE i P0 K X l S psi X' l' sysm1 m1 sysm2 m2 (CP: CompP),
  K = (&& PE) i → (∀ (P: PVar), P from K → ##(fst P) = pid_ids i) →
  ##PE = [i] → nip K →
  ##(fst CP) = [i] → (&& (fst CP)) i = fst P0 →
  ##(snd CP) = [i] → (&& (snd CP)) i = snd P0 →
  (∃ phi, phi m1 ∧ phi m2 ∧ K, i, X, l \- phi, S, psi -: X', l' ) →
  eq_sysmem sysm1 {|il:=[i]; mp:=(fun _ ⇒ m1)|} →
  eq_sysmem sysm2 {|il:=[i]; mp:=(fun _ ⇒ m2)|} →
  state_eq CP PE (i : S) sysm1 (i : S) sysm2 →
  (∀ alpha pidlst S1' sysm1',
     sys_step (pair (i:S) sysm1) pidlst alpha (pair (i:S1') sysm1') →
     ∃ PcCOt bta,
       PcCOt from proj1_sig PChCSet ∧
       act_sk_eq alpha bta PcCOt Pol_Ot ∧
       (∀ (PcCIn : PChC) (vl : list nat),
```

```
        PcCIn from proj1_sig PChCSet →
        act_eq alpha (fill bta vl) PcCIn Pol_In →
        ∃ S2' sysm2' (CP': CompP),
          may_step_simp (pair (i:S) sysm2) pidlst (fill bta vl) (pair (i:S2') sysm2')

          ∧
          (
            (∃ phi1 phi2 X1 l1 X2 l2,
               Included _ (X1 |U| X') (X |U| X') ∧
               Included _ (X2 |U| X') (X |U| X') ∧
               Ordr (lub l1 l') (lub l l') ∧ Ordr (lub l2 l') (lub l l') ∧
               high' K i X1 l1 phi1 S1' psi X' l' (&&sysm1' i)
                     (pair (&&(fst CP') i) (&&(snd CP') i)) ∧
               high' K i X2 l2 phi2 S2' psi X' l' (&&sysm2' i)
                     (pair (&&(fst CP') i) (&&(snd CP') i)) ∧
               state_eq CP' PE (i:S1') sysm1' (i:S2') sysm2'
            ) ∨
            (∃ S' phi' X'' l'',
               Included _ (X'' |U| X') (X |U| X') ∧ Ordr (lub l'' l') (lub l l') ∧
               S' = S1' ∧ S' = S2' ∧
               K, i, X'', l'' \- phi', S', psi -: X', l' ∧
               phi' (&&sysm1' i) ∧ phi' (&&sysm2' i) ∧
               (&&(snd CP') i) (&&sysm1' i) ∧ (&&(snd CP') i) (&&sysm2' i) ∧
               state_eq CP' PE (i:S') sysm1' (i:S') sysm2'
            )
          )
      )
  ).

Lemma sound_one :
  ∀ S PE (Phi Psi: AsnEnv) i,
    (&&Phi) i = (fun _ ⇒ True) →
    PE \- Phi, (i:S), Psi →
    ca_sec (i:S) PE.

Definition true_assn : Assertion := fun _ ⇒ True.
Definition ase_true sys : AsnEnv :=
  mkMapping pid Assertion eq_pid_dec (get_pids sys) (fun _ ⇒ (fun _ ⇒ True)).

Lemma neq_beq_nat_f:
  ∀ (n1 n2: pid),
    n1 ≠ n2 → beq_nat n1 n2 = false.

Hint Resolve neq_beq_nat_f.

Lemma true_outside_dom:
  ∀ PE Phi sys Psi,
    wf sys →
    PE \- Phi, sys, Psi →
    (∀ i, ¬List.In i (get_pids sys) →
                (&&Phi i = true_assn ∧ &&Psi i = true_assn)).
