Information Flow Security in Component-Based Models : From verification to Implementation

Najah Ben Said

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE la Communauté UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Najah Ben Said

Thèse dirigée par **Saddek Ben Salem**
et codirigée par **Marius Bozga**

préparée au sein **Verimag**
et de **École Doctorale Mathématique, Science et Technologie de l´Information, Informatique (MSTII)**

Information Flow Security in Component-Based Models : From Verification to Implementation

Thèse soutenue publiquement le **07 Novembre 2016**,
devant le jury composé de :

Mme. Marie Laure Potet
Professeur, Grenoble INP, Présidente
M. Gilles Barthe
Directeur de Recherche, IMDEA Software Institute , Rapporteur
M. Axel Legay
Chargé de Recherche , INRIA-Rennes , Rapporteur
M. Jean-Louis Lanet
Professeur, Université de Limoges, Examinateur
M. Catalin Dima
Professeur, Université Paris-Est Créteil , Examinateur
M. Saddek Bensalem
Professeur, Université Grenoble Alpes , Directeur de thèse
M. Marius Bozga
Ingénieur de Recherche-HDR, CNRS, Co-Directeur de thèse
Mme. Takoua Abdellatif
Dr. Ecole polytechnique Tunis, Co-Encadrante de thèse

Acknowledgements

Foremost, I would like to thank Saddek Bensalem for giving me the opportunity to join Verimag and to achieve my PhD work under his supervision. He offered me motivation, valuable research directions, and more importantly, a very nice working environment that helped me to accomplish this work.

I am deeply thankful to Marius Bozga, my co-advisor, for his availability, his valuable advice, and for the long and rich discussions, often beyond the work scope. Without his help, this work could never have been the same.

A great thank goes to Takoua Abdellatif for assisting and bringing help in several parts of the work. Especially, the support, guideness and advises she gave me. Her seriousness, competence and rigor have helped me throughout this thesis.

I am very grateful to all the jury members for accepting to review this work and for their valuable feedback on the manuscript and the presentation.

I want to thank all my colleges at Verimag for the nice, healthy, and rich working environment, all the administrative staff for their help and assistance.

A big thank to all my family and my friends for just being there for me.

Tribute to my parents ...

Abstract

The security of information systems is paramount in today's life, especially with the growth of complex and highly interconnected computer systems. For instance, bank systems have the obligation to guarantee the integrity and confidentiality of their costumer's accounts. The electronic voting, auctions and commerce also needs confidentiality and integrity preservation. However, security verification and its distributed implementation are heavy processes in general, advanced security skills are required since both security configuration and coding distributed systems are complex and error-prone. With the diverse security attacks leaded by the Internet advent, how can we be sure that computer systems that we are building do satisfy the intended security property ?

The security property that we investigate in this thesis is the non-interference, which is a global property that tracks sensitive information in the entire system and ensures confidentiality and integrity. Non-interference is expressed by the requirement that no information about secret data is leaked through the observation of public data variation. Such definition is more subtler than a basic specification of legitimate access for sensitive information, allowing to exploit and detect malfunctioning and malicious programs intrusions for sensitive data (e.g, Trojan horse that sends confidential data to untrusted users). However as a global property, the non-interference is hard to verify and implement.

To this end, we propose a model-based design flow that ensures the non-interference property in application software from its high-level model leading to decentralized secure implementation. We present the *secureBIP* framework [**?**] that is an extension for the component-based model with multy-party interactions for security. Non-interference is guaranteed using two practical manners : (1) we annotate the entire variables and ports of the model and then according to a defined set of sufficient syntactic constraints we check the satisfaction of the property, (2) we partially annotate the model way and then by extracting its compositional dependency graphs we apply a synthesis algorithm that computes the less restrictive secure configuration of the model if it exists [**?**].

Once the information flow security is established at a high-level model of the system, we follow a practical automated method to build a secure distributed implementation [**?**]. A set of transformations are applied on the

abstract model to progressively transform it into low-level distributed models and finally to distributed implementation, while preserving information flow security. Model transformations replace high-level coordination using multiparty interactions by protocols using asynchronous Send/Receive message-passing. The distributed implementation is therefore proven "secure-by-construction" that is, the final code conforms to the desired security policy. To show the usability of our method, we apply and experiment it on real case studies and examples from distinct application domains.

Résumé

Il est reconnu que garantir une sécurité de bout-en-bout pour les systèmes distribués est un problème très complexe. En effet, la communication bas niveau entre les différentes parties du programme demande l'implémentation d'un protocole de communication complexe pour garantir une sécurité globale à partir de la sécurité locale de chaque composant. Il est à noter que la décomposition du systme en sous-système est très souvent liée à un objectif fonctionnel et est indépendante de l'objectif de sécurité à atteindre. Donc, les données sont très souvent manipulées par des sous systèmes dont les niveaux de sécurité sont différents. Par ailleurs, il est souvent rare de trouver des programmeurs qui ont une expertise suffisant sur les mécanismes de sécurité à utiliser pour implémenter de tels protocoles de coordination de composants.

Dans le cadre de cette thèse, nous avons opté pour le développement de procédures automatiques pour garantir des propriétés de non-inférence dès la conception jusqu'à l'implémentation d'un système distribué fiable. Pour ce faire, nous avons étendue la plateforme BIP pour la création et vérification de la sécurité de flux d'information des systèmes à base de composants. Une combinaison de deux notions de non-interférence (d'événements et de données), garantissant l'objectif de sécurité, sont définies et formellement prouvés à travers des "unwinding" relations fortement utilisées pour vérifier la non-interférence. Une méthode automatisé basé sur l'utilisation de conditions syntactiques, inspiré danalyse statique, est utilisée pour valider quun modèle annoté par des labels de sécurité satisfait la propriété. Cette méthode est raffinée d'avantage avec l'utilisation d'un algorithme de synthèse (d'inférence) d'annotation de sécurité visant à simplifier et rendre plus pratique la procédure d'annotation du modèle.

La distribution du modèle peut introduire des difficultés de gestion de concurrence (conflits) entre composants et nécessite forcement une revérification des notions de sécurités, ce qui rend la têche fastidieuse. Dans cette thèse nous introduisons une méthode de distribution qui permet de préserver la satisfaction des conditions syntactiques de sécurités une fois vérifiées dans le modèle centralisé. Cette méthode est dite sécurisé-par-construction.

Contents

Chapter 1

Introduction

The amount and complexity of nowadays conceived systems and software knows a continuous increase. Especially, with today's highly connected word, where computers rarely work in isolation and instead, they are organized in heterogeneous distributed systems that collaborate with each others to create systems with improved processing and communication performance and higher storage capacities. However, the implementation of distributed systems may induce several issues related to both functional and security aspects. Indeed, the use of low level communications primitives require to decompose the application into independent parts and then to use or design a protocol that defines how the different parts communicate to ensure a correct and secure execution of the application. Finally, the designer has to implement this protocol by taking the security aspect into account. Each of these tasks is complex and tedious. Besides, secure code writing requires large knowledge and good expertise with security mechanisms, whereas, only a handful of programmers have the right mindset to do so, and few applications have the luxury of being written by such programmers. Given that, information in a distributed system can be manipulated, exchanged, duplicated and modified by entities operating in heterogeneous environments and sometimes unreliable. Hence, it is imperative to find a way to control and secure their use in a seamless manner.

However, except for the often-intuitive access control procedure with the use of cryptography primitives, it is rare to find a clear and well-established end-to-end security strategy for the development of distributed software. Unfortunately, even these mechanisms have been proven incomplete and limited since only by preventing the direct access to data, indirect (implicit) information flows are still possible given rise to the so called covert channels [**?**]. As an alternative, non-interference, one of the most promising

and also hard to establish end-to-end security property, has been studied as a global property to characterize and to develop techniques ensuring information flow security. Initially defined by Goguen and Meseguer [**?**], non-interference ensures that the system's secret information does not affect its public behavior.

Conceiving an approach to control information flow till implementation in an automated manner is the best way to tackle such security issues, where we basically rely on tool-sets that verify security properties at abstract level and then automatically generate codes. Here, we adapt a Model-driven Security (MDS) to build software applications. For avoiding failures in system's security operations that can be either attributed to conceptual or implementations errors, combining a model based approach with the use of formal method proofs appears to be most promising for either cases. Hence this combination provides assurance about system requirements validity and that are consequently well interpreted at implementation. Such approach reduces the reliance on developer's experiences which consequently reduces code errors.

This thesis aims at contributing to bridge the gap between the current theoretical and formal verification techniques for end-to-end information-flow security in distributed systems and implementation aspects on real platforms. The main focus here is to give a rigorous method that ensures the correct and secure building of systems at an abstract level from the early step of defining specifications until implementation and code generation. With focus on the non-interference property, we set in place a framework for system modeling and security verification defining solid guarantees of information confidentiality and integrity. Based on the analysis of security on the system model and by checking security consistency, we enable an automated process that automatically generates code that satisfies requirements (e.g, security) defined at higher level.

1.1 Model-Based Design and Security

Model-based design entails building software matching users-defined specifications and requirements. These specifications include functional aspects, related to functional tasks that a system should provide to the user, and non-function aspects, related to criteria that qualifies the system such us performance and security. The MDA (Model Driven Architecture) approach proposed by OMG (Object Management Group) is among the

main approaches for application design that provides a method to build applications by separating business logic from any technical implementations for platform. Indeed, unlike technical aspects, the business logic of the application is stable and undergoes little changes over time. It is evident then, to separate both to deal with the complexity of information systems and strong technology migration costs. Based on MDA approach, the MDS (Model Driven Security) have been conceived in the purpose to offer solutions to security constraints. Indeed, thanks to model-based design, MDS raises the level of abstraction in the design and development of secure applications. However and for the best of our knowledge, most of the works that adopted MDS for verifying information-flow security when creating the systems are not based on formal and rigorous studies of models and are just limited on applying access control techniques. Recently, [**?**] offers a solution that applies MDS approach to secure software. The authors use UML modeling to describe the functional specifications of the system and the safety specifications. Despite that this work provides a formal characterization of the syntax and semantics of the changes made in the shares, the designer should intervene to refine his model at each transformation which is error-prone task and not practical for large-scale systems.

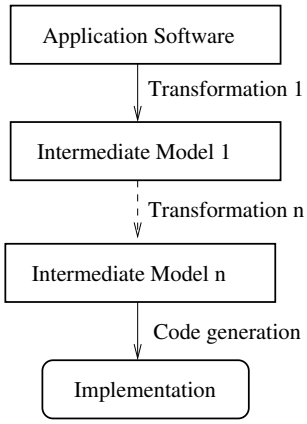

Figure 1.1 – Model-Based Design

Component-based models: The careful selection of a rigorous design model can greatly facilitate the application modeling and the analysis of security properties. By considering CBM (Component-Based Model), we represent systems using heterogeneous components with well-defined in-

terfaces ensuring dynamicity, scalability, transparency and interoperability while implementing distributed systems. Indeed, the decomposition of the systems used to study individual behaviour of each component and the study of inter-component interactions allows controlling the information-flow security. This has the advantage of being on the one hand, modular and secondly to be able to parallelize the verification of components. To do this, we use the BIP model (Behaviour, Interaction, Priorities), introduced in Chapter 4, for the description of the system and try to extend it to specify the security properties. *BIP* model allows describing the behaviour of components, their interactions and priorities to filter those interactions as desired properties, which seems to us very appropriate for the security of the information flow. A component-based model has been proposed in [**?**] and used to study implementation issues of secure information flows. The work we present in this thesis treats the information flow security in a different aspect with formal proofs and finer granularity.

1.2 Information-Flow Security

Specifying the security requirement correctly of a given system is not a trivial task. For instance, keeping an amount of money withdrew from an ATM system confidential would be naively interpreted, as this information must never leave the system. However such approach has serious deficiencies in a way that if you choose to make a tractability of your account money by receiving emails containing such transactions history would not be possible. Otherwise, by changing a bit in the confidential information value (e.g, increment the amount value with 1$) and then send it and decrements the received value to obtain the money amount would be accepted. Given an other classical example, considering a travel service orchestrator, where a businessman is planning a trip to a private meeting in a private destination. The observation of the price information of the trip, an intruder can deduce the destination. Ensuring confidentiality of the system is quietly related to verifying that its outputs does not *depend* on confidential information. Hence, confidentiality should be interpreted as the lack of *dependencies on confidential information*. This approach is interpreted as the *non-interference* property that we consider and treat all over this thesis.

The non-interference property, initially set by Goguen and Meseguer[**?**], ensures that sensitive data does not affect the publicly visible system behavior. This property allows the tracking of information in the system, and the application of an end-to-end confidentiality and integrity. For example,

in contrast with access control policies, which require only users with the appropriate read rights to read a shared file , the non-interference property requires that none of the users having not the appropriate level of security can not have any information about the content f file, even indirectly.

Non-interference verification techniques are based on the use of information flow control mechanisms [**?**, **?**, **?**, **?**]. Mainly, these techniques are based on tracking the spread of information in the system, assigning security labels to the system's various communication channels. An annotation represents a certain level of security and classifies various communication channels according to their level of restriction. Consider a system that processes data from an input channel and broadcasts them using output channels. Labels are assigned to these channels, e.g *High* label for channels handling sensitive or secret information, and *Low* for channels manipulating public information. Non-interference property implies that, during execution, data from High level input channels must not flow to the level of output channels Low, because by observing the behavior of public outputs, an intruder can deduce the value of sensitive inputs.

Despite the long history in literature for non-interference verification that started in the early 80s, the application of the theoretical results in real systems where limited since the non-interference property is hard to ensure. Some operating systems such as Flume [**?**], HISTAR [**?**] and Asbestos [**?**] and programming language like JIF [**?**] have attempt to ensure information flow security in distributed systems but they fall short with effectiveness. Since they consider the information flows at a coarse granularity level, they can cause over-approximations causing false positive: the system can detect interference while they do not exist. In the work we propose in this thesis, we give a practice method to verify the non-interference property using component-based model allowing a fine-grained analysis of security. Then, we follow a set of transformation steps enforcing the security established at a centralized high-level model until the generation of the distributed code.

1.3 Contributions

Based on the previously identified challenges in the state-of-the-art, we provide hereafter an overview of the contributions of this thesis. The presentation below is orthogonal to the global organization of the manuscript where we start by presenting generic theoretical contributions and then their applications on different specific platforms, as presented in Figure 1.2. In the first part, we present a rigorous component-based framework that we extend to analyze information-flow security in the system. Then

we develop a method to generate by-construction secure distributed code. In the second part, we provide tool-supported flow to automate code and configuration generation for different platforms with different contexts.

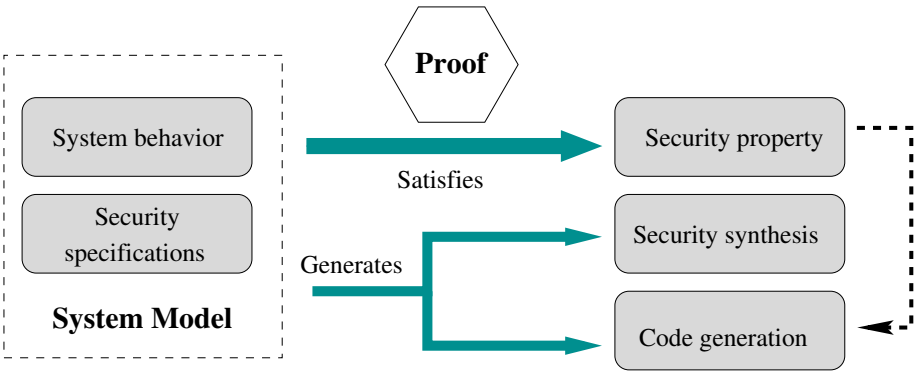

Figure 1.2 – Contributions overview

Framework for Security Verification The first contribution of this thesis is the *secureBIP*, a formal framework that is an extension of the *BIP* framework with security. *SecureBIP* provides a system construction methodology for complex systems where big systems are functionally decomposed into multiple sub-components communicating through well-defined interactions. Sensitive information are tracked in the entire system by using annotation model that defines different security levels tagging ports and data of the *secureBIP* model. Following the annotation model, we privilege a very pragmatic approach and provide simple syntactic conditions to automate the verification of non-interference. These conditions eliminate a significant amount of security leakages, especially covert channels, independently from implementation language or the execution platform.

Moreover, with *secureBIP* we handle two types of non-interference: event and data-flow non-interference, in a single semantic model. The need to consider together event and data flow non-interference has been recently identified in the existing literature. The bottom line is that preserving the safety of data flow in a system does not necessarily preserve safe observability equivalence on system's public behavior (i.e., secret/private executions may have an observable impact on system public events).

Information-Flow synthesis Information flow security verification is mainly based on annotations either at an abstract level (models) or in the

code itself. However, security designers do have intuitive idea about some confidential information, but not all data that a complex system may include. Hence finding a correct configuration to given complex system is a hard and complicated task. In this work, we propose a formal model that enables automated synthesis of security configurations, under the guidance of the system designer. The overall approach is based on propagating annotations on the model following data dependency-graphs and respecting the local security constraints defined with for *secureBIP* model. This approach is practical since the designer does not need to be expert in security and concentrates only on the functional specifications.

Security-by-Construction The distributed computing systems are ubiquitous and do increase with complexity which make it difficult to make strong statement about the security provided by the hole system, especially when dealing with distrusted parties and participants. In this thesis, we propose a unified approach to build and implement distributed systems that enforce end-to-end security (i.e, confidentiality and integrity). Assuming that we use trusted platform and based on the idea of static information flow control, we provide a method, that starting from an initially centralized system satisfying the set of security constraints and produces a functionally equivalent distributed model where these security constraints are preserved, we make an automated transformation that splits multiparty interactions and atomic executions into implementable send/receive messages. The decentralized system is a component-based model where communications between distinct components are managed using interaction protocol components that each can handle data computations to a certain security domain at most (e.g, an upper-bound security level). Here we pursue a constructive approach where, the system designer has only to focus on specifying a secure centralized multiparty model and then a compiler automatically generates a secure send/receive model.

Code Generation To enable rising system performance at execution we rely on automatic code generation. Starting from optimized security annotations on the *secureBIP* model, we introduce cryptography components that ensures the security of communications inter-components and encapsulating the sensitive data exchanged between them. Then we generate from this intermediate model a concrete implementation targeting distributed platform architecture. For each atomic component, we generate a standalone program that implements the Petri net for automaton

representing the behavior of the component. Send/Receive communications are encrypted following a platform dependent configuration file that specifies for each security level the encryption algorithm and the assignment mechanisms to use to secure these channels.

1.4 Organization

The remainder of this thesis is organized as follows:

— The two Chapters 2 and 3 review the state-of-the-art on which the work is based. The Chapter 2, presents the centralized and decentralization of the component-based model, *BIP* , that we use consistently in this thesis. The Chapter 3, presents the non-interference property and the main verification techniques previously given in literature to handle it.

— Chapter 4 formally presents the *secureBIP* framework. We formally define non-interference property and we give two methods to verify it. First method is based on verifying the satisfaction of sufficient conditions and the second method is based on the synthesis of security annotations.

— Chapter 5 describes the secure-by-construction approach to build non-interferent distributed systems. The proposed solution consists on splitting atomic executions on the centralized secure model into Send/Receive messages handled between all atomic components through distributed schedulers. These schedulers are constructed to be security aware components that handle interaction and data exchanged between atomic components while preserving the security constraints of the centralized model.

— The two Chapters 6 and 7 introduce our implementation and experimentation works. Chapter 6 presents the tool-set that we implemented based on the *BIP* framework. The tools are related to non-interference verification, code and configuration generations and some transformation tools that allow to apply verification techniques on real-applications for evaluation. Chapter 7 illustrates this evaluation on two systems: non-interference verification on a Web Service composition and on an AADL-MILS application.

— Finally, we conclude and outline some future work in Chapter 8.

Chapter 2

Component-Based Model

Contents

Component-based design is founded on a paradigm that allows the construction of complex systems by assembling components. The use of component models allows describing systems in high-level of abstraction independently of the implementation details and provides formal semantics to reason about their properties and automate their analysis and code generation. Hence, we consider that such models are very adequate to reason about security and track data and event dependencies of system executions at an abstract level. Our approach to describing system model is based on using BIP [?, ?] framework for the construction of complex, heterogeneous embedded applications. BIP is a highly expressive [?], component-based framework with formal semantics. A BIP model is the superposition of three layers: a behavioral layer, an interaction layer and a priority layer, as presented in Figure 2.1.

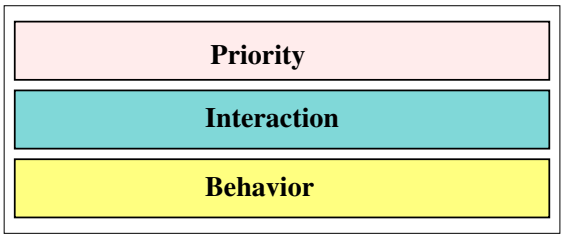

Figure 2.1 – BIP model architecture.

The BIP models describe systems where communications between atomic component are assured using strong synchronized multi-party interactions. Atomic components behavior is represented using state machines, communicate between each other through a set of interface(port). Priorities are used to filter amongst possible interactions and to steer system evolution so as to meet performance requirements, e.g. to express scheduling policies. Priorities define a partial orders between interactions and can change dynamically.

The Decentralized implementation of the BIP models and allowing parallel execution between components and interactions simultaneously is a non-trivial task. Indeed, adding implementation details involves many subtleties(e.g., inherently concurrent, non-deterministic, and non-atomic structure of distributed systems) that can potentially introduce errors to the resulting system. To resolve this problem, we review a source-to-source transformation [?, ?] of the centralized *BIP* model allowing to introduce message passing communication primitives and managing interactions in a distributed way using dedicated scheduler components. A hierarchical

3-layered model is introduced to automatically decentralized *BIP* .

This chapter is structured as follows. The BIP model is described in Section 2.1 . It gives an abstract formalization of the layers Behavior, Interaction, Priority. Section 2.2 describes the automated transformation of BIP that aim to decentralize the model. In this model, we present the challenges introducing concurrency in the decentralized model and we introduce the three layer architecture adapted in distributing this model.

2.1 The BIP Component-based Framework

BIP stands for *behavior*, *interaction* and *priority*, that is, the three layers used for the definition of components and their composition in this framework. This framework allows the construction of complex, hierarchically structured models from atomic components characterized by their behavior and their interfaces. The behavior of atomic components is a transition systems enriched with data. Atomic components are composed by layered application of interactions and priorities. Interaction layer, also referred to as *glue*, express synchronization constraints and do the transfer of data between the interacting components. Priority layer is used to filter interactions and to steer system evolution so as to meet performance requirements (e.g., to express scheduling policies). In this section, we will formally introduce and give the semantics of the three layers of the model.

2.1.1 Atomic Components

In BIP, atomic components are transition systems labelled with ports and extended with variables used to store local data. Transitions are used to move from a source to a destination state. Each transition is associated to a guard and an update function, that are respectively a Boolean condition and a computation defined over local variables. Ports are action names generally used for synchronization with other components. States denote control locations at which the components wait for synchronization. In transitions, data and their related computation are written in C/C++ language. The syntax of an atomic component in BIP is formally defined as follow.