```

```
Theorem soundness:
  ∀ PE sys,
    PE \- (ase_true sys), sys, (ase_true sys) →
    wf sys →
    ca_sec sys PE.
```

Bibliography

[AC12] Aslan Askarov and Stephen Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 308–322, 2012.

[ADZ+12] Torben Amtoft, Josiah Dodds, Zhi Zhang, Andrew W. Appel, Lennart Beringer, John Hatcliff, Xinming Ou, and Andrew Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *Principles of Security and Trust (POST)*, pages 369–389, 2012.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 113–124, 2009.

[AFG+11] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs*, pages 135–150. Springer, 2011.

[AFOTH06] Jim Alves-Foss, Paul W Oman, Carol Taylor, and W Scott Harrison. The mils architecture for high-assurance embedded systems. *International journal of embedded systems*, 2(3-4):239–247, 2006.

[Aga00] Johan Agat. Transforming out timing leaks. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[AH89] Luca Aceto and Matthew Hennessy. Towards action-refinement in process algebras. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 138–145, 1989.

[Ald01] Alessandro Aldini. Probabilistic information flow in a process algebra. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, pages 152–168, 2001.

[Ald06] Alessandro Aldini. Classification of security properties in a linda-like process algebra. *Sci. Comput. Program.*, 63(1):16–38, 2006.

[AM11] Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), 2011.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[Apt84] Krzysztof R. Apt. Ten years of Hoare's logic: A survey part ii: Nondeterminism. *Theor. Comput. Sci.*, 28:83–109, 1984.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, January 1980.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[ATRS15] Daniel Adam, Sergey Tverdyshev, Carsten Rolfes, and Timo Sandmann. Two architecture approaches for mils systems in mobility domains (automobile, railway and avionik). EURO-MILS, 2015.

[Bab00] Istvan Babcsanyi. Equivalence of mealy and moore automata. *Acta Cybern.*, 14(4):541–552, 2000.

[Bal13] Musard Balliu. A logic for information flow analysis of distributed programs. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*, pages 84–99, 2013.

[BBJ13] Frédéric Besson, Nataliia Bielova, and Thomas Jensen. Hybrid information flow monitoring against web tracking. In *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, pages 240–254. IEEE, 2013.

[BC02] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. springer, 2004.

[BCF08] Chiara Braghin, Agostino Cortesi, and Riccardo Focardi. Information flow security in boundary ambients. *Inf. Comput.*, 206(2-4):460–489, 2008.

[BDG11] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011*, page 6, 2011.

[BDR11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[BFG+14] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL '14*, pages 193–206, 2014.

[BFPR03a] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi. Bisimulation and unwinding for verifying possibilistic security properties. In *VMCAI 2003*, pages 223–237, 2003.

[BFPR03b] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi. Refinement operators and information flow security. In *(SEFM 2003)*, pages 44–53, 2003.

[BJSB11] Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal. Verifying object-oriented programs with higher-order separation logic in coq. In *Interactive Theorem Proving - Second International Conference, ITP 2011*, pages 22–38, 2011.

[BL76] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Computer Science Technical Report ESD-TR-75-306. The Mitre Corporation, 1976.

[BN04] Gilles Barthe and Leonor Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 13–22. ACM, 2004.

[BPR04] Annalisa Bossi, Carla Piazza, and Sabina Rossi. Modelling downgrading in information flow security. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 187, 2004.

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *Programming Languages and Systems*, pages 125–140. Springer, 2007.

[BPR13] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.

[BRRS10] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[BS06] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *ESOP 2006, Held as Part of ETAPS 2006*, pages 180–196, 2006.

[BS10] Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. In *37'th POPL*, pages 431–444. ACM, 2010.

[BvDS15] Niklas Broberg, Bart van Delft, and David Sands. The anatomy and facets of dynamic policies. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015*, pages 122–136, 2015.

[Cas07] Ilaria Castellani. State-oriented noninterference for CCS. *Electr. Notes Theor. Comput. Sci.*, 194(1):39–60, 2007.