Let $\mathbb{D} = \{D_j\}_{j\in\mathcal{J}}$ be a universal set of data domains (or data types) including the Boolean domain $\mathbb{D}_{Bool} = \{true, false\}$. Let $\mathbb{E}xpr$ be an universal set of operators, that is, functions of the form $op : \times_{i=1}^{m} D_{j_i} \rightarrow D_{j_0}$, where $m \geq 0$, $D_{j_i} \in \mathbb{D}$ for all $i = 0, m$. We consider typed variables $x : D$ where x denotes the name of the variable and $D \triangleq dom(x) \in \mathbb{D}$ its

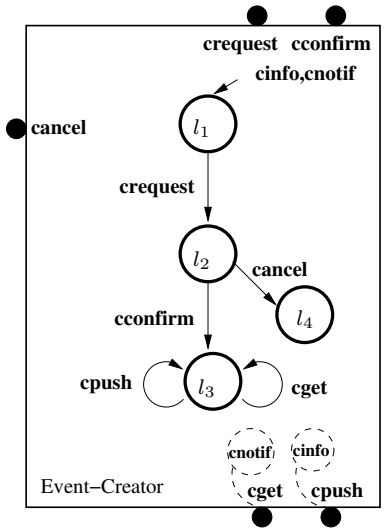

Figure 2.2 – An example of a BIP atomic component.

domain of values. We define expressions $op(x_1, ..., x_m)$ constructed by applying operators op on variables $x_1, ..., x_m$ such that moreover, the number of variables and their domains match exactly the domain of op. We denote by $use(e)$ the set of variables occurring in e and by $\mathbb{E}xpr[X]$ the set of expressions constructed from a set of variables X and operators in $\mathbb{E}xpr$. We denote by $\mathbb{A}sgn[X]$ the set of assignments to variables in X, that is, any subset $\{(x_i, e_i)\}_{i\in I} \subseteq X \times \mathbb{E}xpr[X]$ where $(x_i)_{i\in I}$ are all distinct. An assignment (x, e) is denoted by $x := e$.

Definition 1 (atomic component) *An atomic component B is defined by $B = (L, X, P, T)$ where:*

- *L is a set of locations.*
- *P is a set of ports.*
- *X is a set of variables and we denote by X_p the subset of exported variable through the port p.*
- *$T \subset L \times ¶ \times Exp[X] \times Asgn[X] \times L$ is a set of port labelled transitions. For every transition $\tau \in T$, we denote by $g_\tau \in Exp[X]$ its guard, that is, a Boolean expression defined on X and by $f_\tau \in Asgn[X]$ its update function, that is, a parallel assignment $\{x := e^x_\tau\}_{x\in X}$ to variables of X.*

Example 1 *Figure 2.2 shows an example of an atomic, Event-Creator component that communicates with other atomic components to create an*

Event between them. Here, it contains three control locations, l_1 the initial location, l_2 and l_3 related through transition labeled with ports crequest, $cconfirm$, cget, cpush, and cancel. From l_1, the crequest transition is always enabled and leads to location l_2. From l_2, the component can either execute the transition labelled with cancel or executes the transition labelled with port $cconfirm$ which synchronises with other atomic components to create an event. In case, the transition labelled with $cconfirm$ is executed, the atomic component can execute the transitions labelled with ports cget and cpush and update the exported variables $cnotif$ and $cinfo$ respectively exported by them.

Given a set of variables X, we define valuations V of X as functions $V : X \to \cup_{j\in J} D_j$ which assign values to variables, such that moreover, $V(x) \in dom(x)$, for all $x \in X$. We denote by $V[u/x]$ the valuation where variable x has assigned value u and any other variable has assigned the same values as in V. For a subset $Y \subseteq X$, we denote by $V_{|Y}$ the restriction of V to variables in Y.

Given an expression $e = op(x_1, ..., x_m) \in \mathbb{E}xpr[X]$ and a valuation V on X we denote by $e(V)$ the value $op(V(x_1), ..., V(x_m))$ obtained by evaluating the expression operator op according to values of $x_1, ..., x_m$ on the valuation V. Moreover, given an assignment $a \in \mathbb{A}sgn[X]$ and a valuation V of X we denote by $a(V)$ the new valuation V' obtained by executing a on V, formally $V'(x) = e(V)$ iff $x := e \in a$ and $V'(x) = V(x)$ otherwise.

A *BIP* atomic component, for a given valuation of variables, can execute a transition if and only if its associated guard evaluates to true. When several transitions are simultaneously enabled, a non-deterministic choice is performed to select one of them. Firing a transition implies an exchange of data through the port followed by the atomic execution of its internal computation. Formally:

Definition 2 (atomic component semantics) *The semantics of an atomic component $B = (L, X, P, T)$ is defined as the labelled transition system* $\text{LTS}(B) = (Q_B, \Sigma_B, \underset{B}{\rightarrow})$ *where the set of states $Q_B = L \times \mathbf{X}$, the set of labels is $\Sigma_B = P \times \mathbf{X}$ and the set of labelled transitions $\underset{B}{\rightarrow}$ is defined by the rule:*

$$\text{ATOM}\ \frac{\tau = \ell \xrightarrow{p} \ell' \in T \qquad \mathbf{x}''_p \in \mathbf{X}_p \qquad g_\tau(\mathbf{x}) \qquad \mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])}{(\ell, \mathbf{x}) \xrightarrow[B]{p(\mathbf{x}''_p)} (\ell', \mathbf{x}')}$$

That is, $(\ell', \mathbf{x}')$ is a successor of $(\ell, \mathbf{x})$ labelled by $p(\mathbf{x}''_p)$ iff (1) $\tau = \ell \xrightarrow{p} \ell'$ is a transition of T, (2) the guard g_τ holds on the current valuation $\mathbf{x}$,

(3) $\mathbf{x}''_p$ is a valuation of exported variables X_p and (4) $\mathbf{x}' = f_\tau(\mathbf{x}[X_p \leftarrow \mathbf{x}''_p])$ meaning that, the new valuation $\mathbf{x}'$ is obtained by applying f_τ on $\mathbf{x}$ previously modified according to $\mathbf{x}''_p$. Whenever a p-labelled successor exist in a state, we say that p is *enabled* in that state.

2.1.2 Interactions and Priorities

The interaction layer consist of a set of multiparty interactions relating ports from different sub-components. Each interaction represent a nonempty set of ports that have to be jointly executed at interaction execution. For every interaction, we define a guard and a data transfer function, that are, respectively, an enabling condition and an assignment of data exported across the involved ports.

Consider a composition of an existing set of atomic components $\{B_i = (L_i, X_i, P_i, T_i)\}_{i=1,n}$ such that they have pairwise disjoint sets of states, ports, and variables i.e., for any two $i \neq j$ from $\{1..n\}$, we have $L_i \cap L_j = \emptyset$, $P_i \cap P_j = \emptyset$, and $X_i \cap X_j = \emptyset$. We denote $P = \bigcup_{i=1}^{n} P_i$ the set of all the ports, $L = \bigcup_{i=1}^{n} L_i$ the set of all locations, and $X = \bigcup_{i=1}^{n} X_i$ the set of all variables.

Definition 3 (interaction) *An* interaction *a between atomic components is a triple (P_a, G_a, F_a), where $P_a \subseteq P$ is a set of ports, G_a is a guard, and F_a is an update function. By definition, P_a uses at most one port of every component, that is, $|P_i \cap P_a| \leq 1$ for all $i \in \{1..n\}$. Therefore, we simply denote $P_a = \{p_i\}_{i \in I}$, where $I \subseteq \{1..n\}$ contains the indices of the components involved in a and for all $i \in I, p_i \in P_i$. G_a and F_a are both defined on the variables exported by the ports in P_a (i.e., $\bigcup_{p \in P_a} X_p$).*

Given a set of interactions γ and components $C = (B_1, \ldots, B_n)$, the execution of an interaction $a \in \gamma$ can only take place if its associated guard G_a is evaluated to true and all its involved ports P_a are enabled. The execution of an interaction a is an atomic sequence of two micro-steps:

1. synchronization between participant components of the interaction through the involved ports $\{p_i\}_{i \in I} \in P_a$, with data assignment (exchange) by executing the transfer function at connector level F_a
2. Execution of internal transitions, of distinct participant components in interaction a, labelled with ports $\{p_i\}_{i \in I} \in P_a$

Given a system of interacting components, priorities are used to filter the enabled interactions. They are given by a set of rules, each consisting of an ordered pair of interactions or connectors. When connectors are

specified in a priority, the rules apply between all the respective interactions of the connectors. Dynamic priorities can be specified by providing guard condition, which are boolean expression in C on the variables of the components involved in the interactions. The maximal progress priority is enforced implicitly by the BIP Engine: if one interaction is contained in another one, the latter has higher priority. Below is an example of priority expressed in the BIP language.

$$\boxed{\text{priority } p_1 \text{ iff(G) } a_1 < a_2}$$

This specifies the priority p1 that, when the boolean condition G is true, interactions of connector a2 would be preferred to those of a1.

2.1.3 Composite Components

In component-based modeling, systems are constructed from assembling a set of unitary components using composition operators. BIP offers a layered *glue* which provides mechanisms for coordinating components behaviors, namely interactions.

A composite component $C = \gamma(B_1, \ldots, B_n)$ is obtained by applying a set of interactions γ to a set of atomic components $B_1, \ldots B_n$. Hereafter, we present the semantics of a composite component.

Definition 4 (composite component semantics) *Let* $C = \gamma(B_1, \ldots, B_n)$ *be a composite component. Let* $B_i = (L_i, X_i, P_i, T_i)$ *and* $\text{LTS}(B_i) = (Q_i, \Sigma_i, \underset{B_i}{\longrightarrow})$ *their semantics, for all* $i = 1, n$. *The semantics of* C *is the labelled transition system* $\text{LTS}(C) = (Q_C, \Sigma_C, \underset{C}{\longrightarrow})$ *where the set of states* $Q_C = \otimes_{i=1}^{n} Q_i$, *the set of labels* $\Sigma_C = \gamma$ *and the set of labelled transitions* $\underset{C}{\longrightarrow}$ *is defined by the rule:*

$$\textsc{Comp}\ \frac{\begin{array}{c} a = (\{p_i\}_{i\in I}, G_a, F_a) \in \gamma \qquad G_a(\{\mathbf{x}_{p_i}\}_{i\in I}) \qquad \{\mathbf{x}''_{p_i}\}_{i\in I} = F_a(\{\mathbf{x}_{p_i}\}_{i\in I}) \\ \forall i \in I.\ (\ell_i, \mathbf{x}_i) \xrightarrow[B_i]{p_i(\mathbf{x}''_{p_i})} (\ell'_i, \mathbf{x}'_i) \qquad \forall i \notin I.\ (\ell_i, \mathbf{x}_i) = (\ell'_i, \mathbf{x}'_i) \end{array}}{((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n)) \underset{C}{\xrightarrow{a}} ((\ell'_1, \mathbf{x}'_1), \ldots, (\ell'_n, \mathbf{x}'_n))}$$

For each $i \in I$, $\mathbf{x}_{p_i}$ *above denotes the valuation* $\mathbf{x}_i$ *restricted to variables of* X_{p_i}.

As previously explained, the rule expresses that a composite component $C = \gamma(B_1, \ldots, B_n)$ can execute an interaction $a \in \gamma$ *enabled* in state $((\ell_1, \mathbf{x}_1), \ldots, (\ell_n, \mathbf{x}_n))$, iff (1) for each $p_i \in P_a$, the corresponding atomic

component B_i can execute a transition labelled by p_i, and (2) the guard G_a of the interaction holds on the current valuation of variables exported on ports participating in a. Execution of interaction a triggers first the update function F_a which modifies variables exported by ports $p_i \in P_a$. The new values obtained, encoded in the valuation $\mathbf{x}''_{p_i}$, are then used by the components' transitions. The states of components that do not participate in the interaction remain unchanged.

Any finite sequences of interactions $w = a_1...a_k \in \gamma^*$ executable by the composite component starting at some given initial state q_0 is named a trace. The set of all traces w from state q_0 is denoted by TRACES(C, q_0).

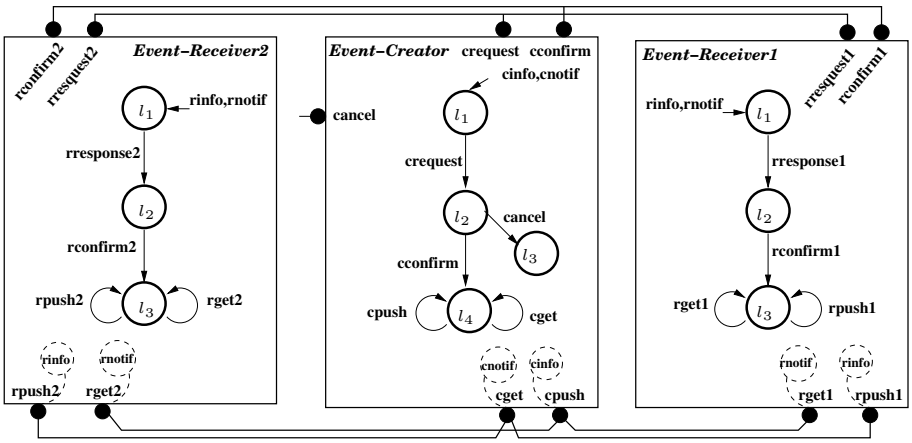

Figure 2.3 – Event-Creation in the Whens-App application

Example 2 *Figure 2.3 presents a composite component that contains three atomic components, Event-Creator and two Event-Receiver in interaction. The composition represents an event creation between the three atomic components. Here interactions are represented using connectors (lines) between the interacting ports. All interactions between components Event-Creator and Event-Receivers are strong synchronized binary interactions. The interactions {get,push} implements a data transfer between the two Event-Receivers , that is, an assignments at exportation between variables $info$ and $notif$.*

2.2 Distributed BIP Framework

In this section, we recall from [**?**], the key steps in decentralizing the functional BIP model. The target model aims to be implementable using

basic message-passing primitives and orchestrator engines to manage multiparty interactions in a distributed way.

2.2.1 Challenges

Deriving from a high-level model a correct and efficient distributed implementation, that allows parallelism between components as well as parallel execution between interactions, is a challenging problem. As adding implementation details involves many subtleties (e.g., inherently concurrent, non-deterministic, and non-atomic structure of distributed systems) that can potentially introduce errors to the resulting system.

The decentralized model introduced in [**?**] is based mainly in introducing engine components that manage interactions execution between distributed atomic components. Engine implementation can be either centralized, a single engine handling all the interactions allowing only concurrency at interaction execution, or decentralized using multiple engines handling interaction partitions and allowing concurrency between interactions and components. On the contrary, introducing concurrency and distribution (and possibly multiple Engines) to the model requires dealing with complex issues:

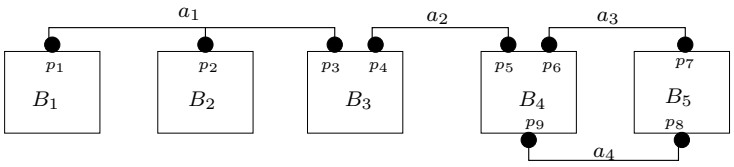

Figure 2.4 – Simple high-level model.

— (Partial observability) Suppose interaction a_1 (and, hence, components $B1, \ldots, B3$) is being executed. If component B_3 completes its computation before B_1 and B_2, and, ports p_4, p_5 are enabled, then interaction a_2 is enabled. In such a case, distributed Engines must be designed so that concurrent execution of interactions does not introduce behaviors that were not allowed by the high-level model. We address the issue of partial observability by breaking the atomicity of execution of interactions, so that a component can execute unobservable actions once a corresponding interaction is being executed [BBBS08].

— (Resolving conflicts) Suppose interactions a_1 and a_2 are enabled simultaneously. Since these interactions share component B_3, they

cannot be executed concurrently. We call such interactions conflicting. Obviously, distributed Engines must ensure that conflicting interactions are mutually exclusive.

— (Performance) On top of correctness issues, a real challenge is to ensure that a transformation does not add considerable overhead to the implementation. After all, one crucial goal of developing distributed and parallel systems is to exploit their computing power.

Conflicts

Generally, there is a conflict between two entities whenever they are competing to use a given resource. When using decentralized orchestrator engines to handle interactions, the resources being used are the offers (requests) sent by the interacting components to execute some interaction. We can distinguish two types of conflicting interactions: *external* or *internal* conflict. Figure 2.5(i) shows an external conflict, where, assuming that interactions a and b are handled by separate orchestrators, the component first sends an offer to both orchestrators to execute either interactions, which here cause a conflict. Figure 2.5 (ii), shows an interaction execution at a non-deterministic state, where a can be executed through port p_1 or b through port p_2, the offer is sent to both orchestrators handling a and b. At this point, a conflict arises as only one of the engines should respond to the offer, since the component can execute only one of the transitions.

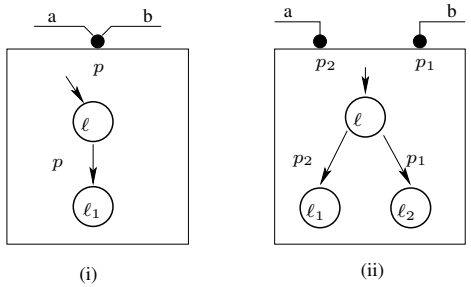

Figure 2.5 – Conflicting Interactions

More formally we define the set of conflicting interactions in the *BIP* model as follows:

Definition 5 *Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component and a, b $\in \gamma$ be two interactions. The interactions a and b are conflicting if*

— *either, they share a common port p; $ports(a) \cap ports(b) \neq \emptyset$*

— *there is at least a component B_i in the intersection containing a control location ℓ with an outgoing transition labeled by a port of an outgoing transition labeled by a port of b. formally, $p_1 \in a$, $p_2 \in b$ and $\ell \xrightarrow{p_1} \wedge \ell \xrightarrow{p_2}$*

We denote this situation by a # b or equivalently b # a.

Considering a partitioning of the set of interactions γ where $\gamma = \gamma_1 \cup \cdots \cup \gamma_k$ and $\forall i, j \in \{1, \ldots, k\},\ i \neq j \Longrightarrow \gamma_i \cap \gamma_j = \emptyset$. Here, $\gamma_1, \ldots, \gamma_j$ is conflict-free partition if the γ_i are pairwise conflict-free, that is if for any two distinct integers i, j in $\{1, \ldots, k\}$, γ_i and γ_j are conflict-free.

In this section we give the 3-layer model as a solution for decentralizing *BIP* model and that tackles the different challenges encountered at enforces concurrency in distributed implementation.

2.2.2 The 3-Layer Model

This decentralization method relies on a systematic transformation of *BIP* components and replacement of multiparty interaction by protocols expressed using send/receive (S/R) interactions. The execution of a distributed components is a sequence of actions that are either message emission, message reception or internal computation. Consequently transformed atomic components in our target model include three types of ports: send ports, receive ports and unary ports. Unary ports correspond to internal computation that is a unary interaction involving only one component. Whereas, the S/R interactions are binary point-to-point and directed interactions from one sender component (send_port), to one receiver component (receive_port). All interactions in the target model are managed via distributed *Interaction Protocol* (orchestrator engines), and *Conflict-resolution Protocol* components. Following these notions, we define a 3-layer decentralized model containing three kinds of distributed components which correspond to the three layers of the distributed model as presented in Figure 2.6. To ensure simple implementation on a distributed platform using basic (i.e. Send/Receive) message-passing primitives, we require that the automaton of atomic components used in centralized model is deterministic. For each port, there is at most one enabled transition labeled by this port at each state. In particular, two transitions outgoing from the same control state and labeled with the same port must have mutually exclusive guards.

Definition 6 (composite S/R component) *$C^{SR} = \gamma^{SR}(B_1^{SR}, \ldots, B_n^{SR})$ is a S/R BIP composite component if we can partition the set of ports of*

B^{SR} into three sets P_s, P_r, P_u that are respectively the set of send-ports, receive-ports and unary interaction ports.

- Each interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ is either (1) a S/R interaction with $P_a = (s, r_1, r_2, ..., r_k)$, $s \in P_s$, $r_1, ..., r_k \in P_r$ and $G_a = true$ and F_a copies the variables exported by the port s to the variables exported by the port $r_1, r_2..., r_k$ or (2) a unary interaction $P_a = \{p\}$ with $p \in P_u$, $G_a = true$, F_a is the identity function.
- If s is a port in P_s, then there exists one and only one S/R interaction $a = (P_a, G_a, F_a) \in \gamma^{SR}$ with $P_a = (s, r_1, r_2, ..., r_k)$ and all ports $r_1, r_2, ..., r_k$ are receive ports. We say that $r_1, r_2, ..., r_k$ are the receive ports associated to F.
- If $a = (P_a, G_a, F_a)$ with $P_a = (s, r_1, r_2, ..., r_k)$ is a S/R interaction in γ^{SR} and s is enabled in some global state of C^{SR} then all its associated receive-ports $r_1, r_2, ..., r_k$ are also enabled at that state.

From a functional point of view, the main challenge when transforming *BIP* models with multiparty interactions towards distributed models with send/receive interactions is to enhance parallelism for execution of concurrently enabled interactions and computations within components. That is, in a distributed setting, each atomic component executes independently and thus has to communicate with other components in order to ensure correct execution with respect to the original semantics. The existing method for distributed implementation of *BIP* relies on structuring the distributed components according to a hierarchical architecture with three layers, as depicted in Figure 2.6:

- The first layer (S/R atomic component) includes transformed atomic components. Each atomic component will publish its offer, that is the list of its enabled ports, and then wait for a notification indicating which interaction has been chosen for execution.
- The second layer (*IP*) deals with distributed execution of interactions by implementing specific interaction protocols. The interaction protocol evaluates the guard of each interaction and executes the associated update function The interface between this layer and the component layer provides ports for receiving offers and notifying the ports selected for execution.
- The third layer (*CRP*) deals with conflict resolution between *IPs*. A conflict occurs if two different *IP* components try to execute two interactions involving a common atomic component in the lower layer. Several distributed algorithms exist for conflict resolution, this layer is generic with appropriate interfaces.

We graphically denote a send port by a trigger (▲) and a receive or a unary port by a circle (●). In the following sections we describe the different components in the three layer model and we give there semantics.

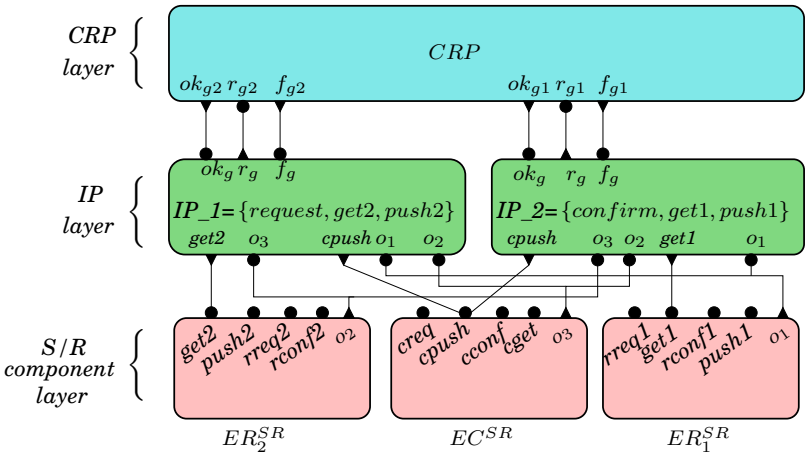

Figure 2.6 – Decentralized 3-Layer Model Architecture.

In Figure 2.6, we give an example of decentralizing the *Whens-up* system model. Here, the interactions are divided in two subsets: first $\{request, get2, push2\}$ and second $\{confirm, get1, push1\}$, that are handled by two IP components. The atomic component are transformed to S/R components that send offers through o ports, and receive notifications on ports p. For the sake of clarity, in Figure 2.6 we only represent interactions $get1$ and $get2$. Following this partitionning the get_i and $push_i$ interactions are in external conflict. Hence, a CRP component is introduced to resolve it.

Distributed Atomic Layer

An atomic component B is transformed to a S/R component B^{SR} by breaking the atrocity of each "atomic" synchronized transition, where the transition is split into a send and receive action. Hence, the execution of the transition is executed in two steps. The synchronization between participants is implemented as a two-phase protocol between the atomic components and orchestrator engines.

First B^{SR} sends an offer through a dedicated send port, then it waits for a notification arriving on a receive port. The offer contains the information to determine whether an interaction is enabled. An offer includes the set of enabled ports of B^{SR} through which the component is currently ready to interact and the values of a set of exported variables. The exported

variables are variables that may be read and written during the interaction from different participant components. These variables can be used in guards or in the update functions.

We encode enabled ports and current control state as follows. For each port p of the transformed component B^{SR}, we introduce a Boolean variable x_p. We add a participation number (state variable) n that contains the participation number of atomic component. The newly added variables are modified by an update function when reaching a new state. The variable x_p is then set to true if the corresponding port p becomes enabled, and to false otherwise. Similarly, before reaching the control state ℓ, the variable n is incremented, indicating that the transition have been executed.

Since each notification from the engine triggers an internal computation in a component, following [**?**], we split each control location ℓ into two control locations, namely, ℓ itself and a busy control location $\perp_\ell$. Intuitively, reaching $\perp_\ell$ marks the beginning of an unobservable internal computation. We are now ready to define the transformation from B into B^{SR}.

Definition 7 (Transformed atomic component) *Let $B = (L, X, P, T)$ be an atomic component within C. The corresponding transformed S/R component is $B^{SR} = (L^{SR}, X^{SR}, P^{SR}, T^{SR})$:*

- *$L^{SR} = L \cup L^{\perp}$, where $L^{\perp} = \{\perp_\ell \mid \ell \in L\}$*
- *$X^{SR} = X \cup \{x_p\}_{p \in P} \cup \{n\}$ where each x_p is a Boolean variable indicating whether port p is enabled, and n is an integer called a participation number.*
- *$P^{SR} = P \cup \{o\}$. The offer ports o export the variables $X_o^{SR} = \{n\} \bigcup \{\{x_p\} \cup X_p\}$ that is the participation number n, the new Boolean variables x_p and the variables X_p associated to ports p. For all other ports $p \in P$, we keep $X_p^{SR} = X_p$.*
- *For each state $\ell \in L^{SR}$, we introduce an offer transition $\tau_o = (\perp_\ell \xrightarrow{o} \ell) \in T^{SR}$ where the guard g_o is true and f_o is the identity function.*
- *For each transition $\tau = (\ell \xrightarrow{p} \ell') \in T$ we include a notification transition $\tau_p = (\ell \xrightarrow{p} \perp_{\ell'})$ where the guard g_p is true. The function f_p applies the original update function f_τ on X, increments n and updates the Boolean variables x_r, for all $r \in P$. That is, $x_r := g_\tau$ if $\exists \tau = \ell' \xrightarrow{r} \ell'' \in T$, and $false$ otherwise.*

Example 3 *Figure 2.7 shows the distributed S/R version of the atomic component from 2.2. For each original control location ℓ_i, $i \in \{1, 2, 4\}$ with an outgoing synchronized transition, a busy control location $\perp_\ell$ has been added with a transition labelled with the offer port o from $\perp_{\ell_i}$ to ℓ_i, $i \in$*

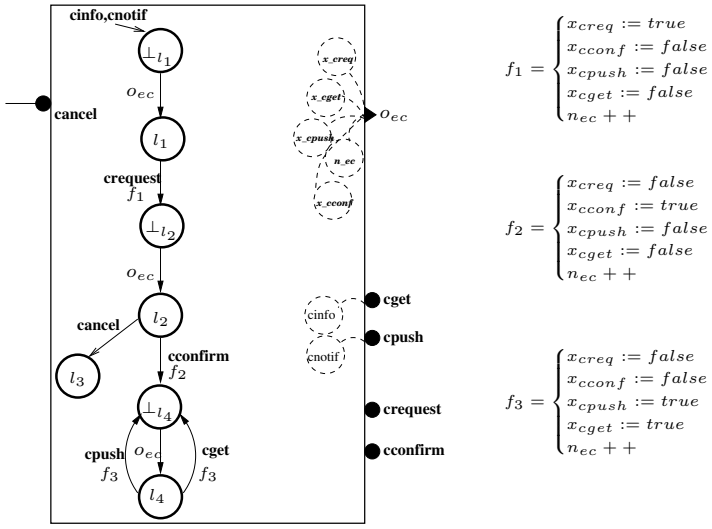

Figure 2.7 – Distributed version of the Event-Creator component from Figure 2.2

$\{1, 2, 4\}$. *For instance, before executing the transition* $\perp_{\ell_4}$ *to* ℓ_4*, the update function* f_3 *is called. If the transition labelled with port cpush is selected to be executed, when reaching the* $\perp_{\ell_4}$ *the variable* n_{ec} *is incremented and the variable* x_{cpush} *is reset to true.*

Interaction Protocol Layer

The *Interacting Protocol* layer contains a non-empty set of engine components. The implementation of the IP can be centralized, where all interactions of the initial *BIP* model are managed by a single engine component, or it can be decentralized, where a set of engine components are used and each of them is in charge of executing a subset of interactions. Each IP component represents a scheduler that receives messages from S/R components then calculates the enabled interaction and selects them for execution. Hereafter, we introduce the functional behavior of the IP component and how to manage interactions.

To execute an interaction, different atomic components send offer messages to the IP component. The received messages contains information about the local state of the atomic components which constitute a partial view of the global state of the system. If the IP does not take any decision, it will eventually receive all offers and know the global state. The simple and correct way for the IP to take decision on executing an interaction

in distributed implementation is to execute the interaction as soon as all participating components send an offer and are ready to execute it.

The following Figure 2.9 presents the IP component handling the set of interactions $\{ r_{request}, r_{confirm} \}$. Here the IP component correspond to a Petri-net that associates to each atomic component a token. The token position of each component in the IP represent the status of the component that can either be in:

— waiting place: Initially, and after each notification, the IP component does not know the state of the component until it sends an offer. In that case the IP is waiting for the component offer. There is one waiting place for each component. In Figure 2.9, the waiting place is labeled by $w\_$.

— received place: The token is there when an offer has been received, and the IP has not scheduled any interaction involving the component. There is one received place for each component. In Figure 2.9, the received place is labeled by $rcv\_$.

— sending place: The token is there when an interaction involving the component has been scheduled. There is one such place for each port of the atomic components, thus indicating which port is involved in the scheduled interaction. In Figure 2.9, there is a single sending place labeled by $s\_$.

Initially each component participating in an interaction handled with the IP have a token in a waiting place. Once an offer is received from a component, the token moves to the receive place and then interaction takes place when all tokens from the participating components are in receive places. After the interaction execution the tokens moves to the send places corresponding to the ports involved in the interaction. From the send place, the token moves back to the waiting place when sending the notification to the atomic component. Here, we are ready to formally define IP components behavior. For clarity sake, we denote by participants (γ) the set of components $B_i, \ldots, B_n$ is a participant in $a \in \gamma$ if it exports a port that is part of a. Besides, we denote $ports(\gamma)$ the set of ports occurring in an interaction $a \in \gamma$.

Let $C = \gamma(B_1, \cdots, B_n)$ be a composite component and $\gamma = \{a_i = (P_i, F_i, G_i) | a_i \in \gamma\}$ the set of interactions. Let $participants(\gamma)$ (resp. $ports(\gamma)$) be the set of atomic components (resp. ports) participating (resp. occurring) in interactions from γ.

Definition 8 (IP component) *The component* $IP = (L^{IP}, X^{IP}, P^{IP}, T^{IP})$ *handling* γ *is defined as:*

— *Set of places* L^{IP} *is the union of the waiting places* $\{w_i \mid B_i \in$

participants(γ)$\}$, *the receiving places* $\{rcv_i \mid B_i \in$ *participants*(γ)$\}$, *the sending places* $\{s_p \mid p \in$ *ports*(γ)$\}$ *and the trying places*$\{t_a \mid a \in \gamma$, *a externally conflicting* $\}$.

— *Set of variables* $X^{IP} = \{n \mid B_i \in$ *participants*(γ)$\} \bigcup \{\{x_p\} \cup X_p \mid p \in$ *ports*(γ)$\}$

— *Set of ports* P^{IP} *that contains:*

  — *the offer ports* $= \{o_i \mid B_i \in$ *participants*(γ)$\}$ *that are associated to variables* n, x_p, *and* X_p *from component* B_i.

  — *the notification send ports* $\{p \mid p \in$ *ports*(γ)$\}$ *associated to variables* X_p.

  — *the send ports* $\{r_a\}$ *and the receive ports* $\{ok_a, f_a\}$ *for each interaction* a *that is externally conflicting. The ports* $\{ok_a, f_a\}$ *do not have any variable attached while we associate to port* r_a *a set of participation numbers* $\{n_i\}_{i \in I}$ *where* I *is the set of all* B_i *components involved in interaction* a.

— *Set of transitions* $T^{IP} \subseteq 2^{L^{IP}} \times P^{IP} \times 2^{L^{IP}}$. *A transition* τ *is a triple* $({}^\bullet\tau, p, \tau^\bullet)$, *where* ${}^\bullet\tau$ *is the set of input places of* τ *and* $\tau^\bullet$ *is the set of output places of* τ. *We introduce three types of transitions:*

  — *receiving offers* (w_i, o_i, rcv_i) *for all components* $B_i \in$ *participants*(γ).

  — *executing interaction* $(\{rcv_i\}_{i \in I}, a, \{s_{pi}\}_{i \in I})$ *for each interaction* $a \in \gamma$ *such that* $a = \{p_i\}_{i \in I}$, *where* I *is the set of components involved in* a. *To this transition we associate the guard* $[G_a \wedge \bigwedge_{p \in a} x_p]$ *and we apply the original update function* F_a *on* $\cup_{p \in a} X_p$.

  — *sending notification* (s_p, p, w_i) *for all ports* p *and component* $B_i \in$ *participants*(γ).

  — *if the interaction* a *is externally conflicting the set of transitions will also contain:*

    — *reservation request* $(\{rcv_i\}_{i \in I}, r_a, \{t_a\})$ *guarded with* $\bigwedge_{p \in a} x_{pi} \wedge G_a$.

    — *positive response* $(\{t_a\}, ok_a, \{s_{pi}\}_{i \in I}$ *with update function* F_a

    — *negative response* $(\{t_a\}, f_a, \{rcv_i\}_{i \in I})$ *with no guard or update function*

By introducing IP engines to manage interaction, we were capable to decentralize the BIP model. However, introducing concurrency while executing component and interaction would require the understanding of more subtleties and complex issue related to resolving conflicts. Here, the interactions $\{rpush1, cget\}$ and $\{push2, cget\}$ are enabled simultaneously. Since these interactions share the same component B_i they can not be executed concurrently. We call such interactions conflicting. Obviously, distributed engines must ensure that conflicting interactions are mutually exclusive.

Conflict-free Interaction Management

Actually, interactions that are conflicting correspond to transitions that share a common input place. The conflict is solved as the Petri-net semantics ensures that if any one of these transitions executes, it consumes the token in the common input place, thus disabling the other transition. In particular, the semantics allows conflicts occurring between two interactions to be resolved locally, within the same IP.

Considering a set of conflict-free partitions of interactions and assume that a separate IP is built for each class of them. We define an IP_i component that manages the interactions of each partition γ_i from γ, the formal definition of conflicts is priviously given in Section 2.2.1 .

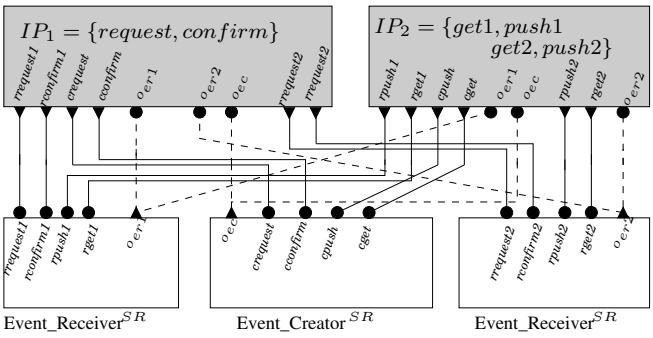

Figure 2.8 – Distributed model of the whens-up example with interactions conflict-free management.

The distributed version of the whens-up example presented in Figure 2.3 in the previous section is given in the Figure 2.8. The interaction set is divided in two separate subsets, each handled by a distinct IP component.

The solution of conflict-free partitioning has a drawback, because grouping interaction following a conflict-free criteria reduces drastically parallelism between interactions. In other words two interactions that are not indirect conflict cannot run in parallel. For instance let us consider the example in Figure 2.5. Assuming that a_1 is conflict only with a_2, and, a_2, a_3, a_4 are pairwise conflict. This situation leads to create only one IP component handling all the interactions. Hence, a parallel execution of the interactions a_1 and a_3 is no longer possible, despite that it is possible without violating the operational semantics of BIP . For this reason a third layer that implements a conflict resolution protocol is created and that we introduce in the following subsection.

As an example considering the IP_1 component handling the interactions

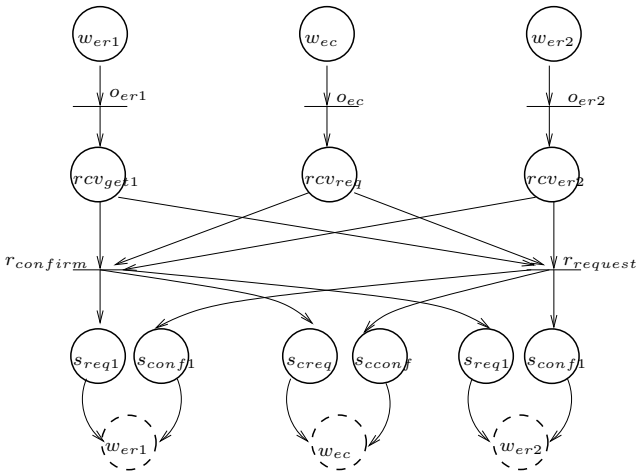

Figure 2.9 – The IP_1 component behavior of the whens-app application from figure 2.6

$\gamma_1 = \{request, confirm\}$ depicted in Figure 2.8. On the figure, all interactions are handled within the IP component. Here we involve only waiting w_a and receive rcv_a places to execute the interactions. Ones an interaction is executed a notification is sent to the involved atomic components.

Conflict-Resolution Protocol Layer

As previously explained, the main task of the CRP layer is to ensure that external conflicting interactions between two IP components are executed mutually exclusive. The implementation of the CRP component is based on the use of any algorithm that solves committee coordination problem [**?**].

The first distributed solution to the committee coordination problem assigns one manager to each interaction [**?**]. Conflicts between interactions are resolved by reducing the problem to the dining or drinking philosophers problems [**?**], where each manager is mapped onto a philosopher. Bagrodia [**?**] proposes an algorithm where message counts are used to solve synchronization and exclusion is ensured by using a circulating token. The principle is to keep the last offer number used for each component.Whenever a reservation arrives from the distributed IP component, each offer number from the request is compared against the value of the last offer number used in the conflict resolution protocol. If each number from the request is greater than the corresponding one in the conflict resolution protocol, the interaction is granted to execute. Otherwise execution is forbidden.

Here we adopt this solution to manage conflicting interactions handled by different *IP* components. The *Conflict-Resolution layer* components implements committee coordination algorithm and provide appropriate interfaces to connect with the *Interaction protocol layer* with a minimal restrictions. The first version implementing the behavior of a *CRP* component is a single process. Here we give the formal definition of the *CRP* component:

Definition 9 *Given a centralized BIP model* $C = \gamma(B_1, \cdots, B_n)$ *and an interaction partition* $\gamma = \gamma_1 \cup, \ldots, \cup\gamma$. *The CRP component* $IP = (L^{CP}, X^{CP}, P^{CP}, T^{CP})$*:*

X^{CP} *contains the last offer variable* N_i *for each component* B_i.

for each externally conflicting interaction a that involve a set of components invol(a):

- L^{CP} *contains the waiting place* w_a *and the treat place* tr_a.
- P^{CP} *contains the ports* rcv_a, ok_a *and* $fail_a$
- X^{CP} *contains the variables* $\{n_i^a \mid B_i \in invol(a)\}$. *The variables associated to the port* rcv_a *are* $X_{rcv_a} = \{n_i^a \mid B_i \in invol(a)\}$. *The ports* ok_a *and* $fail_a$ *do not have associated variables.*
- T^{CP} *contains the transitions: (1)* $\tau_{rcv_a} = (w_a, rsv_a, r_a)$, $\tau_(ok_a) = (r_a, ok_a, w_a)$ *and* $\tau_{fail_a} = (r_a, fail_a, w_a)$. *The transition* τ_{rcv_a} *has no guard and no update function. The transition* τ_{ok_a} *is guarded by* $G_{\tau_{oka}} = \bigwedge B_i \in invok(a)\ n_i^a > N_i$ *and its update function updates the* N_i *variables of the participants: foreach* $B_i \in participants(a)$ *do* $N_i := n_i^a$. *The transition* τ_{fail_a} *has no guard and no update function.*

As previously mentioned, the behavior of the *CRP* layer is based on the use of the message-count technique [**?**]. This technique is based on counting the number of times that a component interacts. Each component keeps a participation counter n which indicates the number of times a component have participated in interactions. The *CRP* ensures that each participation number is used only once. That is, each component takes part in only one interaction per transition. To this end, in the Conflict Resolution Protocol, for each component B_i, we keep a variable N_i which stores the latest number of participation of B_i. For each reservation message r_a received by the the *CRP* component to execute an interaction $a = \{p_i\}_{i \in participants(a)}$, a set of participation numbers $(\{n_i^a\}_{i \in participants(a)})$ for all participating components in a is received.

If for each component B_i, the participation number n_{ai} is greater than N_i, then the Conflict Resolution Protocol acknowledges successful reservation through port ok_a and the participation numbers in the Conflict Resolution Protocol are set to values sent by the Interaction Protocol. On the

contrary, if there exists a component whose participation number is less than or equal to what Conflict Resolution Protocol has recorded, then the corresponding component has already participated for this number and the Conflict Resolution Protocol replies failure via port f_a.

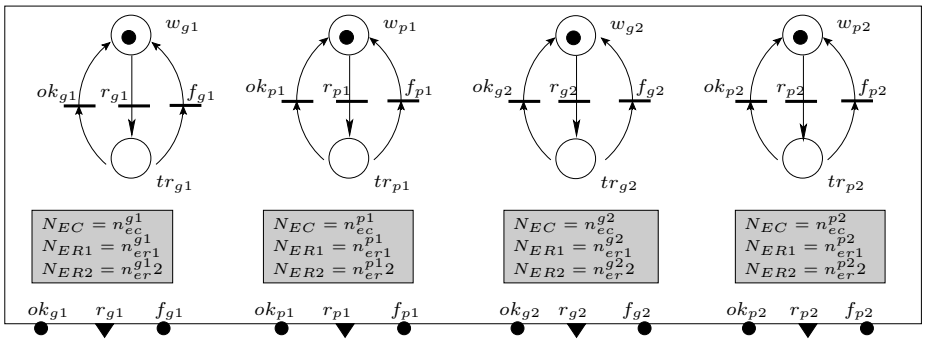

Figure 2.10 – Decentralized interaction management

The decentralized management of interactions relies on the additional conflict resolution layer to solve conflicts between different IP_j components. Usually, this layer implements a distributed algorithm that solves a committee coordination problem [**?**]. Several solutions have been presented in [**?**] and can be re-used directly in our context.

In this subsection, we define the interactions in the 3-layer model. Following Definition 6, we introduce S/R interactions by specifying the sender and the associated receivers. Given a composite component $C = \gamma(B_1, \cdots, B_n)$, and partitions $\gamma_1, \cdots, \gamma_m$ of $\gamma_i \subseteq \gamma$, the transformation produces a S/R composite component $C^{SR} = \gamma^{SR}(B_1^{SR}, \cdots, B_n^{SR}, (IP_1, \cdots, IP_m), \{CRP_i\})$. We define the S/R interactions of γ^{SR} as follows:

- For every atomic component B_i^{SR} participating in a set of interactions γ_i, for every IP components $IP_1, \cdots, IP_m$ handling γ_i, include in γ^{SR} the *offer* interaction $off = (B_i^{SR}.o, IP_1.o_i, \cdots, IP_m.o_i)$.
- For every port p in component B_i^{SR} and for every IP_{js} component handling an interaction involving p, we include in γ^{SR} the *response* interaction res_p=($IP_j.p$, $B_i^{SR}.p$)
- For every IP_j component handling an interaction a that is in conflict with an other interaction a' handled by some different IP, include in γ^{SR} the *reserve* interaction r=($IP_j.r_a$, $CRP.r_a$). Likewise, include in γ^{SR} the *ok* interaction ok=($CRP.ok_a$, $IP_j.ok_a$) and the *fail* interaction f=($CRP.f_a$, $IP_j.f_a$).

Figure 5.3 illustrates the *IP_1* component constructed for the set of interactions $\gamma_1 = \{request, get1, push2\}$ for the example shown in Figure

2.6. For all B_i components involved in interactions γ_1, we introduce a waiting (w_a) and receiving (rcv_a) places. The interactions $get2$ and $push2$ are in external conflict with the interaction $get1$ and $push1$ respectively, hence when executing both interaction we involve try_a try places. For all ports p involved in γ_1 we introduce a sending place s_{p_i}. The IP component moves from w_i to rcv_i whenever it receives an offer from the corresponding component B_i After choosing and executing interactions, the IP component moves to sending (s_p) places to send notification through ports p to the corresponding component. In the case of conflicting interactions $get2$ and $push2$, we introduce a r_a transition used to communicate with the upper layer to resolve conflict. The response of the execution request sent by the IP component can either positive through execution of the ok_a transition or either negative executing a fail transition f_a.

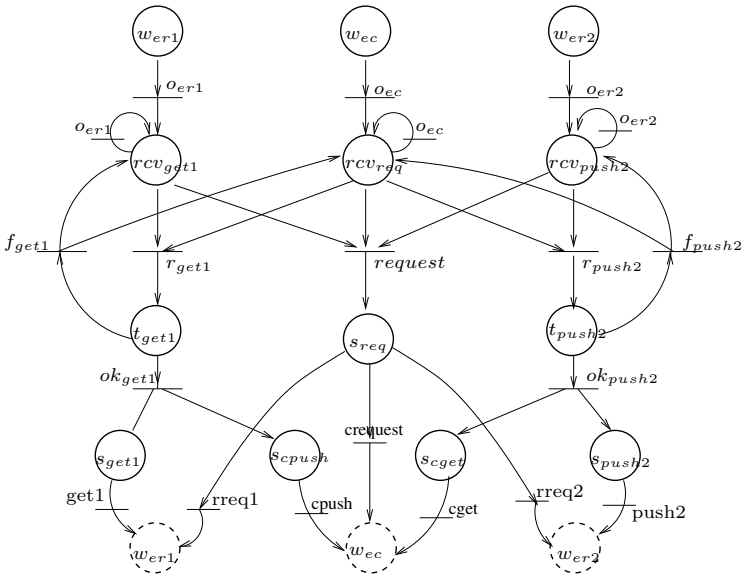

Figure 2.11 – The IP_1 component behavior of the whens-app application from figure 2.6

2.2.3 Functional Equivalence

In this section, we recall from [**?**] the proofs showing the correctness of the target S/R model. First, we show that the target model is a valid S/R model, that is, (1) only S/R or unary interaction are allowed, (2) each send port participates in exactly one S/R interaction and (3) whenever a

send port is enabled, the associated receive port becomes unconditionally enabled within a finite set of transitions in the receiver component. Lemma 1 proves a stronger assumption states that the receive port is enabled as soon as the send port becomes enabled.

Lemma 1 *Let $\gamma^{SR}(B_1^{SR}, \ldots, B_2^{SR}, IP_i, CRP_i)$ be the S/R model obtained from $\gamma(B_1, \ldots, B_n)$. Then, at each reachable stateof the S/R model, whenevera send-ports is enabled, the associated receive-port r is already enables.*

As explained earliers, each atomic component start listening to notifications directly after sending an offer. Dually, *IP* componentwait for offers from a given atomic component as soon as it sends a notification to that component.