[CCDR10] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In *CONCUR 2010*, 2010.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 354–368, 2008.

[CF01] Agostino Cortesi and Riccardo Focardi. Information flow security in mobile ambients. *Electr. Notes Theor. Comput. Sci.*, 54:58–68, 2001.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *POST 2014, Held as Part of ETAPS 2014, Grenoble, France*, pages 265–284, 2014.

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 57–70, 2010.

[CH08] David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008*, pages 50–66, 2008.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[CHM05] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *J. Log. and Comput.*, 15(2), April 2005.

[CHRS14] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, page 30, 2014.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 198–209, 2004.

[CM06] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 242–256, 2006.

[Coh77] Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139, 1977.

[CPA] The Coq Proof Assistant. Webpage: http://coq.inria.fr.

[CR05] Silvia Crafa and Sabina Rossi. A theory of noninterference for the pi-calculus. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, pages 2–18, 2005.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[Dam06] Mads Dam. Decidability and proof systems for language-based noninterference relations. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 67–78, 2006.

[DD77] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DNFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *Software Engineering, IEEE Transactions on*, 24(5):315–330, 1998.

[DNH84] Rocco De Nicola and Matthew CB Hennessy. Testing equivalences for processes. *Theoretical computer science*, 34(1):83–133, 1984.

[DP02] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 109–124, 2010.

[Dwo06] Cynthia Dwork. Differential privacy. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, pages 1–12, 2006.

[EGP08] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Pasareanu. Assume-guarantee verification for interface automata. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Proceedings*, pages 116–131, 2008.

[Fen74] Jeffrey Stewart Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[FG01] Riccardo Focardi and Roberto Gorrieri. Classification of security properties. In R Focardi and R Gorrieri, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*, pages 331–396. Springer Berlin Heidelberg, 2001.

[FGM00] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Information flow analysis in a discrete-time process algebra. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00*, pages 170–184, 2000.

[Flo10] FlowSafe. Webpage: https://wiki.mozilla.org/FlowSafe, 2010.

[FR06] Riccardo Focardi and Sabina Rossi. Information flow security in dynamic contexts. *Journal of Computer Security*, 14(1):65–110, 2006.

[FRS05] Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005*, pages 299–315, 2005.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Proceedings*, pages 163–179, 2013.

[GLMS14a] Sylvia Grewe, Alexander Lux, Heiko Mantel, and Jens Sauer. A formalization of declassification with what-and-where-security. *Archive of Formal Proofs*, 2014, 2014.

[GLMS14b] Sylvia Grewe, Alexander Lux, Heiko Mantel, and Jens Sauer. A formalization of strong security. *Archive of Formal Proofs*, 2014, 2014.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the 17th annual ACM symposium on Theory of computing*, pages 291–304. ACM, 1985.

[GMS14] Sylvia Grewe, Heiko Mantel, and Daniel Schoepe. A formalization of assumptions and guarantees for compositional noninterference. *Archive of Formal Proofs*, 2014, 2014.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, page 333, 2007.

[HBBS14] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: tracking information flow in javascript and its apis. In *Symposium on Applied Computing, SAC 2014*, pages 1663–1671, 2014.

[HHOAF05] W Scott Harrison, Nadine Hanebutte, P Oman, and Jim Alves-Foss. The mils architecture for a secure global information grid. *Crosstalk: The Journal of Defense Software Engineering*, 18(10):20–24, 2005.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[HS12] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347. 2012.

[HY07] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 273–284, 2008.

[IPA] The Isabelle Proof Assistant. Webpage: https://isabelle.in.tum.de.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, June 1981.

[KK07] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Kob05] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005.

[KT09] Maxwell N. Krohn and Eran Tromer. Noninterference for a practical difc-based operating system. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20*, pages 61–76, 2009.