Proof 1 *Let B_i^{SR} be a S/R atomic component. We show that all S/R interactions involving B_i^{SR} meet the statement of the lemma. Recall that to each atomic component we associate a token in the engine. Within the execution of the engine, the token can be either in a waiting place, in a receive place or in a sending place. Hence we can distinguish three configurations of the atomic component state and it's corresponding token:*

- *Initial state: B_i^{SR} is in a busy state and the corresponding token is in the waiting place w_i. The send-port(in atomic component) o_i is enabled as well as the receiveport(in IPcomponent). Genarlly, the property holds for configurations of this form, in particular the initial configuration. The excution of this configuration lead as to the next one.*
- *B_i^{SR} is in a stable state and the associated token is in the receiving pace r_i. From this configuration, no send-port involving a communication between B_i^{SR} and $\{IP\} \cup \{CRP\}$ is enabled. Only the token in the engine can move, provided a unary interaction is executed.*
- *B_i^{SR} is in a stable state and the associated token is in a sending place s_p. With such configuration, the send port p is enabled. By construction, the s_p state can be reached only if the variable x_p was previously set to true by B_i^{SR} before sending the offer only if the variable x_p was previously set to true by B_i^{SR} when the offer was sent. Thus, the S/R interaction between send and receive ports is possible and by executing it, we reach back the first configuration.*

2.3 Conclusion

In this chapter, we presented the *BIP* component-based model, a formalism for modeling heterogeneous systems with synchronized multi-party

interactions. A system is described by assembling a set of atomic components communicating through set of interfaces (ports).

The second part of this chapter, we presented method to automatically decentralize *BIP* models. This method transform an arbitrary *BIP* model into a send/receive model directly implementable on a distributed platform. This transformation consist on adapting a 3-layer model architecture that consists on (1) breaking the atomicity of actions in atomic components of the centralized *BIP* model, (2) introducing scheduler IP components to coordinate the execution of interactions in distributed way according to a user partitioning and (3) using CRP components to resolve conflicts between schedulers.

Component-based models are highly recommended to reason about security properties at an abstract level in parallel with functional behavior modeling. For this reason, we extend, in the next chapter, the *BIP* model to handle information flow security. Where we formally define a set of sufficient conditions to preserve security in both models.

Chapter 3

Information-Flow Security

Contents

In this chapter, we present the security property targeted in this work, non-interference. We first briefly cite different definitions given in literature for non-interference informally and then we present distinct approaches used to tackle the security issues defined using this property according to different system types.

3.1 Non-Interference Definitions

Noninterference is a model for information-flow control(CIF), firstly defined by Goguen and Meseguer [**?**], which ensures that sensitive data does not affect the publicly visible system behavior. This security feature allows tracking of information in the system and the establishing of end-to-end confidentiality and integrity properties. Here, we informally define the non-interference property as given in [**?**], but in order to verify that a system satisfy it, we need a formal definition, which we give in the next Chapter 4. Gogen and Mesugers presented the non-interference property using a model of deterministic state machine that accepts commands from its various users as input, processes them using transition functions and return outputs according to output functions. For a given system, the non-interference property holds between two users U_1 and U_2 (denoted $U_1 \not\rightsquigarrow U_2$) if the output given on U_2 does not depend whether U_1 provides input to the system or not. More precisely, $U_1 \not\rightsquigarrow U_2$ holds if for every sequence of commands the output to U_2 after execution of this sequence is identical to the output to U_2 after execution of the purged sequence, i.e the sequence that results after removing all commands issued by U_1, [**?**]. This reduces the non-interference to a problem of indistinguishability between multiple execution traces of the system (e.g, states of memory), while considering that the network is abstract to a shared memory. Numerous further variant of non-interference have been proposed over the last 20 years [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Although that these last definition were given different names like, non-deducebility, generalized non-interference, restrictiveness or non-interference, they all share the same underlying idea of non-interference.

The intuitive way to explain the non-interference property violation may be shown in program code. Hereafter, we present two examples of security leaks that are the explicit and the implicit flow, where we consider that variables h is a sensitive data while l is a public one. The illegal **explicit** flow can be summarized in the instruction ***l:=h*** that assign a secret variable h to a public one l. The **implicit** flow is presented in the conditioned instructions, i.e, ***if h then l=True; else l=False;***, where

the value of the public variable is modified within a context of a secret variable h. Such illegal flow is a kind of covert channels that can not be managed properly by a purely dynamic mechanism such as access control.

3.2 Non-Interference Verification: Methods and Approaches

Many formal models and tractability techniques have been used in the literature to deal with information flow leaks issues. Some of these techniques are based on the use of formal models which define in precise terms the system behavior as well as properties, allowing to specify requirements without ambiguities and verifying critical requirements with mathematical rigor. While formal models are used for verification at abstract level, security typed languages are used to analyze programs and verifying security at code level. In this thesis, we use a combination of both approaches to track and secure critical information at data as well as event levels.

3.2.1 Formal Trace-Based Security Model

A formal security model is a formal specification of system's security requirements. As depicted in Figure 3.1, the system model component that specifies *how* the system operates, interpreted in a specific formalism and a security component which specifies *what* security property is required. The verified satisfaction relation between both components ensures that the security requirements are fulfilled by the system.

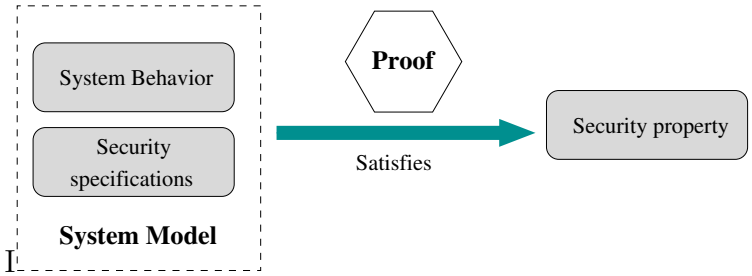

Figure 3.1 – Structure of formal security model.

Formal methods are related to secure system model (security and behavior components) based on a common semantic or a syntactic specification formalism. As presented in the previous Chapter 2, the system model that

we employ through out this thesis is a component-based model, *BIP* , where the behavior can be expressed by a set of traces where every trace models a possible execution sequence of the system. Formally, a trace is a sequence of events where every event models an atomic step execution (e.g. sending or receiving a message). In trace-based system models several system properties such as safety (e.g. invariants) and aliveness (e.g. termination) properties can be modeled by a set of traces and are referred to as a *property of traces*. Such properties are satisfied if every possible trace of the given system is contained in the set that specifies the property [**?**]. Based on this, the security property that we treat in this thesis, non-interference, is modeled by a *set* of *set of traces*.

The trace-based interpretation of non-interference property have been often employed with event system. These systems were quite popular for the investigation of information flow security properties [**?**, **?**, **?**, **?**]. The particularity of *BIP* model that we employ in this thesis is the use of a set variables in atomic component. Hence, the security properties that we give using trace-based definitions are not limited to event-traces but we extended to observation of variables variation with different execution traces.

3.2.2 Security Typed Languages

The most popular security techniques, used earlier in the 80s, to secure distributed system are access control techniques or deny access. Despite there wide use in practice [**?**, **?**, **?**], these techniques rely only on defining *who* has the right to read or modify a variable or a resource over the network instead of defining *what* exactly it is accessed to read or to write. For example, to prevent an authorized processes to access a shared file they restrict write-read permissions defined in role-based dictionaries(tables) that may be very complicated to put in place in case of complicated systems. As a useful complement to traditional security mechanisms such as access control and cryptography, security typed language provides an end-to-end protection allowing to enforce different security policies. Critical information are checked not just at a certain point (e.g. system output) but throughout the duration of computation. The control of information flows for typed languages, uses the program code to discover security leaks. It was introduced for the first time by Denning [**?**]. The use of typing aim to ensure that a program may not contain an illegal information flow compared to the effective flow of policy. The idea is to associate security levels on different variables of the program and then check that no assignment can occur in a high security level variable to another lower security level. Dif-

ferent implementation of typed languages have been given, particularly we mention the *JIF* language [**?**, **?**]. *JIF* is an extension of the *Java* language to control information flow using a *Decentralized Label Model* [**?**]. The labels (annotations) associated to data in the code represent a security policy that restricts the flow of information in the system. The verification can be done through annotations check at compile time verifying if the program meets a set of constraints that statically regulates the information flow, or at run-time execution to enforce these constraints.

3.2.3 Security Labels

Security labels are used to annotate program variables in order to control the flow of information in the programs. The label of a variable controls how the data stored in this variable can be used. If the contents of one variable affects the other variable is that there is a flow of information from the first to the second. The key to protecting the confidentiality and integrity is to ensure that when the data flows through the system, the security labels are consistent with the defined policy. There are several models of labels in the literature. Hereafter we give example to a three of them, [**?**, **?**, **?**].

Decentralized Label Model

In this section we describe the Decentralized Label Model (DLM) [**?**]. This model is said to be "decentralized" because security policies are not defined by a central authority, but controlled by the various participants in the system. The system will then behave in ways that respect these security policies. This feature is possible thanks to the concept of "ownership" of the labels used to annotate the data. This concept allows each participant to define the security level of its data, but not those of others. To protect secret information, Labels are expressed within a set of constraints. Each constraint is defined using *principals* representing authorities that control the information to protect.

Authority The authorities are the entities that own, modify and publish the information. They represent users, groups or roles. A process has the right to act on behalf of a group of authorities. Some authorities have the right to act for others. When A authority can act to another authority A', A has all the power and potential of privileges A'. The relationship is for is reflexive and transitive, and to define a hierarchy or a partial order between

authorities. The relationship *acts_for* allows to delegate all the privilege of an authority (A) to another (A'). We denote such relationship by ($\prec$).

Labels To track data information flow in programs, authorities uses labels as security constraints. A security label can be either a confidentiality label or an integrity label. A constraint in a security label has two parts : an *owner* and a set of *Readers* in confidentiality label or *Writers* in an integrity label. The *owner* represent the authority that owners the data and the set of *Readers* and *Writers* represent authorities that can respectively access the information to read or to modify.

A label consists of two major parts: a privacy label and an integrity label. Each of these labels contain a set of elements that express need of security defined by authorities. A label element has two parts: an owner and set of authorities that represent players in a privacy label and writers in an integrity label. The purpose of a label element is to protect the private information of the owner of the item. It is called the Data Use Policy. Thus, data is labeled with a number of policies set by the data owners authorities.

$$L_c = \{o_1 : r_1; o_2 : r_1, r_2\} \qquad (*)$$

The confidentiality label gives the security level wanted to data by designating the potential readers list. For instance, considering a confidential label $L_c *$. The L_c label contains two security policies separated by semicolon where o_1, o_2, r_1 and r_2 are all authorities. Both o_1 and o_2 are owners of the label policies and $\{r_1\}$ and $\{r_1, r_2\}$ are respectively the set of readers of the policies. Actually, readers are the authorities to which this element provides access to the data. So the owner is the source of the data and its readers are possible recipients. The authorities are not listed as readers do not have the right to access the data. As the data flows in the system, the label must be respected.The meaning of a label is that every policy in the label must be respected as the data flows through the system, so that each tagged information is not disclosed only following a consensus between all policies. A user can read the data only if this user representative authority *can_act* for each political reader in the label. Thus, only users whose authorities may act to r1 can read the data labeled by L_c. The same authority may be the owner of several policies; policies are enhanced independently of one another in this case.

The integrity policy are dual privacy policies. As privacy policies protect against improperly read data, even if they cross or are used by programs that do not trust, the integrity policies protect data against inap-

propriate modification. A label integrity keeps track of all the sources that affected its value, even if these sources affect it indirectly. They prevent untrusted data to have an effect on the stored data confidence. The structure of an integrity policy is identical to that of a confidential policy. It has an owner, the authority for which the policy is applied, and a group of writers (instead of readers), the authorities who are authorized to modify the data. A label integrity can contain a set of integrity policies with different owners.

A security domain is defined over the set of confidentiality labels and a partial order relation. The partial order relation is the *flows to* relation, denoted $\subseteq$ and defined as follows:

$$\begin{aligned} L_1 \subseteq L_2 \quad \equiv \quad & \forall o_1 \in O(L_1).\ \forall o_2 \in O(L_2).\ o_1 \prec o_2 \wedge \\ & \forall r_1 \in R(L_1, o_1).\ \exists r_2 \in R(L_2, o_2).\ r_1 \prec r_2 \end{aligned}$$

The intuition behind the *flows to* relation $\subseteq$ above is that *(1)* the information can only flow from one owner o_1 to either the same or a more powerful owner o_2 where o_2 can act for o_1 and *(2)* the readers allowed by $R(L_2, o)$ must be a subset of the readers allowed by $R(L_1, o)$ where we consider that the readers allowed by a policy include not only the principals explicitly mentioned but also the principals able to act for them.

Token-Based Label Model

The following model, inspired by [**?**, **?**] represents the labels as a set of chips (tags). A token is an opaque term, out of context, has no precise meaning, but that is assigned to the data to associate them with some type of confidentiality or integrity. The label belonging to a set of security levels L and contains two sets: S (for Secrecy), which represents the level of privacy and I (for Integrity), which represents the level of integrity. We then note $L = \{S; I\}$.

Associate a confidentiality token j ($j \in S$) at a given data implies that this data contains information of a confidentiality level j. To reveal the contents of this given data, the system must obtain the approval for all of privacy chips that tag this data. The level of integrity of a data represents a guarantee of the authenticity of the information in this database. It allows the recipient to ensure that the data that was sent to it was not modified by untrusted parties. Assign a token of integrity i ($i \in I$) for a given data represent an additional guarantee for this data.

Definition 10 (Order Relation) *All labels in a system are governed by the partial order relation can_flow to (can flow to), denoted by $\subseteq$. Following*

this relationship, the flow is directed from the least restrictive to the most restrictive label. The relationship can_flow to can be decomposed into two equations:

— $\subseteq_C$*: This relationship orders confidentiality levels*

— $\subseteq_I$*: This relationship ordered integrity levels*

Considering two security labels $L_1 = \{S1; I1\}$ *and* $L_2 = \{S2; I2\}$*. We denote:*

$$L_1 \subseteq L_2 \quad if\ and\ only\ if \quad S_1 \subseteq_C S_2 \ \ and \ \ I_1 \subseteq_I I_2 \tag{3.1}$$

Definition 11 (Confidentiality) *Considering two confidentiality levels* S_1 *and* S_2 *each containing a set of tokens:*

$$S_1 \subseteq_C S_2 \quad if\ and\ only\ if \quad \forall j_1 \in S_1\ ,\ \exists j_2 \in S_2 \ \ such\ that\ j_1 = j_2 \tag{3.2}$$

Definition 12 (Integrity) *Considering two Integrity levels* I_1 *and* I_2 *each containing a set of tokens:*

$$I_1 \subseteq_I I_2 \quad if\ and\ only\ if \quad \forall i_2 \in I_2\ ,\ \exists i_1 \in I_1 \ \ such\ that\ i_1 = i_2 \tag{3.3}$$

Theorem 2 (Union) *Considering two security labels* $L_1 = \{S_1, I_1\}$ *and* $L_2 = \{S_2, I_2\}$*. The Union of both labels is* $L = \{\}$*:*

$$L_1 \cup L_2 = L \quad \Longleftrightarrow \quad S = S_1 \cup S_2 \ \ et \ \ I = I_1 \cap I_2 \tag{3.4}$$

Corollary 1 *If* L_1 *and* L_2 *are two Labels in L, then* $L_1 \cup L_2 \in L$ *such that:*

$$\forall L_i \in L,\ if\ L \subseteq L_1\ and\ L \subseteq L_2\ then\ L \subseteq L_1 \cup L_2 \tag{3.5}$$

Theorem 3 (Transitivity) *For two security levels,* L_1 *and* L_2

$$If \ \ L_1 \subseteq L_2 \ \ and \ \ L_2 \subseteq L_3 \ \ then \ \ L_1 \subseteq L_3 \tag{3.6}$$

Theorem 4 (Reflexively) *For two security levels,* L_1 *and* L_2

$$If \ \ L_1 \subseteq L_2 \ \ and \ \ L_2 \subseteq L_1 \quad \Longleftrightarrow \quad L_1 = L_2 \tag{3.7}$$

Based on the use of the previous definitions and theorems, we can define security domain as follows:

Definition 13 (security domain) *A security domain is a lattice of the form* $\langle S, \subseteq, \cup, \cap \rangle$ *where:*

— S *is a finite set of security levels.*

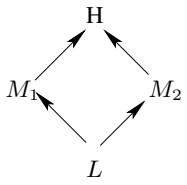

Figure 3.2 – Example of security domain

— $\subseteq$ *is a partial order "can flow to" on a set of security levels S that indicates that information can flow from one security level to an equal or a more restrictive one.*
— $\cup$ *is a "join" operator for any two levels in S and that represents the upper bound of them.*
— $\cap$ *is a "meet" operator for any two levels in S and that represents the lower bound of them.*

As a trivial example, consider the set $S = \{L, M_1, M_2, H\}$ of security levels that are governed by the relation $L \subseteq M_1$, $L \subseteq M_2$, $M_1 \subseteq H$ and $M_2 \subseteq H$. M_1 and M_2 are incomparable and we note $M_1 \nsubseteq M_2$ and $M_1 \not\supseteq M_2$. Edges between the different security levels represent the way the information have the right to flow. This security domain is graphically illustrated in Figure 3.2.

3.2.4 From Verification to Implementation

The application of formal approaches that we have cited above appears to be the most appropriate in order to ensure security. However, the implementation of formal model is difficult, error-prone, time consumption and expensive task. The implementation of information flow control solutions for centralized systems are mostly based on flow analysis, conducted with safety typed languages that intermingle security policy with the functional application code, and so help create non-interfering systems construction. Such languages have the advantage of treating the flow of information at a fine-grained level, but it suffers from certain drawbacks: the security constraints and the functional behavior of the system are not separated, which implies that, in hand, the developer must be sufficiently versed in security to enforce these constraints himself on the code, and secondly, he must undergo a fairly heavy training phase to take over the new language.

Distributed solutions for verification information flow is either based on applying access control techniques, which could allow certain information

leaks due to the implicit flows, for example, or by distributing the system after applying constraints safety, dividing it into non-interfering mutually modules. The problem is that the systems created by these solutions are distributed according to the security constraints, not as functional constraints. In addition to this, the non-interference property still a difficult property to implement, restrictive and considered to be unpractical which make it not always desired by the system designers.

It is for these reasons that a solution usable for non-interference in real systems must be flexible, integrated and easy to apply, especially for the non-expert on security. It must also be able to combine other properties (i.e, safety and aliveness properties) with non-interference. Enforcing a total separation between functional modeling and verification and the implementation of security mechanisms is the most promising approach while treating non-interference. Indeed, the application of non-interference in complex systems is a difficult task because the information can take many forms and operate on several types of channels, which makes it difficult to follow. The verification made on an abstract level and followed by set of secure-by-construction transformations to decentralize the model, allows to efficiently introduce security mechanisms (i.e, encryption, signature tokens) while generating the application code. Hence, we can reach an optimal security configuration. The use of component-based systems to treat non-interference is very adequate where, thanks to their modularity and interfaces, we can decompose the security issue which facilitate the interference detection and easily add security mechanisms.

3.3 Security Verification Techniques and Languages

Several solutions to tackle different security issues in distributed systems exists in literature. Most of theses solutions treat security in distributed systems with separation of concerns in an ad-hock manner, where security and functional specifications are treated separately. Hereafter, we give examples of different systems that treat security at a different levels, either at an abstract level or with separate modules for the implementation of security mechanisms for execution.

3.3.1 Security Abstraction and Configuration

Several solutions based on high-level security configuration have been developed. This work, like ours, are based on a separate configuration

code to describe non-functional properties such as security and generate the appropriate code. Hereafter, give examples of models ensuring security at abstract level, the SecureUML and JASON.

SecureUML

SecureUML is a modeling language allowing to specify security requirements in arbitrary UML design models using access control techniques. [**?**] proposes a scheme to a different security concepts provided the use of Role-Based Access Control (RBAC) that restricts access to authorized users. SecureUML is based on the MDS approach, where starting from extension of the UML model encompassing security using Access Control techniques, a set of transformations are done till a platform dependent implementation. This approach is very used especially with target platforms including EJBs (Enterprise Java Beans), Enterprise Services for .NET and Java Servlets.

JASON

Jason was introduced in [**?**] as a canevas with a compiler that aims to simplify adding security policies to services. The system developer uses the Java annotations (defined in JIF) to define and represent multiple security policies in the code such as confidentiality, integrity and RBAC access-control. Next, a cryptographic code is automatically generated by the JASON canvas. Hence, the system designer can focus on functional specifications of the application.

3.3.2 Component-Based Security Systems

Standard security approaches for distributed systems, we have cited above, are essentially based on the access control policies that control the execution of actions on individual objects. However, access control policies suffer from the Trojan Horse problem or other information leakage using hidden channels. This is due to the fact that no check is made on the use of a given once issued to authorized service. Inappropriate calculation of confidential information and calls to third-party services may disclose secrets to other entities that are not authorized to read this information. The Mandatory Information flow control can solve some of these problems by labeling every information and service with a given level of security.

CIF Based Architecture: CIF aims to represent the distributed systems using a component oriented architecture, applying the CIF technique by assigning parameters to elements of the system at a higher level of abstraction. In the CIF project, the model base component used is an extended Fractal model that expresses security properties in addition to the architecture. A security interface has been added allowing to mark the inter-component connections at the ports where a pair of confidentiality and integrity appointed label that represent security constraints imposed on traded variables. The label applies to a port is applied to all messages sent (or received) via this port. Each component can have a set of attributes, each with a security label.

CIF uses the ADL fractal extended language to present the security configuration in addition to the architecture of the system. Figure 3.3 represents an example of a description in ADL CIF. Compared with the Fractal ADL, the ADL CIF allows the specification of the component data labels (attributes), the labels of communication ports and the desired communication protocol. The description of the protocol guides the administrator for the selection of the security protocol and communication between components. In addition, the bonds in the ADL CIF are considered unidirectional because we need to distinguish between an inquiry and a response that may have different security requirements, while a bond may be bidirectional in Fractal.

```
<component name="C">
  <component name="C1">
    <port name="P" role="client" signature="security.pltf"
        label="{C:I}"/>
    <attribute name="M" label="{Cm:Im}" value="mon message"/>
    <content class="security.C1Impl">
  </component>
  <component name="C2">
    <port name="P" role="server" signature="security.pltf"
        label="{C':I'}"/>
    <content class="security.C1Impl">
  <binding client="C1.P" server="C2.P">
    <protocol name="RMI"/>
  </binding>
  </component>
</component>
```

The specification of the security features is achieved at a higher abstraction level. It is done in the ADL language by using security tags assigned to the component interfaces. Annotating a component is where the component attributes, incoming and outgoing messages through the

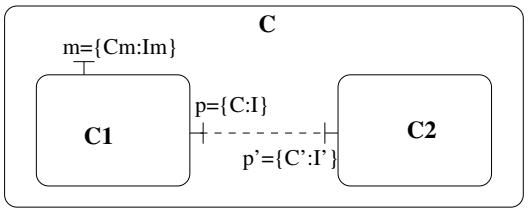

Figure 3.3 – A simple ADL CIF example.

communication port are tagged with security levels. The annotation mechanisms forces the system to comply with the security restrictions at two levels:

— Intra-component level: Security property has to be satisfied locally at each component. Confidentiality and integrity is verified at compilation and at code generation for each component.

— Inter-component level: Exchanged messages that are classified confidential have to be kept secure in a non-secure network. To define the security constraints on the exchange of messages between components, the communication ports are annotated with labels. The semantics of different labels is defined according to the port type either it is a server or client.

3.3.3 Distributed Implementation Solutions

The solutions presented earlier mainly concern centralized systems, and do not address the problems that can cause distribution of the program on an unreliable network. Some of these solutions, as JIF were extended to accommodate distributed systems.