[LC13] Luísa Lourenço and Luís Caires. Information flow analysis for valued-indexed data security compartments. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013*, pages 180–198, 2013.

[LC15] Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328. ACM, 2015.

[LM08] Alexander Lux and Heiko Mantel. Who can declassify? In *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers*, pages 35–49, 2008.

[LNN15] Ximeng Li, Flemming Nielson, and Hanne Riis Nielson. Factorization of behavioral integrity. In *Computer Security - ESORICS 2015, 20th European Symposium on Research in Computer Security. Proceedings*, pages 500–519, 2015.

[LNNF15] Ximeng Li, Flemming Nielson, Hanne Riis Nielson, and Xinyu Feng. Disjunctive information flow for communicating processes. In *Proceedings of the 10th International Symposium on Trustworthy Global Computing. Springer*, 2015.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MAC12] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-sensitive security. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 881–893, 2012.

[Man01] Heiko Mantel. Preserving information flow properties under refinement. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 78–91. IEEE, 2001.

[Man03] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Juli 2003.

[Man11] Heiko Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 605–607. 2011.

[MB09] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.

[MC12] Stefan Muller and Stephen Chong. Towards a practical secure concurrent language. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012*, pages 57–74, 2012.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[Mil89] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall, 1989.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, October 1997.

[MPP13] Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. A theory of information-flow labels. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 3–17. IEEE Computer Society, 2013.

[MPTB12] Kevin Müller, Michael Paulitsch, Sergey Tverdyshev, and Holger Blasum. MILS-related information flow control in the avionic domain: A view on security-enhancing software architectures. In *Dependable Systems and Networks Workshops (DSN-W)*, pages 1–6. IEEE, 2012.

[MR07] Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In *ESOP 2007, Held as Part of ETAPS 2007*, pages 141–156, 2007.

[MS01] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Computer Security Foundations Workshop, IEEE*, pages 0126–0126. IEEE Computer Society, 2001.

[MS03] Heiko Mantel and Andrei Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, 2003.

[MS10] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security. Proceedings*, pages 116–133, 2010.

[MS15] Heiko Mantel and Artem Starostin. Transforming out timing leaks, more or less. In *Proceedings of the 20th European Symposium on Research*, 2015.

[MSS11] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 218–232. IEEE, 2011.

[Mye99] Andrew C Myers. *Mostly-static decentralized information flow control*. PhD thesis, Massachusetts Institute of Technology, 1999.

[MZ08] Ron van der Meyden and Chenyi Zhang. Information flow in systems with schedulers. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 301–312. IEEE, 2008.

[Nie85] Flemming Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):359–379, 1985.

[NN13] Hanne Riis Nielson and Flemming Nielson. Safety versus security in the quality calculus. In *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, 2013.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005.

[NNL15a] Flemming Nielson, Hanne Riis Nielson, and Ximeng Li. Limitations of non-interference. In *Proceedings of the 27th Nordic Workshop on Programming Theory, NWPT 2015*, 2015.

[NNL15b] Hanne Riis Nielson, Flemming Nielson, and Ximeng Li. Hoare logic for disjunctive information flow. In *Programming languages with applications to biology and security - Essays dedicated to Pierpaolo Degano for his 65th birthday*, 2015.

[NNV12] Hanne Riis Nielson, Flemming Nielson, and Roberto Vigo. A calculus for quality. In *Formal Aspects of Component Software, 9th International Symposium, FACS 2012*, pages 188–204, 2012.

[OCC06] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*, pages 190–201, 2006.

[OL] The OCaml Language. Webpage: http://ocaml.org/.

[Org12] Common Criteria Sponsoring Organizations. Common criteria for information technology security evaluation. version 3.1, revision 4. Webpage: https://www.commoncriteriaportal.org/cc/, 2012.

[Orl11] Claudio Orlandi. Is multiparty computation any good in practice? In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011*, pages 5848–5851, 2011.