JIF/Split: The JIF/Split is the extension of the JIF language by [**?**] to allow automated partitioning of program across distributed and mutually hostile hosts, while preserving information flow security across the entire resulting ensemble. According to a trust relation (privilege) expressed by principals to hosts, Jif/split can partition a program such that a code fragment that executes on a host only has access to data owned by principals who trust that host. Hence, sub-programs reproduce original program with added satisfaction to security requirement without the need of a universally secure machine.

Fournet Compiler [?]: Recently and similarly to Jif/Split, [**?**] presented a compiler that distributes programs with a strengthening of security communications by adding cryptographic mechanisms. This compiler is built for a simple imperative language with security annotations and distributed code linked to concrete cryptographic libraries. The compiler presented here is structured into four stages. The first is the partitioning, it can cut the sequential code into a set of routines following the annotations applied by the Security designer. The second step is the control flow which can protect the program against malicious scheduler in a code that keeps track of the status of this program. The replication step transforms a distributed program based on a shared memory in a program where the variables are replicated at each node of the system. Finally, the last step is cryptography, which inserts encryption operations to protect these replications and generates an initial protocol to distribute their keys.

Fabric: Fabric [**?**] is a platform for building secure distributed systems that promotes secure resource sharing between heterogeneous nodes that do not necessarily trust. The nodes are divided into storage nodes (storage nodes), dissemination (dissemination nodes) and computing (worker nodes). These nodes share a set of objects Fabric, which are similar to Java objects tagged with the DLM model labels. The Fabric language is also based on an extension of the JIF language for distributing programs with support for transactions and for remote method invocation.

3.4 Conclusion

The main objective of our work is to ensure non-interference with fine granularity for distributed systems based on a sound and rigorous verification technique. None of the solutions we have presented above handles the non-interference issues from centralized to distributed systems at high level of abstraction till secure code generation. In this respect, we propose a framework equipped with the necessary practical and automation tools for control information flow in the system and generating secure code.

Chapter 4

Secure Component-Based Model

Contents

As presented in the state of the art, security insurance is more related to problems in design and implementation rather than vulnerabilities in cryptographic mechanisms. Therefore, considering a rigorous model with well-defined semantics allows to formally verify a system with respect to mathematical descriptions of security(properties).

As presented in the previous Chapter 2, component-based modeling approach allows to build complex systems from assembling simple components. It can be formalized as an operation that takes in components and their integration constraints. From these, it provides the description of a new, more complex components. Different part of the system can be treated independently which saves time and cost and increases productivity via component reuse. Modeling is henceforth seen as a matter of re-configuring and reorganizing existing components, which is not always trivial and should not be neglected. Introducing security requirements at an abstract level and integrate it in the design process aim to simplify the verification and detecting leaks at early stages at system creation. Hence, models are a combination of security and functional requirements, where the information flow are controlled following set of predefined constraints. As our modeling languages have a well-defined semantics, we can formally analyze these designs. Consequently, once the system is proved secure, we can use tools to automatically generate code directly from the model. In this context, we introduce *secureBIP* framework for verifying information flow in component-based systems. This is built as an extension of the component-based modeling formalism *BIP* [**?**, **?**].

This chapter is organized as follows. In Section 4.2.2, we introduce the proposed security extension, *secureBIP* , with focus on the definitions of non-interference property. Then in Section 4.4, we provide formal steps towards the usability of automated method based on the use of unwinding theorem, providing sufficient conditions for non-interference. These conditions verify the security of system composition (atomic behavior and interactions) providing a key factor in the scalability of our approach, which is also an important criterion for the success of verification in realistic settings. In Section 18, we improve the verification technique to a practical security level synthesis allowing to generate security configuration for the model. We then illustrate the use of *secureBIP* in Section 4.7.

4.1 Security Model

In order tackel security issues related to sensitive information flow in early design phases, we extend previously defined *BIP* model with specific

mechanisms to track and analyse security. Here we present the *secureBIP* component-based model that will constitute the foundation upon which we will build our approachs for building secure systems. We explore information flow policies [**?**, **?**, **?**] with focus on the non-interference property. *SecureBIP* was first introduced in [**?**]. It allows, first, to defne a security policy, by classifing different part of the model (data variables, ports and interactions) with annotations describing how information can flow from one classification with respect to the other, and second, to put in place a practical security analysis approach by defining a set of sufficient conditions. Hereafter, we present the adapted security mechanisms to define the non-interference property and statically verify it.

We consider that the used labelling model forms a complete lattice. Based on this, we formally represent security domains as finite lattices $(\subseteq, \leq)$ where $\subseteq$ denotes the security levels and $\leq$ the *flows to* relation. For a level s, we denote by $[-, s]$ (resp. by $[s, -]$) the set of levels allowed to flow into (resp. from) s. Moreover, for any subset $S \subseteq \mathbb{S}$, we denote by $\sqcup S$ (resp. $\sqcap S$) the unique least upper (resp. greatest lower) bound of S according to $\leq$.

Let $C = \gamma(B_1, \dots B_n)$ be a composite component, fixed. Let X (resp. P) be the set of all variables (resp. ports) defined in all atomic components $(B_i)_{i=1,n}$. Considering $(S, \subseteq, \leq)$ a security domain, fixed.

Definition 14 (security assignment) *A security assignment for component C is a mapping $\sigma : X \cup P \cup \gamma \to S$ that associates security levels to variables, ports and interactions such that, moreover, the security levels of ports matches the security levels of interactions, that is, for all $a \in \gamma$ and for all $p \in P$ it holds $\sigma(p) = \sigma(a)$.*

Figure 4.1 shows an example of an annotated atomic component. We use two intuitive security levels H and L, where $L \subseteq H$. Here, we tag each port with a security level, presented in dashed square, where we tag variables $cnotif$ and $cinfo$ with the same security level H.

In atomic components, the security levels considered for ports and variables allow to track intra-component information flows and control the intermediate computation steps. Inter-components communication, that is, interactions with data exchange, are tracked by the security levels assigned to interactions.

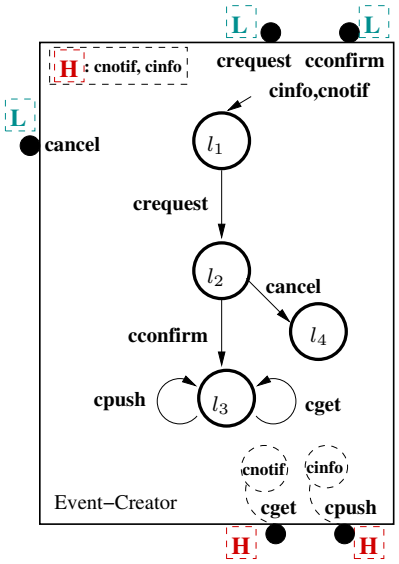

Figure 4.1 – Example of an annotated atomic component, Event-Creator.

4.2 Non-Interference

Let σ be a security assignment for C, fixed. For a security level $s \in S$, we define $\gamma{\downarrow_s^\sigma}$ the restriction of γ to interactions with security level at most s that is formally, $\gamma{\downarrow_s^\sigma} = \{a \in \gamma \mid \sigma(a) \subseteq s\}$.

For a security level $s \in S$, we define $w|_s^\sigma$ the projection of a trace $w \in \gamma^*$ to interactions with security level lower or equal to s. Formally, the projection is recursively defined on traces as $\epsilon|_s^\sigma = \epsilon$, $(aw)|_s^\sigma = a(w|_s^\sigma)$ if $\sigma(a) \subseteq s$ and $(aw)|_s^\sigma = w|_s^\sigma$ if $\sigma(a) \not\subseteq s$. The projection operator $|_s^\sigma$ is naturally lifted to sets of traces W by taking $W|_s^\sigma = \{w|_s^\sigma \mid w \in W\}$.

For a security level $s \in S$, we define the equivalence $\approx_s^\sigma$ on states of C. Two states q_1, q_2 are equivalent, denoted by $q_1 \approx_s^\sigma q_2$ iff (1) they coincide on variables having security levels at most s and (2) they coincide on control locations having outgoing transitions labeled with ports with security level at most s. We are now ready to define the two notions of non-interference.

4.2.1 Event Non-Interference

Definition 15 (event non-interference) *The security assignment σ ensures event non-interference of $\gamma(B_1, \ldots, B_n)$ at security level s iff,*

$$\forall q_0 \in Q_C^0 : \quad \textsc{traces}(\gamma(B_1, \ldots, B_n), q_0)|_s^\sigma = \textsc{traces}((\gamma{\downarrow_s^\sigma})(B_1, \ldots, B_n), q_0)$$

Event non-interference ensures isolation/security at interaction level. The definition excludes the possibility to gain any relevant information about the occurrences of interactions (events) with strictly greater (or incomparable) levels than s, from the exclusive observation of occurrences of interactions with levels lower or equal to s. That is, an external observer is not able to distinguish between the case where such higher interactions are not observable on execution traces and the case these interactions have been actually statically removed from the composition. This definition is very close to Rushby's [**?**] definition for transitive non-interference. But, let us remark that event non-interference is not concerned about the protection of data.

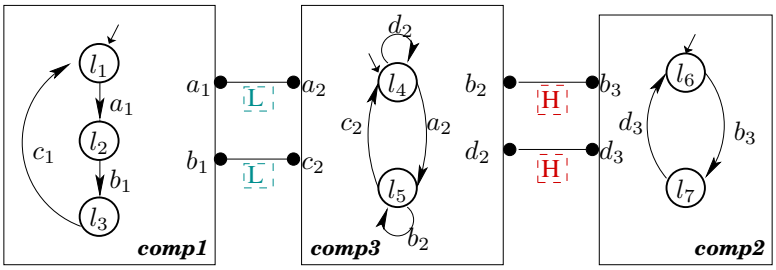

Figure 4.2 – Example for event non-interference.

Example 4 *Figure 4.2 presents a simple illustrative example for event non-interference. The model consists of three atomic components* $comp_{i,i=1,2,3}$. *Different security levels have been assigned to ports and interactions:* $comp_1$ *is a low security component,* $comp_2$ *is a high security component, and* $comp_3$ *is mixed security component. The security levels are represented by dashed squares related to interactions, internal ports and variables. As a convention, we apply high (H) level for secret data and interactions and low(L) level for public ones. The set of traces is represented by the automaton in figure 4.3 (a). The set of projected execution traces at security level L is represented by the automaton depicted in figure 4.3 (b). This set is equal to the set of traces obtained by restricted composition, that is, using interaction with security level at most L and depicted in figure 4.3 (c). Therefore, this example satisfies the event non-interference condition at level L.*

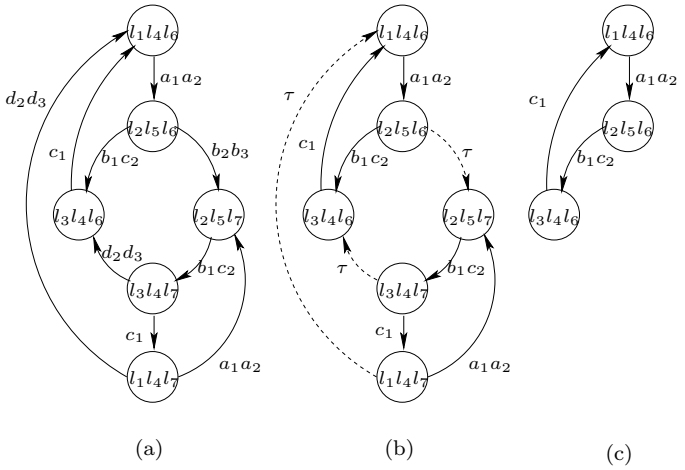

Figure 4.3 – Sets of traces represented as automata.

4.2.2 Data Non-Interference

Definition 16 (data non-interference) *The security assignment σ ensures data non-interference of $C = \gamma(B_1, \ldots, B_n)$ at security level s iff,*

$$\begin{array}{l} \forall q_1, q_2 \in Q_C^0 : \ q_1 \approx_s^\sigma q_2 \Rightarrow \\ \quad \forall w_1 \in \text{TRACES}(C, q_1), w_2 \in \text{TRACES}(C, q_2) : \ w_1|_s^\sigma = w_2|_s^\sigma \Rightarrow \\ \quad\quad \forall q_1', q_2' \in Q_C : \ q_1 \xrightarrow[C]{w_1} q_1' \wedge q_2 \xrightarrow[C]{w_2} q_2' \Rightarrow q_1' \approx_s^\sigma q_2' \end{array}$$

Data non-interference provides isolation/security at data level. The definition ensures that, all states reached from initially indistinguishable states at security level s, by execution of arbitrary but identical traces whenever projected at level s, are also indistinguishable at level s. That means that observation of all variables and interactions with level s or lower excludes any gain of relevant information about variables at higher (or incomparable) level than s. Compared to event non-interference, data non-interference is a stronger property that considers the system's global states (local states and valuation of variables) and focus on their equivalence along identical execution traces (at some security level).

Example 5 *Figure 4.4 presents an extension with data variables of the previous example from figure 4.2. We consider the following two traces $w_1 = \langle a_1a_2, b_2b_3, c_2b_1, d_2d_3, c_1, a_2a_1\rangle$ and $w_2 = \langle a_1a_2, b_2b_3, c_2b_1, c_1, a_2a_1\rangle$ that start from the initial state $((l_1, u = 0, v = 0), (l_4, x = 0, y = 0), (l_6, z = 0, w = 0))$. Although the projected traces at level L are equal, that is,*

$w_1|_L^\sigma = w_2|_L^\sigma = \langle a_1 a_2, c_2 b_1, c_1, a_1 a_2 \rangle$, *the reached states by* w_1 *and* w_2 *are different, respectively* $((l_2, u = 4, v = 2), (l_5, x = 3, y = 2), (l_6, z = 1, w = 1))$ *and* $((l_2, u = 4, v = 2), (l_5, x = 2, y = 2), (l_7, z = 1, w = 0))$ *and moreover non-equivalent at low level* L*. Hence, this example is not data non-interferent at level* L*.*

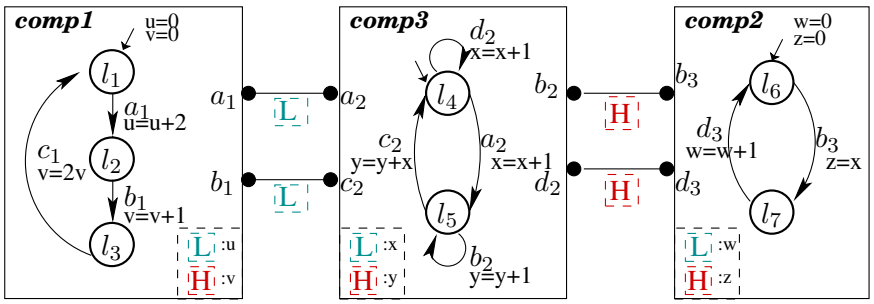

Figure 4.4 – Example for data non-interference.

Definition 17 (secure component) *A security assignment* σ *is secure for a component* $\gamma(B_1, \ldots, B_n)$ *iff it ensures both event and data non-interference, at all security levels* $s \in S$*.*

The verification of system design for information flow security, particularly non-interference, is hard, error prone and computationally difficult task. Indeed, to accurately verify non-interference definitions using universal quantifiers on variables and ports of the systems, we usually run and compare all possible execution sequences. Here, we show how can we obtain sufficient conditions for non-interference analysis in complex component-based systems by extending unwinding theorems for *secureBIP* .

4.3 Unwinding Relations

The unwinding conditions were first introduced by Goguen and Meseguer [**?**]. They provide sufficient conditions for non-interference that are efficient to verify since they rely basically in local conditions of pairs of states. The existence of unwinding relations is tightly related to non-interference. The basic idea is that if there exist an unwinding relation R of the states in an input/output state machine M for a policy dividing events in high security levels (H) and low (L), then non-interference holds on M for the (H)

and (L) partition. In general, the unwinding approach reduces the verification of information flow security to the existence of certain unwinding relation. This relation is usually an equivalence relation on system states that respects some additional properties on atomic execution steps, which are shown sufficient to imply non-interference. In the case of *secureBIP* , the additional properties are formulated in terms of individual interactions/events and therefore easier to handle. The following two definitions formalize this relation for the two types of non-interference defined.

Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let σ be a security assignment for C.

Definition 18 (unwinding relation) *An equivalence $\sim_s$ on states of C is called an unwinding relation for σ at security level s iff the two following conditions hold:*

1. *local consistency*
   $\forall q, q' \in Q_C : \ \forall a \in \gamma : \ q \xrightarrow[C]{a} q' \Rightarrow \sigma(a) \subseteq s \vee q \sim_s q'$
2. *output and step consistency*
   $\forall q_1, q_2, q_1' \in Q_C : \ \forall a \in \gamma :$
   $q_1 \sim_s q_2 \wedge q_1 \xrightarrow[C]{a} q_1' \wedge \sigma(a) \subseteq s \Rightarrow$
   $\exists q_2' \in Q_C : \ q_2 \xrightarrow[C]{a} q_2' \wedge$
   $\forall q_2' \in Q_C : \ q_2 \xrightarrow[C]{a} q_2' \Rightarrow q_1' \sim_s q_2'$

The unwinding conditions where applied first on labelled transition systems [**?**]. The main difference between classical labelled transition systems and the behavior of atomic components used in the *secureBIP* the use of variables and guards for keeping history of the state and for parametrizing ports. For data non-interference we have extended the unwinding theorem accordingly for coping soundly with these differences in the notation in an efficient way. Intuitively, by tagging the the set of variables at each atomic component, we statically analyze guarded transitions with actions by simultaneously extending an unwinding relation that considers the union of variable sets associated to each global state. Tagging keeps track of variables whose value is directly or indirectly dependent on higher security level ones, in the spirit of language based information flow analysis. This information allows to soundly decide on the output consistency of the relation.

In the case of *secureBIP* , the additional properties are formulated in terms of individual interactions/events and therefore easier to handle. The following two theorems formalize this relation for the two types of non-interference defined.

The next result allows to relate event non-interference and the existance of an unwinding relation.

Theorem 5 (event non-interference) *If an unwinding relation $\sim_s$ exists for the security assignment σ at security level s, then σ ensures event non-interference of C at level s.*

Proof 2 *We shall prove* $\text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)|_s^\sigma = \text{TRACES}((\gamma{\downarrow_s^\sigma})(B_1,\ldots,B_n),q_0)$ *by double inclusion. "$\supseteq$" inclusion: Independently of the unwinding relation, by using elementary set properties it holds that* $\text{TRACES}((\gamma{\downarrow_s^\sigma})(B_1,\ldots,B_n),q_0) = \text{TRACES}((\gamma{\downarrow_s^\sigma})(B_1,\ldots,B_n),q_0)|_s^\sigma \subseteq \text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)|_s^\sigma$. *"$\subseteq$" inclusion: This direction is an immediate consequence of Lemma 6 hereafter. It states that for every trace w in* $\text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)$ *its projection $w|_s^\sigma$ is also a valid trace in* $\text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)$. *But, this also means that $w|_s^\sigma$ is a valid trace in* $\text{TRACES}((\gamma{\downarrow_s^\sigma})(B_1,\ldots,B_n),q_0)$ *which proves the result.*

Lemma 6 *In the conditions of theorem 5, for every trace w in* $\text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)$, *for every state q such that $q_0 \xrightarrow[C]{w} q$, the projected trace $w|_s^\sigma$ is also a valid trace in* $\text{TRACES}(\gamma(B_1,\ldots,B_n),q_0)$ *and moreover, for every state q' such that $q_0 \xrightarrow[C]{w|_s^\sigma} q'$ it holds $q \sim_s q'$.*

Proof 3 *The lemma is proved by induction on the length of the trace w. For the empty trace $w = \epsilon$ verification is trivial: $\sim_s$ holds for the initial state $q_0 \sim_s q_0$ and $\epsilon = \epsilon|_s^\sigma$. By induction hypothesis, let assume the property holds for traces of length n. We shall prove the property for traces of length $n+1$. Let $w' = wa$ be an arbitrary trace of length $n+1$, let w be its prefix (trace) of length n and let a be the last interaction. Consider states q, q_1 such that $q_0 \xrightarrow[C]{w} q \xrightarrow[C]{a} q_1$. By the induction hypothesis we know that $w|_s^\sigma$ is a valid trace and for all states q' such that $q_0 \xrightarrow[C]{w|_s^\sigma} q'$ it holds $q \sim_s q'$. We distinguish two cases, depending on the security level of a:*

- *$\sigma(a) \not\subseteq s$: In this case, $w'|_s^\sigma = w|_s^\sigma$ hence, $w'|_s^\sigma$ is a valid trace as well, reaching the same states q'. Moreover, since a is invisible for s, the unwinding condition (1) ensures that $q \sim_s q_1$. By transitivity, this implies that $q_1 \sim_s q'$, which proves the result.*
- *$\sigma(a) \subseteq s$: In this case, $w'|_s^\sigma = w|_s^\sigma a$. From the unwinding condition (2), since $q \sim_s q'$ and a is visible and enabled in q then, a must also be enabled in q'. Therefore, $w|_s^\sigma$ can be extended with a from state q' to some q_1' hence, $w'|_s^\sigma$ is indeed a valid trace. Moreover, since $q \sim_s q'$ the unwinding condition (2) ensures also that $q_1 \sim_s q_1'$, which proves the result.*

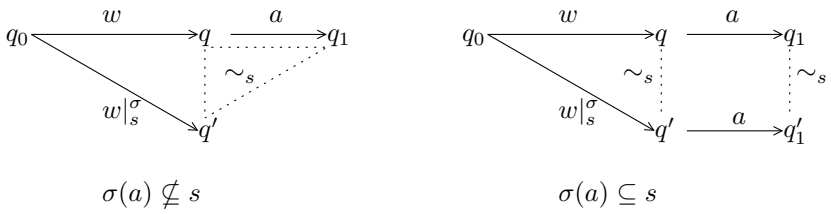

Figure 4.5 – Proof illustration for lemma 6

Based on the unwinding technique, we formally give a verification of data non-interference.