[PCG+] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. Webpage: http://www.cis.upenn.edu/ bcpierce/sf/current/index.html.

[PHN13] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Noninterfering schedulers - when possibilistic noninterference implies probabilistic noninterference. In *Algebra and Coalgebra in Computer Science, CALCO 2013*, pages 236–252, 2013.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Pot02] François Pottier. A simple view of type-secure information flow in the p-calculus. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002)*, pages 320–330, 2002.

[RG01] Arend Rensink and Roberto Gorrieri. Vertical implementation. *Inf. Comput.*, 170(1):95–133, 2001.

[RH13] A. W. Roscoe and Jian Huang. Checking noninterference in timed CSP. *Formal Asp. Comput.*, 25(1):3–35, 2013.

[RHS12] Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing interactive programs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA*, pages 293–307, 2012.

[Ros95] A. W. Roscoe. Csp and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, SP '95, pages 114–, Washington, DC, USA, 1995. IEEE Computer Society.

[RS09] Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *J. Log. Algebr. Program.*, 78(7):593–618, 2009.

[RS13] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 33–48, 2013.

[RS14] Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 277–292, 2014.

[Rus08] John Rushby. Separation and integration in mils (the mils constitution). *Computer Science Laboratory SRI International, Technical Report SRI-CSL-08-XX*, 2008.

[Sab03] Andrei Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003.*, pages 260–274, 2003.

[SES] SESAMO, a European Artemis project. Webpage: http://sesamo-project.eu/.

[SM02] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In Manuel V. Hermenegildo and German Puebla, editors, *Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer Berlin Heidelberg, 2002.

[SM03a] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SM03b] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, pages 174–191, 2003.

[Smi06] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009*, pages 288–302, 2009.

[SRMM12] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. Disjunction category labels. In *Information Security Technology for Applications*, pages 223–239. Springer, 2012.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00*, pages 200–214, 2000.

[SS09] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[Sut86] D. Sutherland. A model of information. In *Proceedings of the 9th National Computer Security Conference*, 1986.

[SV98] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98*, pages 355–364, 1998.

[SW01] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

[TA05] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005*, pages 352–367, 2005.

[Tea14] The Coq Developmnet Team. Reference manual of the coq proof assistant, 2014.

[TLHL11] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14, 2011.

[TNH06] Terkel K. Tolstrup, Flemming Nielson, and René Rydhof Hansen. Locality-based security policies. In *Formal Aspects in Security and Trust, Fourth International Workshop, FAST 2006*, pages 185–201, 2006.

[Tur50] Alan M Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

[VB03] L. Von Bertalanffy. *General System Theory: Foundations, Development, Applications*. G. Braziller, 2003.

[vBV11] Steffen van Bakel and Maria Grazia Vigliotti. Note on a simple type system for non-interference. *CoRR*, abs/1109.4843, 2011.

[vDHS15] Bart van Delft, Sebastian Hunt, and David Sands. Very static enforcement of dynamic policies. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of ETAPS 2015, Proceedings*, pages 32–52, 2015.

[vdM12] Ron van der Meyden. Architectural refinement and notions of intransitive noninterference. *Formal Asp. Comput.*, 24(4-6):769–792, 2012.

[VDMZ13] Ron Van Der Meyden and Chenyi Zhang. Information flow in systems with schedulers, part ii: Refinement. *Theoretical Computer Science*, 484:70–92, 2013.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[Web89] DG Weber. Formal specification of fault-tolerance and its relation to computer security. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 273–277. ACM, 1989.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 144–161, 1990.

[ZAM11] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*, pages 563–574, 2011.

[ZM01] Steve Zdancewic and Andrew C Myers. Robust declassification. In *Computer Security Foundations Workshop, IEEE*, pages 15–15. IEEE Computer Society, 2001.

[ZM05] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop*, pages 272–286, 2005.

Index of Subjects

Index of Symbols