Theorem 7 (data non-interference) *If the equivalence relation $\approx_s^\sigma$ is also an unwinding relation for the security assignment σ at security level s, then σ ensures data non-interference of C at level s.*

Proof 4 *Let us consider two equivalent states $q_1 \approx_s^\sigma q_2$. The* first condition *for data non-interference requires that, for any trace w_1 from q_1 there exists a trace w_2 from q_2 having the same projection at level s, that is, $w_1|_s^\sigma = w_2|_s^\sigma$.*

We shall prove a slightly stronger property, namely, the trace w_2 can be chosen such that, the successors q_1' and q_2' of respectively q_1 by w_1 and q_2 by w_2 are moreover equivalent, that is, $q_1' \approx_s^\sigma q_2'$. The proof is by induction on the length of the trace w_1. The base case*: for the empty trace $w_1 = \epsilon$ we take equally $w_2 = \epsilon$ we immediately have $q_1' = q_1 \approx_s^\sigma q_2 = q_2'$.* The induction step: *we assume, by induction hypothesis that the property holds for all traces w_1 such that $|w_1| \leq n$ and we shall prove it for all traces w_1' such that $|w_1'| = n + 1$. Let a be the last interaction executed in w_1', that is, $w_1' = w_1 a$ with $|w_1| = n$. Let q_1'' be the state reached from q_1 by w_1. From the induction hypothesis, there exists a trace w_2 that leads q_2 into q_2'' such that $w_1|_s^\sigma = w_2|_s^\sigma$ and moreover $q_1'' \approx_s^\sigma q_2''$. We distinguish two cases, depending on the security level of a:*

- *$\sigma(a) \not\subseteq s$: since $\approx_s^\sigma$ is unwinding and $q_1'' \xrightarrow[C]{a} q_1'$ it follows that $q_1'' \approx_s^\sigma q_1'$. In this case, we take $w_2' = w_2$ and $q_2' = q_2''$ which ensures both $w_1'|_s^\sigma = w_1|_s^\sigma = w_2|_s^\sigma = w_2'|_s^\sigma$ and $q_1' \approx_s^\sigma q_1'' \approx q_2'' = q_2'$.*
- *$\sigma(a) \subseteq s$: since $\approx_s^\sigma$ is unwinding and $q_1'' \approx_s^\sigma q_2''$ and $q_1'' \xrightarrow[C]{a} q_1'$ there must exists q_2' such that $q_2'' \xrightarrow[C]{a} q_2'$ and moreover, for any such choice $q_1' \approx_s^\sigma q_2'$. Hence, in this case, the trace $w_2' = w_2 a$ executed from q_2 and leading to q_2' satisfies our property, namely $w_1'|_s^\sigma = w_1|_s^\sigma a = w_2|_s^\sigma a = w_2'|_s^\sigma$ and $q_1' \approx_s^\sigma q_2'$.*

The second condition *for data non-interference requires that, for any traces* w_1 *and* w_2 *with equal projection on security level* s*, that is* $w_1|_s^\sigma = w_2|_s^\sigma$*, any successor states* q_1' *and* q_2' *of respectively* q_1 *by* w_1 *and* q_2 *by* w_2 *are also equivalent at level* s*. This property is proved also by induction on* $|w_1|+|w_2|$*, that is, on the sum of the lengths of traces* w_1*,* w_2*.* The base case*: for empty traces* $w_1 = w_2 = \epsilon$ *we have that* $q_1' = q_1$ *and* $q_2' = q_2$ *and hence trivially* $q_1' \approx_s^\sigma q_2'$*.* The induction step*: we assume, by induction hypothesis that the property holds for any traces* w_1*,* w_2 *such that* $|w_1| + |w_2| \leq n$ *and we shall prove it for all traces* w_1'*,* w_2' *such that* $|w_1'| + |w_2'| = n + 1$*. We distinguish two cases, depending on the security levels of the last interactions occurring in* w_1' *and* w_2'*.*

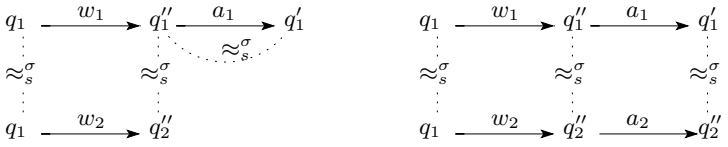

Figure 4.6 – Proof illustration for theorem 7

— *at least one of the last interactions in* w_1' *or* w_2' *has a security level not lower or equal to* s*. W.l.o.g, consider that indeed* $w_1' = w_1 a_1$ *and* $\sigma(a_1) \not\subseteq s$*. This situation is depicted in figure 4.6, (left).*
Let q_1'' *be the state reached from* q_1 *after* w_1*. Since* $w_1'|_s^\sigma = w_2'|_s^\sigma$ *and* $\sigma(a_1) \not\subseteq s$ *it follows that* $w_1|_s^\sigma = w_1'|_s^\sigma = w_2'|_s^\sigma$*. The induction hypothesis holds then for* w_1 *and* w_2' *because* $|w_1| + |w_2'| = n - 1$ *and hence we have that* $q_1'' \approx_s^\sigma q_2'$*. Moreover,* q_1' *is a successor of* q_1'' *by interaction* a_1*. Since the security level of* a_1 *is not lower or equal to* s*, and* $\approx_s^\sigma$ *is an unwinding relation at level* s*, it follows from the local consistency condition that* $q_1'' \approx_s^\sigma q_1$*. Then, by transitivity of* $\approx_s^\sigma$ *we obtain that* $q_1' \approx_s^\sigma q_2'$*.*

— *the last interactions of both traces* w_1' *and* w_2' *have security level lower or equal to* s*. That is, consider* $w_1' = w_1 a_1$ *and* $w_2' = w_2 a_2$ *with* $\sigma(a_1) \subseteq s$*,* $\sigma(a_2) \subseteq s$*. This situation is depicted in figure 4.6, (right).*
Let q_1'' *and* q_2'' *be the states reached respectively from* q_1 *by* w_1 *and from* q_2 *by* w_2*. Since* $\sigma(a_1) \subseteq s$*,*$\sigma(a_2) \subseteq s$ *we have* $w_1'|_s^\sigma = w_1|_s^\sigma a_1$*,* $w_2'|_s^\sigma = w_2|_s^\sigma a_2$*. From the hypothesis,* $w_1'|_s^\sigma = w_2'|_s^\sigma$*, it follows that both* $a_1 = a_2$ *and* $w_1|_s^\sigma = w_2|_s^\sigma$*. Therefore, the induction hypothesis can be applied for traces* w_1, w_2 *because* $|w_1|+|w_2| = n-2$ *and hence, we obtain* $q_1'' \approx_s^\sigma q_2''$*. But now,* q_1' *and* q_2' *are immediate successors of two equivalent states* q_1'' *and* q_2'' *by executing some interaction*

$a = a_1 = a_2$, having security level lower or equal to s. Since, $\approx_s^\sigma$ is an unwinding relation at level s, it follows from the step consistency condition that successors states q_1' and q_2' are also equivalent at level s, hence, $q_1' \approx_s^\sigma q_2'$.

4.4 Static Security conditions

The two theorems above are used to derive a practical verification method of non-interference using unwinding. We provide hereafter sufficient syntactic conditions ensuring that indeed the unwinding relations $\sim_s$ and $\approx_s$ exist on the system states. These conditions aim to effectively reduce the verification of non-interference to the checking on local constraints on both transitions (intra-component conditions) and interactions (inter-component conditions). Especially, they give an direct way to automate the verification.

Definition 19 (security conditions) *Let $C = \gamma(B_1, \ldots, B_n)$ be a composite component and let σ be a security assignment. We say that C satisfies the security conditions for security assignment σ iff:*

(i) the security assignment of ports, in every atomic component B_i is locally consistent, that is:

— for every pair of causal transitions:

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \ \tau_2 = \ell_2 \xrightarrow{p_2} \ell_3 \ \Rightarrow \\ \ell_1 \neq \ell_2 \ \Rightarrow \ \sigma(p_1) \subseteq \sigma(p_2)$$

— for every pair of conflicting transitions:

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p_1} \ell_2, \ \tau_2 = \ell_1 \xrightarrow{p_2} \ell_3 \ \Rightarrow \\ \ell_1 \neq \ell_2 \ \Rightarrow \ \sigma(p_1) \subseteq \sigma(p_2)$$

(ii) all assignments $x := e$ occurring in transitions within atomic components and interactions are sequential consistent, in the classical sense:

$$\forall y \in use(e) : \ \sigma(y) \subseteq \sigma(x)$$

(iii) variables are consistently used and assigned in transitions and interactions, that is,

$$\forall \tau \in \cup_{i=1}^n T_i \ \ \forall x, y \in X \ : x \in def(f_\tau), y \in use(g_\tau) \ \Rightarrow \\ \sigma(y) \subseteq \sigma(p_\tau) \subseteq \sigma(x)$$
$$\forall a \in \gamma \ \ \forall x, y \in X \ : x \in def(F_a), y \in use(G_a) \ \Rightarrow \\ \sigma(y) \subseteq \sigma(a) \subseteq \sigma(x)$$

(iv) all atomic components B_i are port deterministic:

$$\forall \tau_1, \tau_2 \in T_i : \ \tau_1 = \ell_1 \xrightarrow{p} \ell_2, \tau_2 = \ell_1 \xrightarrow{p} \ell_3 \quad \Rightarrow \\ (g_{\tau_1} \wedge g_{\tau_2}) \textit{ is unsatisfiable}$$

The first family of conditions (i) is similar to Accorsi's conditions [**?**] for excluding causal and conflicting places for Petri net transitions having different security levels. Similar conditions have been considered in [**?**, **?**] and lead to more specific definitions of non-interferences and bisimulations on annotated Petri nets. The second condition (*ii*) represents the classical condition needed to avoid information leakage in sequential assignments. The third condition (*iii*) tackles covert channels issues. Indeed, (*iii*) enforces the security levels of the data flows which have to be consistent with security levels of the ports or interactions (e.g., no low level data has to be updated on a high level port or interaction). Such that, observations of public data would not reveal any secret information. Finally, conditions (*iv*) enforces deterministic behavior on atomic components.

The relation between the syntactic security conditions and the unwinding relations is precisely captured by the following theorem.

Theorem 8 (unwinding theorem) *Whenever the security conditions hold, the equivalence relation $\approx_s^\sigma$ is an unwinding relation for the security assignment σ, at all security level s.*

The following result is the immediate consequence of theorems 5, 7 and 8.

Corollary 2 *Whenever the security conditions hold, the security assignment σ is secure for the component C.*

4.5 Configuration Synthesis

When modeling sophisticated systems using *secureBIP* model, it surely requires advanced security skills and tools to ensure critical information security. However, tracking illicit information flows in huge models with complex interactions can be challenging. Hence, we consider that simplifying and automating the verification process is very important. In this chapter we consider a practical method to synthesize and verify the information flow security in the *secureBIP* model, where starting from intuitively given security annotations for some variables in the system model we automatically generate, a full annotation configuration for variables and ports if

the system is secure. Otherwise, the security issue is located and an error referring to security levels inconsistency regarding the initial security annotations is generated. The proposed synthesis approach is based on the use of previously defined security conditions ensuring by this a formally proved end-to-end information flow security. For clarity sake some restriction to the previous model are applied while presenting the approach. In this section we present some restrictions on the model applied to the *secureBIP* model for which we provide a compositional approach to synthesize security configurations in *secureBIP* models.

4.5.1 Model Restriction

The restrictions that we impose on the (centralized) *secureBIP* model aim at simplifying model representation towards tracking information flows. More precisely, the simplified model is a refined model that allows defining information flow directions, which allows automatically propagate annotations in the system and synthesize them. The restrictions are applied to atomic components of the *secureBIP* allowing to have a centralized send/receive model with strong synchronized interactions, where:

— we distinguish respectively input ports $P^{in} \subseteq P$ and output ports $P^{out} \subseteq P$ in the *secureBIP* model and we assume that they are disjoint, $P^{in} \cap P^{out} = \emptyset$. Here the data is exchanged between atomic components through ports.

— every input or output port $p \in P^{in} \cup P^{out}$ is associated to a unique variable $var(p) \in X$ such that no sharing exists, that is, whenever $var(p) = var(p')$ we have $p = p'$.

— the system evolves either by performing asynchronously an internal step of some component B_i (INTER rule) or by performing a synchronous communication between two components B_i, B_j involving respectively ports p_i^{out}, p_j^{in} related by a connector in Γ (COMM rule).

Figure 4.7 provides an example of atomic components. The *Producer* component contains two states l_1 and l_2 and one output ports *out*. The transition labelled with port *produce* takes place only if the guard $[w \geq 3]$ is true. Then, the variable x is incremented by executing the assignment $x := x + 1$. The transition labelled with the *out* port takes place

In this section, we first recall the main aspects of information flow analysis that we require in controlling the information flow in the system model. Then, we apply these notions on *secureBIP* model where we extract implicit and explicit information flow dependencies and we give an algorithm to automatically synthesize and verify security annotations. Here we give an optimized, rapid and compositional solution to secure the system.

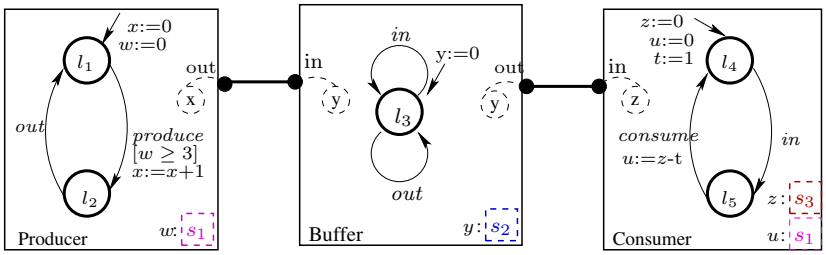

Figure 4.7 – A *Producer-Buffer-Consumer* example

4.5.2 Data-flow analysis

Introduced in 1987, data-flow graphs *DFG*, represent a combination of both control and data dependence allowing to analyze tasks such as program slicing [**?**, **?**], in parallelization and vectorization optimizations [**?**, **?**], and as a software engineering and testing tool [**?**, **?**]. In this chapter, we use data-flow graphs to extract and construct information dependence in the model composition in an automated way. We define, hereafter, the control and data dependence that we require to analyze and control information flow in sequential programs.

```
a1:   if (x>5) then
a2:       y:=0
a3:   z:=1
```

(a)

```
s1:   a:=5
s2:   b:=a
s3:   c:=b
```

(b)

Figure 4.8 – Example: control and data dependency

— control dependence (implicit flow): considering the set of instructions presented in example 4.8(a), a1 has a control dependence on a predecessor instruction a2 where the outcomes of a2 determines whether or not the instruction a1 should be executed or not. Hence, variable y has a control dependence on variable x.
— data dependence (explicit flow): data or flow dependency occurs when an instruction depends on the result of a previous instruction. The simple way of defining data dependency is the use of assignments instructions. As presented in example 4.8(b), instruction s3 (resp. s2) is dependent on instruction s2 (resp. s1). The variable c depends on variable b and variable b depends on a, hence the variable

c is also dependent on b and a.

Roughly speaking, the information flow analysis in *secureBIP* model is based on the construction of a *DFG* that considers the set of variables and ports in the model as nodes on which we calculates there dependencies at each execution step of the model. Then, we apply an iterative algorithm that verifies a transfers function, expressed using the *can-flow to* information flow relation, till reaching a fix point. Here we recall the basic steps to the iterative algorithm:

Algorithm 1: Data-flow Algorithm

```
1 Set enter.after = init. Set all other n.after to T.  ▷ initialize n.afters:
2 Initialize a worklist to contain DFG nodes
3 while the worklist is not empty do
4     Remove a node n from the worklist.
5     Compute n.before by combining all p.after such that p is a
      predecessor of n in the CFG.
6     Compute tmp = fn ( n.before )
7     if tmp != n.after then
8         Set n.after = tmp
9         Put all of n's successors on the worklist
```

In the following we will extend the data-flow analysis algorithm, Algorithm 1, to handle and control information flow in the system composition.

4.5.3 Security Synthesis

The configuration synthesis problem is defined as follows. Given a partial security annotation of a system, extend it towards a complete annotation which is provable secure according to Corollary 2, or show that no such annotation actually exists. We assume that system components are port deterministic.

We rely on flow dependency graphs as an intermediate artifact for solving this problem. For every component $B_i = (Q_i, X_i, P_i, T_i)$, we define the flow dependency graph $\mathcal{G}_i = (N_i, \hookrightarrow_i)$ where the set of vertices $N_i = X_i \cup P_i$ contains the ports and variables of B_i and edges $\hookrightarrow_i \subseteq N_i \times N_i$ correspond to flow dependencies required by Corollary 2 and are defined below, for

every $x, y \in X_i$, $p, r \in P_i$:

$y \hookrightarrow_i x$	*iff*	$\exists t \in T_i.\ x := e \in asgn(t),\ y \in use(e) \cup use(guard(t))$
$p \hookrightarrow_i x$	*iff*	$\exists t \in T_i.\ x := e \in asgn(t),\ p = port(t) \bigvee p \in P_i^{in}, x = var(p)$
$y \hookrightarrow_i p$	*iff*	$\exists t \in T_i.\ y \in use(guard(t)), p = port(t) \bigvee p \in P_i^{out}, y = var(p)$
$p \hookrightarrow_i r$	*iff*	$\exists t, t' \in T_i.\ p = port(t), r = port(t'), (dst(t) = src(t') \vee src(t) = src(t'))$

Using flow dependency graphs, the configuration synthesis problem is formally rephrased as follows:

— Given system $\Gamma(B_1, ..., B_n)$, partial annotation $\sigma_0 : X \to \mathbb{S} \cup \{\bot\}$

— Find complete annotation $\zeta : X \cup P \to \mathbb{S}$ such that

(C1) (initial annotation) $\forall x \in X.\ \sigma_0(x) \neq \bot \implies \zeta(x) = \sigma_0(x)$

(C2) (flow preservation) $\forall i = 1, n.\ \forall x, y \in P_i \cup X_i.\ x \hookrightarrow_i y \implies \zeta(x) \leq \zeta(y)$

(C3) (connector consistency) $\forall \gamma = (p^{out} p^{in}) \in \Gamma.\ \zeta(p^{out}) = \zeta(p^{in})$

If a complete annotation ζ exists and satisfies the conditions (C1-C3) above, then the system $\Gamma(B_1, ..., B_n)$ is provable secure for $\sigma = \zeta_{|X}$ and $\varsigma = \zeta_{|P}$, which are respectively the projections of ζ to variables X and ports P. That is, all conditions required by Corollary 2 on annotation of ports and variables within components are captured by dependency graphs $(\mathcal{G}_i)_{i=1,n}$ and satisfied according to (C2). Connectors are consistently annotated according to (C3). Moreover, the initial annotation is preserved by (C1).

An iterative algorithm to compute the complete annotation ζ is depicted as Algorithm 2 below. If the algorithm terminates without detecting inconsistencies, then ζ is the less restrictive annotation satisfying conditions (C1-C3). If an inconsistency is detected, then no solution exists. In this case, the initial annotation is inconsistent with respect to the information flow within the system.

Initially, all system variables are either annotated by security levels given from system designer σ_0 if it exist or a default level that correspond to the greatest lower bound security levels ($\sqcap\mathbb{S}$) in the lattice (line 1). The algorithm visits iteratively all components (lines 2-18). For every component B_i, it propagates forward the current annotation ζ within the flow graph $\mathcal{G}_i$ (lines 3-13). The security level $\zeta(n_i)$ of every node n_i is eventually increased to become more restrictive than the levels of its predecessors (lines 8-13). An inconsistency is reported if the security level increases for an initially annotated variable (lines 10-11). Any change triggers recomputation of successors nodes of n_i (lines 12-13). Finally, once the annotation within $\mathcal{G}_i$ is computed, any change on security levels on input/output ports is propagated to connected ports (lines 14-18). The involved components

Algorithm 2: Annotation Synthesis

1 $\zeta(n) \leftarrow \begin{cases} \sigma_0(n) & \text{if } n \in X, \sigma_0(n) \neq \bot \\ \sqcap \mathbb{S} & \text{otherwise} \end{cases}$ ▷ initialization

2 $BList \leftarrow \{B_i\}_{i=1,n}$ ▷ inter-component outer loop

3 **while** $BList \neq \emptyset$ **do**

4 *choose-and-remove*($BList$, B_i)

5 $nList \leftarrow X_i \cup P_i$ ▷ intra component inner loop for $\mathcal{G}_i$

6 **while** $nList \neq \emptyset$ **do**

7 *choose-and-remove*($nList$, n_i)

8 $s_i \leftarrow \sqcup\{\zeta(n) \mid n \hookrightarrow_i n_i\}$ ▷ recompute security level of n_i

9 **if** $\zeta(n_i) \leq s_i$ *and* $s_i \neq \zeta(n_i)$ **then**

10 **if** $n_i \in X_i$ *and* $\sigma_0(n_i) \neq \bot$ **then**

11 **stop** ▷ inconsistency detected

12 $\zeta(n_i) \leftarrow s_i$ ▷ update and propagate change within $\mathcal{G}_i$

13 $nList \leftarrow nList \cup \{n \mid n_i \hookrightarrow_i n\}$

14 **foreach** $p_i \in P_i^{out} \cup P_i^{in}$ **do**

15 *find* $p_j \in P_j^{out} \cup P_j^{in}$ *with* $(p_i p_j) \in \Gamma$ *or* $(p_j p_i) \in \Gamma$

16 **if** $\zeta(p_i) \neq \zeta(p_j)$ **then**

17 $\zeta(p_j) \leftarrow \zeta(p_i)$ ▷ update and propagate change across connectors

18 $BList \leftarrow BList \cup \{B_j\}$

need to be revisited again (line 18). Notice that both while loops are guaranteed to terminate as the number of annotation changes is bounded for every node. That is, the security level can only be increased finitely many times in a bounded lattice $(\mathbb{S}, \leq)$.

Proposition 1 *Algorithm 2 solves the configuration synthesis problem.*

Proof 5 *Initially, the annotation ζ is defined to satisfy initial annotation condition (C1). The algorithm propagates this annotation along the flow graphs, without changing the initially annotated variables. It terminates only when ζ satisfies flow preservation condition (C2) and connector consistency (C3).*

As an example, we apply Algorithm 2 to the Producer-Buffer-Consumer presented in Figure 4.7 with initial annotation $\{x \mapsto s_2, y \mapsto s_2, z \mapsto s_3, t \mapsto s_1\}$, for security levels s_1, s_2 and s_3, such that $s_1 \leq s_2 \leq s_3$. The three flow dependency graphs and their dependencies through connectors are depicted in Figure 4.9. For this initial labelling, the algorithm succeeds to generate a complete annotation $\{x \mapsto s_2, w \mapsto s_2, out \mapsto s_2, produce \mapsto s_2, y \mapsto s_2, in \mapsto s_2, out \mapsto s_2, z \mapsto s_3, t \mapsto s_1, u \mapsto s_3, in \mapsto s_2\}$. If however we add to the initial configuration a label to the guard variable w, $\{w \mapsto s_3\}$, Algorithm 2 detects an inconsistency at the *Producer* component and an illicit flow from the w variable to y variable through port *produce* is reported to the user. To illustrate the efficiency of information flow

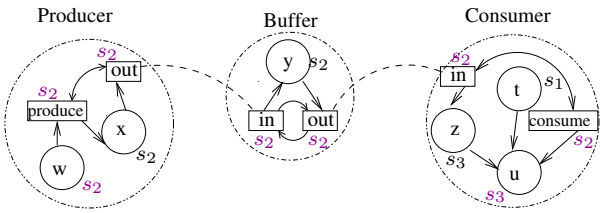

Figure 4.9 – Dependency graphs of Producer-Buffer-Consumer from Figure 4.7

verification using *secureBIP* framework, we consider two applications: a social network application whens-App and the Travel Reservation system.

4.6 Use-Case 1: Whens-App Application

We consider *Whens-App*, an Online Social Network (OSN) application for organizing events, such as business meeting where participants can exchange data when these meetings take place. Figure 4.10 shows an overview

of a fragment of the system which consists of a finite number of users communicating using the *Whens-App* application. Each one of the users is capable of creating or receiving an event. An event creation has to include at least two users, an *Event-Creator* and an *Event-Receiver*. The behavior of *Event-Creator* and *Event-Receiver* components are given in Figure 2.3. Communication channels are represented by lines in the figure.

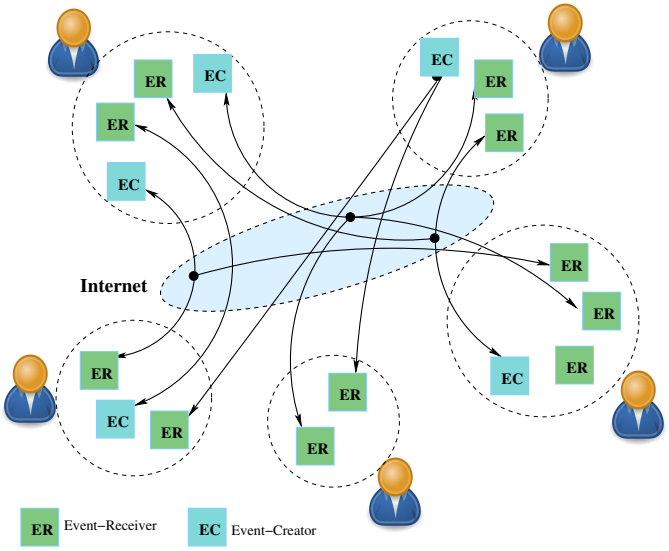

Figure 4.10 – High-level description of the Whens-App system.

As social network application, *Whens-App*, entails a large variety of security requirements, here however, we focus on some relevant requirements related to information flow security: Assuming that components are trustful and the network is insecure, *(1)* the interception and observation of exchanged data messages does not reveal any information about event participants and *(2)* confidentiality of classified data as well as events is always preserved and kept secret inter- as well as intra-components. Both requirements are ensured by using security annotations model for tracking events and data in the system and checking that the formal model satisfies the security constraints given in Section 4.4.

Several possibilities of security assignment can be applied to the system model. Here we give in the following examples two of these possibilities: the first one, presented in figure 4.11, correspond to securing data transfer between different participants of the event where we consider that all communications that contain a data transfer are secure, while the second one, presented in 4.12, shows that only data outgoing from the *Event-Creator*

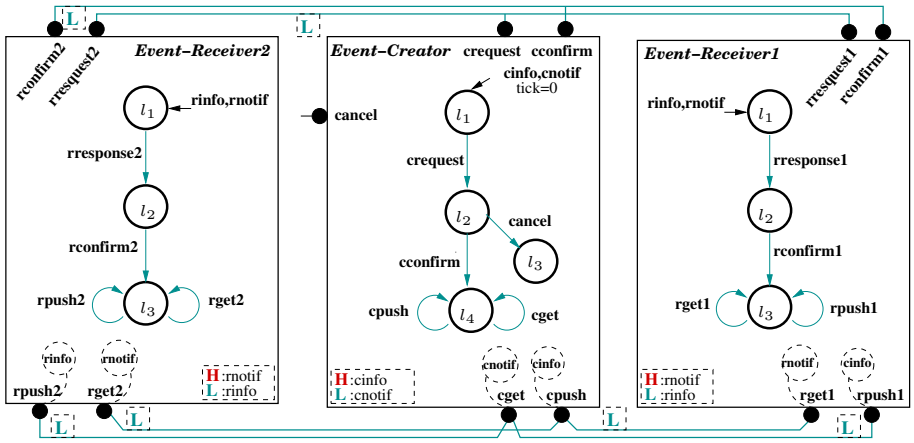

Figure 4.11 – Non-interferent Event-Creation from the Whens-App application

is considered secure while data received by the *Event-Creator* from different *Event-Receivers* can be considered public. with these examples, the security conditions are respected at port annotations as well as variables assignments. Where the both variables at all component have the same security level (H) and transferred at at an interaction of ports labelled with a (H) label.

The third one, presented in figure 4.13 where the system is considered interferent. Here, we have a security level inconsistency at the conflicting interation *cancel*, $confirm$. The system is considered insecure at the two states l_2, since the execution of one of the interactions would reveal the non-execution of the other. Hence, the observation of the execution of the *cancel* interaction reveals information about the event creation which have been canceled. Also, another information leak is detected at data level, where the assignment of variables at the *push* interactions have an inconsistent security level. Indeed, the variable $rinfo1$ and $rinfo2$, having both (H) label, are assigned to the variable $cinfo$ that is annotated with an (L) label.

4.7 Use-Case 2: Travel Reservation

We illustrate the *secureBIP* framework to handle information flow security issues for a classical example, the web service reservation system proposed in [**?**]. A businessman, living in France, plans to go to Berlin for a private and secret mission. To organize his travel, he uses an intelligent

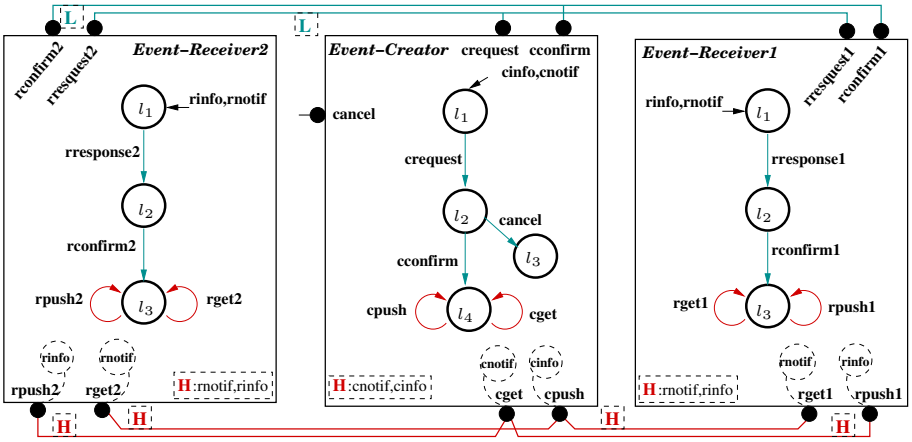

Figure 4.12 – Non-interferent Event-Creation from the Whens-App application

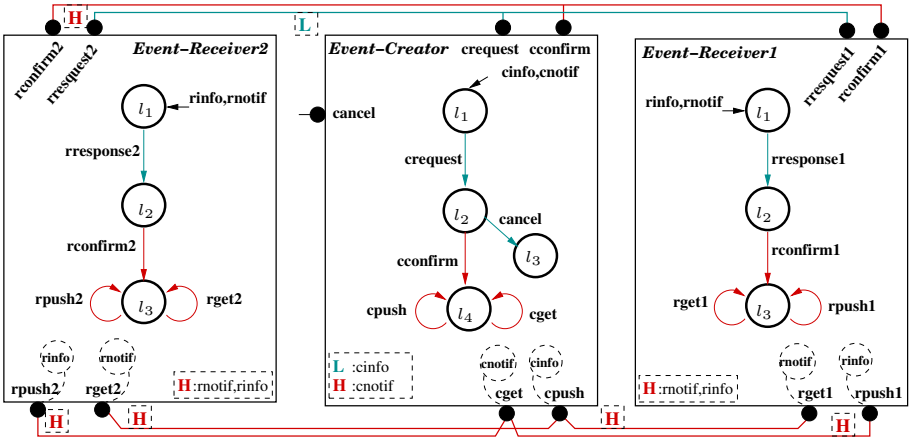

Figure 4.13 – Interferent Event-Creation from the Whens-App application

web service who contacts two travel agencies: The first agency, *AgencyA*, arranges flights in Europe and the second agency, *AgencyB*, arranges flights exclusively to Germany. The reservation service obtains in return specific flight information and their corresponding prices and chooses the flight that is more convenient for him.

In this example, there are two types of interference that can occur, (1) data-interference since learning the flight price may reveal the flight destination and (2) event interference, since observing the interaction with *AgencyB* can reveal the destination as well. Thus, to keep the mission private, the flight prices and interactions with *AgencyB* have to be kept confidential.

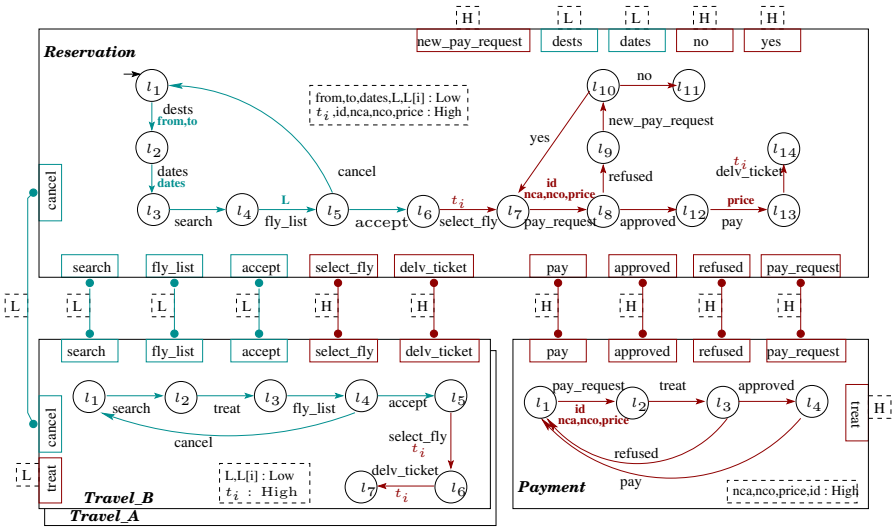

Figure 4.14 – Model of reservation web service in *secureBIP*

The modeling of the system using *secureBIP* involves two main distinct steps: first, functional requirements modeling reflecting the system behavior, and second, security annotations enforcing the desired security policy. The model of the system has four components denoted: *Travel_A* and *Travel_B* who are instances from the same component and correspond respectively to *AgencyA* and *AgencyB*, and components *Reservation* and *Payment*. To avoid Figure 4.14 cluttering, we did not represent the interactions with *Travel_A* component. Search parameters are supplied by a user through the *Reservation* component ports *dests* and *dates* to which we associate respectively variables $(from, to)$ and *dates*. Next, through search interaction, *Reservation* component contacts *Travel_B* component to search for available flights and obtains in return a list L of specific flights

with their corresponding prices. Thereafter, *Reservation* component selects a ticket t_i from the list L and requests the *Payment* component to perform the payment.

All the search parameters $from$, to, $dates$, as well as the flights list L are set to low since users are not identified while sending these queries. Other sensitive data like the selected flight t_i, the price variable p and the payment parameters (identity *id*, credit card variable *cna* and code number *cno*) are set to high. Internal ports *dests* and *dates* as well as *search*, *fly_list*, *accept* interactions are set to low since these interactions (events) do not reveal any information about the client private trip. However, the $select\_fly$ interaction must be set to high since the observation of the selection event from *AgencyB* allow to deduce deduce the client destination. In the case of a selected flight from *AgencyA*, the $select\_fly$ interaction could be set to low since, in this case, the destination could not be deduced just from the event occurrence.

We recall that any system can be proven non-interferent iff it satisfies the syntactic security conditions from Definition 19. Indeed, these conditions hold for the system model depicted in Figure 4.14. In particular, it can be easily checked that all assignments occurring in transitions within atomic component as well as within interactions are sequential consistent. For example, at the $select\_fly$ interaction we assign a low level security item from the flight list L to a high security level variable ti, formally $t_i = L[i]$. Besides, the security levels assignments to ports exclude inconsistencies due to causal and conflicting transitions, in all atomic components.

4.8 Conclusion

In this chapter we presented *secureBIP*, a framework to secure component-based systems. Here we formally define a security model by specifying two types of non-interference, respectively event and data non-interference in a single semantic model. To the best of our knowledge, these properties have never been jointly considered for component-based models. Nevertheless, the need to consider together event and data flow non-interference has been recently identified in the existing literature. The bottom line is that preserving the safety of data flow in a system does not necessarily preserve safe observability on system's public behavior (i.e., secret/private executions may have an observable impact on system public events).

We extended the *BIP* framework to encompass security annotations to track sensitive information throughout the system. Based on these annotations, we provided a set of sufficient syntactic conditions to practically

and automatically verify the non-interference. These conditions are extensions of security typed language rules applied to our model. Compared to security-typed programming languages [**?**, **?**] and operating systems [**?**, **?**, **?**] enforcing information flow control, *secureBIP* is a component-based modeling approach where non-interference is established at a more abstract level. Thus, *secureBIP* can be implemented using different programming languages and is independent from a specific operating system and execution platform.

The use of our framework has been demonstrated on a Web Service application and an social online network application. Distinct security annotations can be applied to these systems according to the perspective of the system designer and his security intuitions. The verification model generates a verdict about the security. In the next chapter we introduce a automated and practical method to decentralize the system while preserving security.

Chapter 5

Distributed Secure Component-Based Model

Contents

The decentralized *BIP* model, previously presented in the Chapter 2, is a set of components executing in a group of distributed nodes such that there is at least one component per node. An executing component in the decentralized model is able to send and receive data between other component. Assuming that the decentralized *BIP* components are running on a trusted hosts, it still paramount to the non-leakage of sensitive information intra-component as well as inter-component while exchanging data at interaction level. In Chapter 2, we verified the non-interference property in a multi-party interaction model. Following the transformation steps that aim to decentralize the model, new variables, ports and transitions are added and interactions are splitted. It would be interesting to find an automated method to generate from a centralized secure model a secure distributed model implementable on a distributed platform.

In this chapter, we extend the decentralized *BIP* framework by modifing the previously presented transformation such that to encompass and preserve information flow security from centralized to distributed model. Following a set of typing rules, we show that the security annotations of a high-level model can be transferred to the three-layer distributed model while preserving non-interference. The new proposed transformation imposes additional restrictions on the partitioning of interactions as well as on the structure of the conflict resolution layers. That is, interactions and conflict resolution must be statically partitioned according to their security levels to avoid information leaks. Moreover, we give a set of rules to propagate annotations in way to build a secure-by-construction model.

In this chapter, we describe the transformation steps with respect to security aspect for the decentralized 3-layer architecture model, where Section 5.1.1 presents distributed *Atomic Component Layer*, Section 5.1.2 presents the *Interaction-Protocol Layer* and Section 5.1.3 presents the *Conflict-Resolution Layer*.

5.1 Architecture Secure Decentralized Model

5.1.1 Distributed Atomic Layer

The transformation of an atomic component B_i in *secureBIP* model into a Send/Receive atomic component B^{SR} is based on decomposing each atomic synchronization into a send and a receive action. Precisely, each transition is split into two consecutive steps: (1) an offer that publishes the current state of the component, and (2) a notification that triggers the update function. As previously explained in Chapter 4, ports and variables

in atomic component B_i in a centralized model are classified with security levels.

Let $C = \gamma(B_1, \cdots B_n)$ be a composite component and σ be a security assignment for C with domain S, fixed. Moreover, assume that σ satisfies the security conditions for C, defined in Section 4.4 frm Chapter 4 . Furthermore, to simplify presentation, consider that atomic components are deterministic, that is, for every state ℓ and for every port p there exists at most one transition outgoing ℓ which is labelled by p.

Definition 20 (Transformed atomic component) *Let $B = (L, X, P, T)$ be an atomic component within C. The corresponding transformed S/R component is $B^{SR} = (L^{SR}, X^{SR}, P^{SR}, T^{SR})$:*

- *$L^{SR} = L \cup L^{\perp}$, where $L^{\perp} = \{\perp_\ell \mid \ell \in L\}$*
- *$X^{SR} = X \cup \{x_p\}_{p \in P} \cup \{n_s | s \in S\}$ where each x_p is a Boolean variable indicating whether port p is enabled, and n_s is an integer called a participation number (for security level s).*
- *$P^{SR} = P \cup \{o_s \mid s \in S\}$. The offer ports o_s export the variables $X^{SR}_{o_s} = \{n_s\} \bigcup \{\{x_p\} \cup X_p \mid \sigma(p) = s\}$ that is the participation number n_s, the new Boolean variables x_p and the variables X_p associated to ports p having security level s. For all other ports $p \in P$, we define $X^{SR}_p = X_p$.*
- *For each state $\ell \in L^{SR}$, let S_ℓ be the set of security levels assigned to ports labelling all outgoing transitions of ℓ. For each security level $s \in S_\ell$, we include the following transition $\tau_{o_s} = (\perp_\ell \xrightarrow{o_s} \ell) \in T^{SR}$, where the guard g_{o_s} is true and f_{o_s} is the identity function.*
- *For each transition $\tau = \ell \xrightarrow{p} \ell' \in T$ we include a notification transition $\tau_p = (\ell \xrightarrow{p} \perp_{\ell'})$ where the guard g_p is true. The function f_p applies the original update function f_τ on X, increments n_s and updates the Boolean variables x_r, for all $r \in P$. That is, $x_r := g_\tau$ if $\exists \tau = \ell' \xrightarrow{r} \ell'' \in T$, and $false$ otherwise.*

To preserve security classification and enforce its consistency in the decentralized model we consider different levels of offer port named o_s in B^{SR}_i for each defined security level in the B_i component. Furthermore, relevant variables security level are annotated in a way to preserve initially defined security partitioning. Indeed, the set of initially defined variables do preserve their annotations whereas relevant variables, such as port enabling denoted x_p and interaction execution counters denoted n_s, are annotated respectively to there interactions levels. The exported variables through the o_s port has different security levels and are not limited to security level s. These security assignment corresponding to the decentralized atomic

components are given following a set of propagation rules. In this section we formally present the secure decentralized atomic component B^{SR} and the security assignment rules.

Example 6 *Considering a descriptif example depectited in figure 5.1 that shows a set of atomic components B_1, B_2 and B_3. The B_2 component contains two control states l_2 and l_3 and three ports p_3, p_4 and p_5. Initially at state l_2, the transition labelled by p_4 can only occur if the transition labelled by p_3 was executed once, at least, and the variable x is incremented. The exported variable x is associated to the port p_5. The dashed squares represent security annotations and will be presented in the coming sections.*

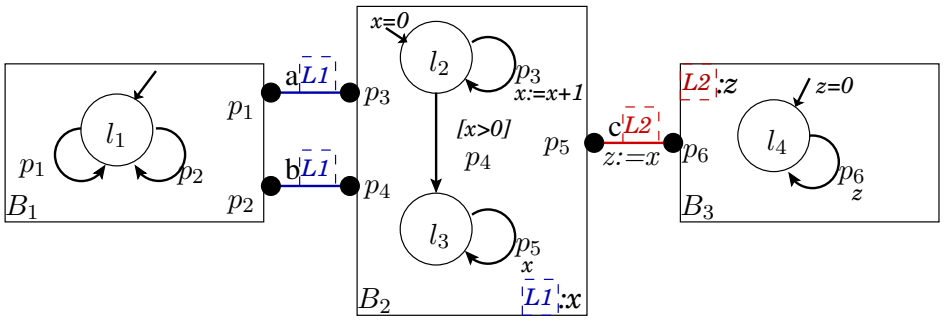

Figure 5.1 – Composite component

Example 7 *Figure 5.2 represents the transformed S/R version of the atomic component B_2 presented in Figure 5.1. The component is initially in control state $\perp_{l2}$. It sends an offer through the corresponding offer port o_{2L1} containing the current enabled ports x_{p_3}, x_{p_4} and the participation number n_{2L1}, then reaches state l_2. In that state, it waits for a notification on either port p_3 or p_4. The notification on p_3 triggers the execution of the update function which consists on incrementing the variable x, incrementing the value of n_{2L2} and re-evaluating x_{p_3} and x_{p_4} based on the guards of transition labelled with p_3 and p_4 from l_2.*

Definition 21 (security assignment σ^{SR} for B^{SR}) *The security assignment σ^{SR} is defined as an extension of the original security assignment σ. For variables X^{SR} and ports P^{SR} of a transformed atomic components B^{SR}, define*

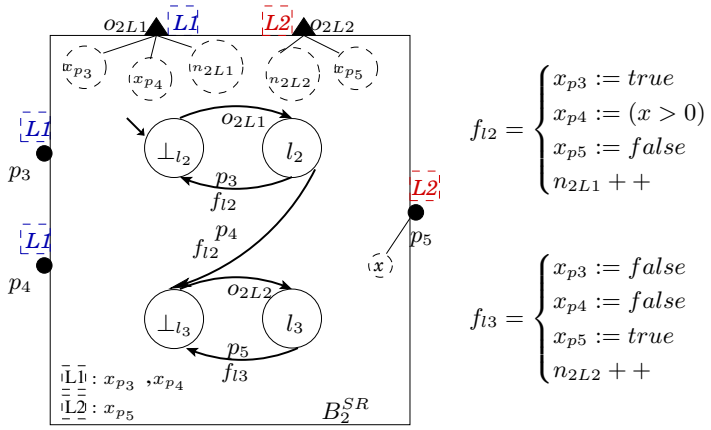

Figure 5.2 – Transformation of atomic component B_2 (Figure 5.1).

$$\sigma^{SR}(x) = \begin{cases} \sigma(p) & \text{if } x = x_p \text{ for some } p \in P \\ s & \text{if } x = n_s \text{ for some } s \in S \\ \sigma(x) & \text{otherwise, for any other } x \in X^{SR} \end{cases}$$

$$\sigma^{SR}(p) = \begin{cases} s & \text{if } p = o_s \text{ for some } s \in S \\ \sigma(p) & \text{otherwise, for any other } p \in P^{SR} \end{cases}$$

As an illustration, we reconsider the example depicted in Figure 4.1. Following the above definition and as depicted in Figure 5.2, ports $crequest, cconfirm$ and o_{ecL_1} are tagged with (L_1) and respectively ports $cget, cpush$ and o_{ecL_2} are tagged with (L_2), where $L_1 \subseteq L_2$.

Lemma 9 *If the security assignment σ satisfies the security conditions for the atomic component B then the security assignment σ^{SR} satisfies the security conditions for the transformed S/R component B^{SR}.*

Proof 6 (Lemma 9) *We show that transformed S/R atomic components are secure by construction, that is, security conditions (*i*), (*ii*), (*iii*) and (*iv*), related to events and data are preserved by the transformation.*

— *Condition (*i*): In a transformed atomic B^{SR} component we distinguish two cases of conflicting transitions:*

1. $\tau_1 = \perp_\ell \xrightarrow{o_s} \ell$ *and* $\tau_2 = \perp_\ell \xrightarrow{o_{s'}} \ell$
2. $\tau_1 = \ell \xrightarrow{p_1} \perp_{\ell_1}$ *and* $\tau_2 = \ell \xrightarrow{p_2} \perp_{\ell_2}$.

*From hypothesis, B annotated by σ satisfies security conditions. Hence, from condition (*i*) related to conflicting transitions in B the*

case (1) can not take place since ports labelling outgoing transitions of state ℓ have the same security level and, moreover, in case (2), it implies that $\sigma^{SR}(p_1) = \sigma^{SR}(p_2)$.
Similarly, in the transformed component there are two cases of causal transitions:

1. $\tau_1 = \bot_\ell \xrightarrow{o_s} \ell$ *and* $\tau_2 = \ell \xrightarrow{p} \bot_{\ell_1}$ *and*
2. $\tau_1 = \ell \xrightarrow{p} \bot_{\ell_1}$ *and* $\tau_2 = \bot_{\ell_1} \xrightarrow{o_{s'}} \ell_1$

*By construction, in (1) $\sigma^{SR}(o_s) = \sigma(p)$. Hence, condition related to causal transitions is verified and $\sigma^{SR}(o_s) \subseteq \sigma(p)$. In (2) $\sigma^{SR}(o_{s'}) = \sigma^{SR}(p')$ such that p' belong to the set of ports labelling outgoing transitions of ℓ'. By assumption, initial atomic component B satisfies security conditions, thus $\sigma(p) \subseteq \sigma(p')$ and by construction $\sigma^{SR}(p) \subseteq \sigma^{SR}(p')$. Therefore $\sigma^{SR}(p) \subseteq \sigma^{SR}(o_{s'})$ which satisfies security conditions (*i*).*

— *Condition (*ii, iii*): we verify the security level consistency of variables assigned in transitions. All actions defined on transitions of atomic component B are kept unchanged in B^{SR} and the security level of all variables are preserved with σ^{SR}. Hence, by construction these actions are still secure and satisfy conditions (*ii*) and (*iii*). The x_{p_i} variables of enables ports p_i on a state $\{\bot_{\ell_i}\}_{\ell_i \in L^{SR}}$ are modified at the received notification transition labelled with port p at the same state $\bot_{\ell_i}$ where (1) $\sigma^{SR}(x_{p_i}) = \sigma^{SR}(p_i)$. From condition (*i*), we have security level consistency of causal transition, thus (2) $\sigma^{SR}(p) \subseteq \sigma^{SR}(p_i)$. Each variable x_{p_i} is evaluated according to the guard $g_{\tau_{p_i}}$. For all $y \in g_{\tau_{p_i}}$ we have (3) $\sigma^{SR}(y) \subseteq \sigma^{SR}.(p_i)$. From (1), (2) and (3) we can deduce that the condition (iii) is preserved for all x_{p_i} variables. The participation number n_s is only incremented with notification transitions labelled with port p having the same security level s. Thus $\sigma^{SR}(p) \subseteq \sigma^{SR}(n_s)$, condition (ii), is valid in all transitions.*

— *condition (*iv*): This condition is trivially verified whenever the atomic component B is deterministic where, for every state there is at most one transition that is labelled by each port.*

5.1.2 Interaction Protocol Layer

The *Interaction Protocol Layer* consists of a set of components, each in charge of execution of a subset of interactions in the initial *secureBIP* model. Each component represent a scheduler that receives messages from S/R components then calculates the enabled interaction and selects them

for execution. Hereafter, we focus on how to secure the interactions managed by these engines in a constructive way by preserving initially defined conditions.

Here we consider that each IP_s component is a security-aware scheduler that takes into account interaction's security level as well as participation numbers of all security domain initially defined in the centralized *secure-BIP* model. We follow isolation at security levels while representing *IP* components. That is, all interactions having the same security level s are managed by the same IP_s. However, several IP_{js} component can handle subsets of the same security level interaction, s. In this case, conflicts can occur between these interactions. In case of conflicting interactions, the IP_{js} components has to communicate with the upper layer *CRP layer* to take decisions and select the interactions to execute.

However, while deciding which interaction to execute, mixed security level variables can be handled by IP_s components where s is most restrictive security level of these variables, . The use of the *IP* components allow parallel execution at components level as well as interactions level in a distributed environment. In this section we show how to construct a secure *IP* without introducing unexpected behavior nor disallowing interleaving, which represents a compromise between liveness and security property in distributed systems. Indeed, a parallel execution of two distinct security level interaction would not need to suspend one of them to maintain security, thus there will be no potential for a delay in executing some interactions.

The security annotations are propagated in the distributed model with respect to the following rules:

Definition 22 (security assignment σ^{SR} for IP_s) *The security assignment σ^{SR} is built from the original security assignment σ. For variables X^{IP} and ports P^{IP} of the IP_s component that handles γ_s, we define*

$$\sigma^{SR}(x) = \begin{cases} \sigma(x) & \text{if } x \in X_p \text{ and } s \subseteq \sigma(x) \\ s & \text{otherwise} \end{cases}$$
$$\sigma^{SR}(p) = s, \text{ forall } p \in P^{IP}$$

Within IP_s component, all ports are annotated with security level s. Whereas, the security assignment σ^{SR} maintains the same security level for all variables having their level greater than s in the original model and upgrades the others to s. That is, all variables within the IP_s component will have security levels at least s. This change is mandatory to ensure consistent copy of data in offers (resp. notifications) from (resp. to) components to the IP_s.

Example 8 *Figure 5.3 illustrates the IP_{L1} component constructed for interactions $\gamma_{L1} = \{p_1p_3, p_2p_4\}$ for the example shown in Figure 5.1. For all B_i components involved in interactions γ_{L1}, we introduce a waiting (w_i) and receiving (rcv_i) places (i.e, (w_1 ,w_2) and (rcv_1, rcv_2). For all ports p involved in γ_{L1} we introduce a sending place s_{p_i} (i.e, (s_{p_1}, s_{p_2}, s_{p_3}, s_{p_4}). The* IP_{L1} *component moves from w_i to rcv_i whenever it receives an offer from the corresponding component B_i. After choosing and executing interactions, the IP_{L1} component moves to sending (s_p) places to send notification through ports p to the corresponding component.*

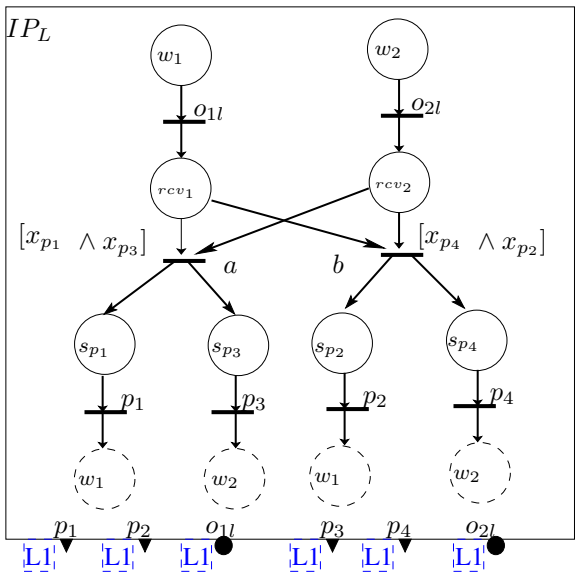

Figure 5.3 – IP_{L1} Event-secure interactions scheduler component

Example 9 *Figure 5.4 (a) presents a data transfer between two atomic components B_1 and B_2 on a synchronized interaction c where the variable y from component B_2 is assigned to the variable x from variable B_1 if $(y < 5)$. The variable $x$is tagged with $L2$ annotation and variable y is tagged with $L2$ annotation where $L1 \subseteq L2$. In the decentralized model shown in Figure 5.4 (b), the IP_{L2} component executes interaction c. To this end, variable x is imported into a same security level variable x', while the variable y is imported into a higher security level variable y', through the corresponding offer. Once the interaction takes place, x' is copied back to x on the notification transition. No copy is performed back to the y variable. Here we manage different level variable in the same IP schedulers.*

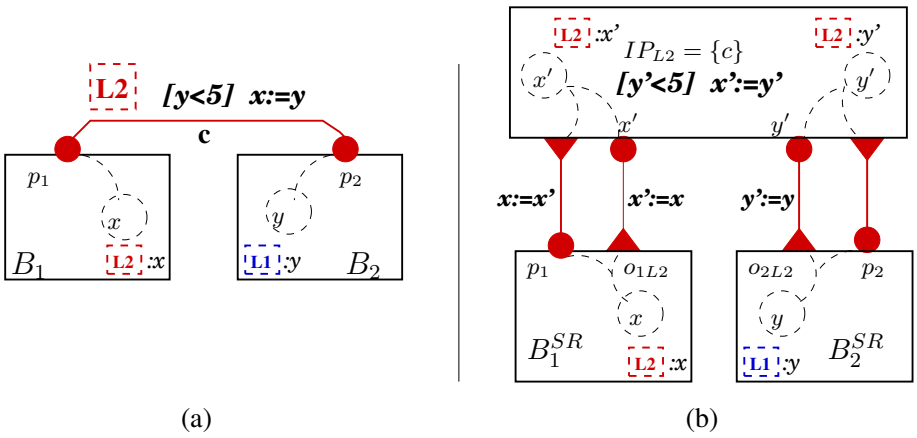

Figure 5.4 – Secure data exchange between atomic and IP components.

Figure 5.5 represents the distributed model of the system shown in Figure 5.1 with centralized interaction management. Indeed, low-level security interactions a and b are managed with a single IP_{L1}, where high level interaction c is managed with a centralized IP_{L2} component.

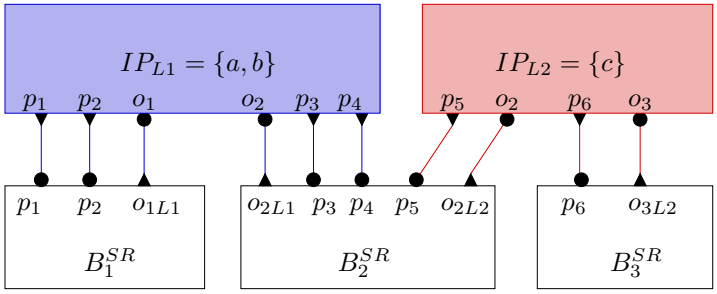

Figure 5.5 – Centralized interaction management

5.1.3 Conflict-Resolution Layer

When trying to improve system performance, we split the set of same security level interactions γ_s into distinct subsets of interactions γ_{js} that are executed by different IP_{js} components. Hence, as previously explained in chapter 2, conflicts can occur between interactions handled by distinct IP_{js} components. In case of conflicting interactions, the *IP layer* has to communicate with the upper layer *CRP layer* to take decisions and select

the interaction to execute. Here, we present a partition of *CRP*layer component to resolve conflicts as well as preserving information flow security of conflicting interactions. To preserve non-interference, however, conflicts between interactions having the same security level s and hosted by distinct IP_{js} components must be solved by a dedicated CRP_s component. That is, a distinct conflict resolution component is actually needed for every security level. This solution is adequate to the fact that, as explained earlier, conflicts can occur only between interactions having the same security level, due to the assumption that *secureBIP* system satisfies the security conditions.

The functional behavior of the CRP_s component is preserved and it's formal definition is given in 8 from chapter 2. We propagate the annotations in all ports and variables of the CRP_s component according to the two following rules.

Definition 23 (security assignment σ^{SR} for CRP_s) *The security assignment σ^{SR} is built from the original security assignment σ. For variables X^{CP} and ports P^{CP} of the CRP_s component that manage a set of conflicting interactions a, we define*

$\sigma^{SR}(x) = s$, *forall* $x \in X^{CP}$

$\sigma^{SR}(p) = s$, *forall* $p \in P^{CP}$

Cross-layer Interactions

In this subsection, we define the interactions in the 3-layer model. Following Definition 6, we introduce S/R interactions by specifying the sender and the associated receivers. Given a composite component $C = \gamma(B_1, \cdots, B_n)$, and partitions $\gamma_{1s}, \cdots, \gamma_{ms}$ of $\gamma_s \subseteq \gamma$, for every security levels $s \in S$, the transformation produces a S/R composite component $C^{SR} = \gamma^{SR}(B_1^{SR}, \cdots, B_n^{SR}$,
$(IP_{1s}, \cdots, IP_{ms})_{s\in S}, \{CRP_s\}_{s\in S})$. We define the S/R interactions of γ^{SR} as follows:

- For every atomic component B_i^{SR} participating in interactions of security level s, for every IP components $IP_{1s}, \cdots, IP_{ms}$ handling γ_s, include in γ^{SR} the *offer* interaction $off_s = (B_i^{SR}.o_s, IP_{1s}.o_{is}, \cdots, IP_{ms}.o_{is})$.
- For every port p in component B_i^{SR} and for every IP_{js} component handling an interaction involving p, we include in γ^{SR} the *response* interaction res_p=($IP_{js}.p$, $B_i^{SR}.p$)
- For every IP_{js} component handling an interaction a that is in conflict with an other interaction a' handled by some different IP, include

in γ^{SR} the *reserve* interaction r=($IP_{js}.r_a$, $CRP_s.r_a$). Likewise, include in γ^{SR} the *ok* interaction ok=($CRP_s.ok_a$, $IP_{js}.ok_a$) and the *fail* interaction f=($CRP_s.f_a$, $IP_{js}.f_a$).

Definition 24 (security assignment σ^{SR} for γ^{SR}) *The security assignment σ^{SR} is build from the security assignment σ. For interactions γ^{SR} between all atomic components of the transformed model, we define $\sigma^{SR}(a) = s$ for any interaction a involving an IP_{js} component handling interactions with security level s.*

Lemma 10 *All the cross-layer interactions of C^{SR} are secure with σ^{SR}.*

Proof 7 (Lemma 10) *We verify the security level consistency of transferred data on different interactions:*

- *At* offer *interaction off_s, we perform a copy of received variables through offer ports from B_i^{SR} component to the IP component, such that $\forall x \in \{\{X_p\}_{p\in P} \cup \{x_p\}_{p\in P} \cup \{n_{si} | B_i \in participants(\gamma_s), s \in S\}\}$ there exist a $x' \in X^{IP}$ where $x' := x$. By transformation, $\sigma^{SR}(x') = s$ if $\sigma(x) \subseteq s$ and $\sigma^{SR}(x') = \sigma(x)$ otherwise. The security level of updated variables (x') and used variables (x) are consistent with security level of there corresponding offer interaction, where $\sigma^{SR}(off_s) = s$. Thus, the security condition (ii) and (iii) are preserved at offer interactions.*
- *At* response *interactions res_p, we send notifications to the corresponding ports p associated with the updated variables x', where $\sigma^{SR}(p) \subseteq \sigma^{SR}(x')$. By construction, $\sigma(p) = \sigma^{SR}(p) = \sigma^{SR}(res_p)$, thus, $\sigma^{SR}(res_p) = \sigma^{SR}(x')$ which satisfies condition (iii).*
- *With interactions r, f and ok, the only exchanged variables are the participation numbers n_{si} of each participating component i. By construction, each CRP_s and IP_s components are handling interactions of the same security level s, thus $\sigma^{SR}(r) = \sigma^{SR}(f) = \sigma^{SR}(ok) = \sigma^{SR}(n_{si})$. Therefore the security condition (iii) is preserved at these interactions.*

The following theorem states the correctness of our transformation, that is, the constructed S/R model satisfies the security conditions by construction.

Theorem 11 (Security-by-construction) *If the component C satisfies security conditions for the security assignment σ then the transformed component C^{SR} satisfies security conditions for the security assignment σ^{SR}.*

5.2 Use-Case: Decentralized WhenApp Application

In this section, we resume the Whens-App application previously presented in Chapter 4 where we already showed that both requirements related to the event and data non-interference are ensured by using security annotations for tracking the information flow in the system. Here, we show how the annotated model can be automatically and systematically transformed towards a distributed implementation while preserving the security properties.

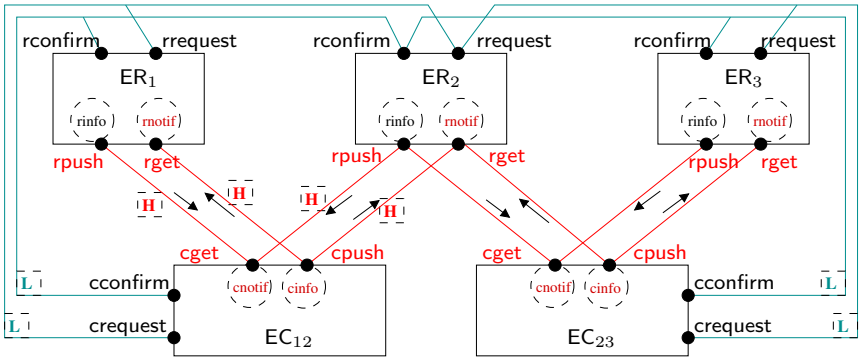

Figure 5.6 – Example of the Whens-App system.

Figure 5.6 presents a simplified composite component for an instance of the Whens-App application with two event creators and three event receivers. Interactions are represented using connecting lines between the interacting ports. Binary interactions (rpush cget) and (cpush rget), to store and report information, include data transfers between components, that is, assignments of data across interacting components.. The decentralization of this model is done mainly in two steps: first the transformation of the atomic components to hold message passing and breaking the atomicity of executions and second by introducing *IP*and *CRP*components according to a centralized or decentralized management of the interactions.

Atomic component transformation: From the Whens-App system, we take as example the *Event-Receiver* component to transform it. The component transformation and the extended security assignment for the *Event-Receiver* are depicted in Figure 5.7. Variables n_L , $x_{request}$, $x_{confirm}$ and the offer port o_L are assigned to Low (L). Variables n_H , x_{push} , x_{get} and

the port o_H are assigned to High (H). Ones can check that this assignment obeys all the (local) security conditions related to B^{SR} . Actually, security conditions are preserved along the proposed transformation of atomic components with respect to extended security assignment.

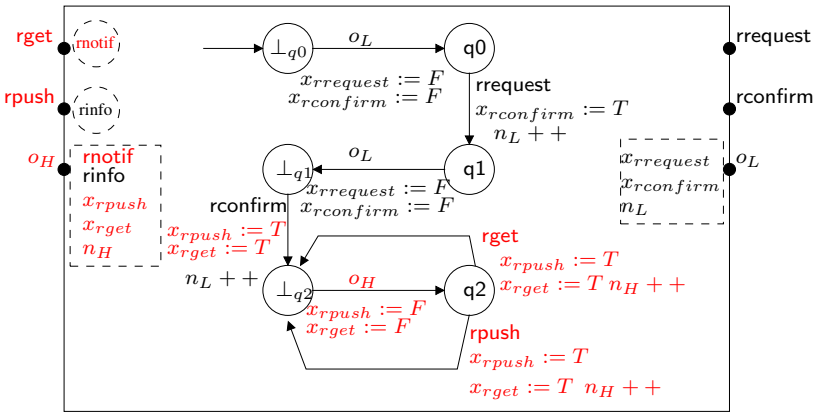

Figure 5.7 – Transformation of atomic components illustrated on the Event Receiver

Interactions management: Here we give an example of a centralized interaction management where different sets of interactions are handled by *IP*components from the same security level.

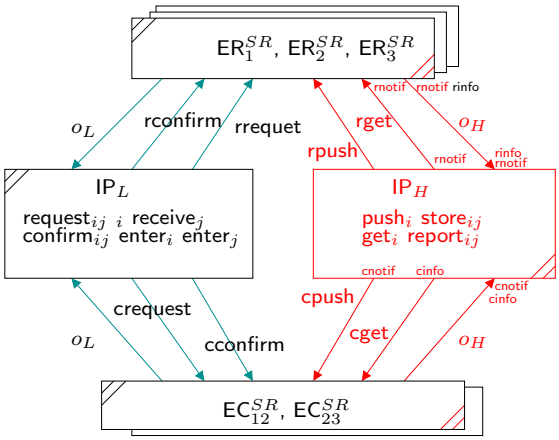

Figure 5.8 – Decentralized model for the WhensApp example.

The security assignment σ^{SR} is naturally lifted from offer/notification

ports to the interactions of γ^{SR}. Intuitively, every S/R interaction involving component IP_s has security level s. The construction is illustrated for the running example in Figure 5.8. We omitted the representation of ports and depict only the interactions and their associated data flow. In particular, consider the rinfo variable of Event Receiver which is upgraded to H when sent to IP_H and not sent back on the notification of the push interaction.

5.3 Conclusions

In this chapter, we present a practical approach to automatically secure information flow in distributed systems. Starting from an abstract component-based model with multiparty interactions, *secureBIP* , we verify security policy preservation as defined by the user, that is, verifying non-interference property at both event and data levels. Then, we generate a distributed model where multiparty interactions are replaced with asynchronous message passing. We apply to the generated model a set of rules that aim to preserve information flow security. We show that the obtained distributed model is "secure-by-construction" where all the sufficient conditions defined at the centralized *secureBIP* model are preserved at transformation.

Chapter 6

Implementations

Contents

This chapter presents the implementation of different methods for security using the *BIP* framework. We start by presenting the *BIP* contextual language and it's extension for security. Then we present in Section 6.2 an overview of the *secureBIP* tools with focus on verification and code generation tools implementing approaches presented in previous chapters.

6.1 The BIP Language

The *BIP* language provides structural syntactic constructs for describing systems. The language entails components which behavior is defined using variables, data type declarations, expressions and statements written in C language and the coordination between components are defined through connectors. The basic constructs of the *BIP* language are the following:

- atomic component: the main artifact of the model, described with a behavior defined with a set of transitions and communicating with a set of interfaces (ports).
- connector: specifying synchronization between ports of components associated to a set of actions allowing the transfer of data between components.
- composite component: specifying the hierarchy of the system by assembling instance of components (atomic or compound) using connectors.
- package: specifying the top level of the system and encapsulating the definition of the components and types declarations.

6.1.1 Language Features

We give, hereafter, an example of the *Event-Creator* component given in Figure 2.2 from Chapter 3. In the *BIP* language, we start by defining different types (ports, atom and connectors) that are later instantiated and used in system composition. Each used port and variable has a type. The port type defines a generic name for each variable exported by such a port.

```
#include <stdlib.h>
package WensApp
    port type dataport(string msg)
    port type eport()

connector type multiparty(eport p1, eport p2, eport p3)
  define p1,p2,p3
  on p1 p2 p3
```

```
    down {}
end

connector type SendRec(dataport p1, dataport p2)
  define p1 p2
  on p1 p2
    down {p1.var=p2.var;}
end
```

In this given piece of code we describe the different types of ports that are used in our case, that is, either event port type, *eport*, or a port extended with variables *dataport*. The *BIP* language supports most of the data types defined with the C language by just importing their corresponding definitions. Here, to be able to use string type for both $cinfo$ and $cnotif$ variables we imported the stdlib library. We also defined connector types that are parameterized by a list of port types that describes their support. The **define** construct defines the set of interactions that are allowed by the connector. In the example, we have two connectors allowing a multiparty (ternary) interaction and an other binary one. For each of them a guard and an update function can be provided. In the given example no guard is used on the interaction and the update function is provided with the **down** construct. To access variables associated to the ports, we use the dotted notation, **p1.var**.

```
atom type event-creator()
    data string cnotif, cinfo

    export port eport crequest(), cconfirm()
    export port dataport cget(string cnotif)
    export port dataport cpush(string cinfo)
    port eport cancel()

    place l1,l2,l3,l4
    initial to l1 do {}
    on crequest from l1 to l2
    on cconfirm from l2 to l3
    ....
end
```

The description of an atomic component starts with declaring variables, ports and control locations. Upon port declaration, some variables are bound to the port and their type match those in the port type definition. Using the construct **initial** we specify the initial control location and possibly some initialization functions. The transitions are declared with the trigger port (after **on**), a source location (after **from**) and a destination location (after **to**). The actions defined at each transition can be declared

with a guard (after provided) and update function (after do) Each transition of the behavior is declared, with a port (after on), a (set of) initial and final state(s) (after from and to), a guard (after provided) and an update function (after do). The functions and guards are written using a subset of the C syntax.

```
compound type WensApp()
  component event-creator EC()
  component event-receiver ER1()
  component event-receiver ER2()

  connector multiparty request(ER1.rrequest1, ER2.rrequest2, EC
      .crequest)
  connector multiparty confirm(ER1.rconfirm1, ER2.rconfirm2, EC
      .crequest)
  connector SendRec out_in1(EC.cinfo, ER1.rnotif)
  connector SendRec out_in2(ER2.rinfo, EC.cnotif)
  ...

end
```

The compound component is a new component type that contains instances from existing atomic component types. The interactions (communications) between the instances of atomic components is defined by instantiating connectors between them. The compound component offers the same external interface as an atomic component, hence a compound component can have an exported port connected to an atomic component forming a hierarchical composition of the system.

6.1.2 Security Extension

```
atom type event-creator()
    @security(pl="event-creator:event-receiver1,event-receiver2"
        )
     data string cnotif, cinfo

     export port eport crequest(), cconfirm()
     export port dataport cget(string cnotif)
     export port dataport cpush(string cinfo)
     port eport cancel()

     ....
end
```

To specify the security levels to the *BIP* model, we add annotations to different ports and variables in atomic components. The security annota-

tion is declared using the **@security** construct and confirms to the label model adopted. Using the Decentralized label model, we define the security label as a set of rules constructed from *owners* and *readers* and defined in the constructor **pl**. For instance, the security label *@security(pl="event-creator: event-receiver1,event-receiver2")* defined before a variable or a port states that the owner of the information is the *Event-Creator* and readers are *Event-Receiver1* and *Event-Receiver2*.

6.2 Tool-set Implementation

This section presents the complete implementation of the different tools available for the *secureBIP* framework. The tool set we provide aims to facilitate the modeling, verifying and executing of *secureBIP* models. A overview of the tool-set is shown in Figure 6.1. Here, we distinguish between three categories of tools, languages transformations, verification and Execution. We now detail each of theses categories.

6.2.1 Languages Transformations

According to the application domain, the verification of security property requires to translate into *secureBIP* existing languages from different model of computation. These languages can model the application software, the hardware architecture or both of them. Here, we consider as examples of input language for the Tool-set the *Business Process Execution Language* (BPEL) [**?**, **?**] or *Architecture Analysis and Design Language* for multi-independent level model(AADL) [**?**]. BPEL is a software language for Web Services composition while AADL is a modeling languages that combines hardware aspects such as memory, buses and processors and software component such as threads and systems.

From BPEL to *BIP*

BPEL provides structuring mechanisms to compose several WS into a new one. We particularly focus on *BPEL4WS* [**?**] processes which compose services from activities, that are either (1) *basic* such as receive, reply, invoke, assign, throw, exist, or (2) *structured* such as sequence, if, while, repeatuntil, pick, flow.

The representation of *BPEL* processes in our component model is structural. A process is represented as an atomic component where the behavior encompasses all its basic and structured activities. All process variables

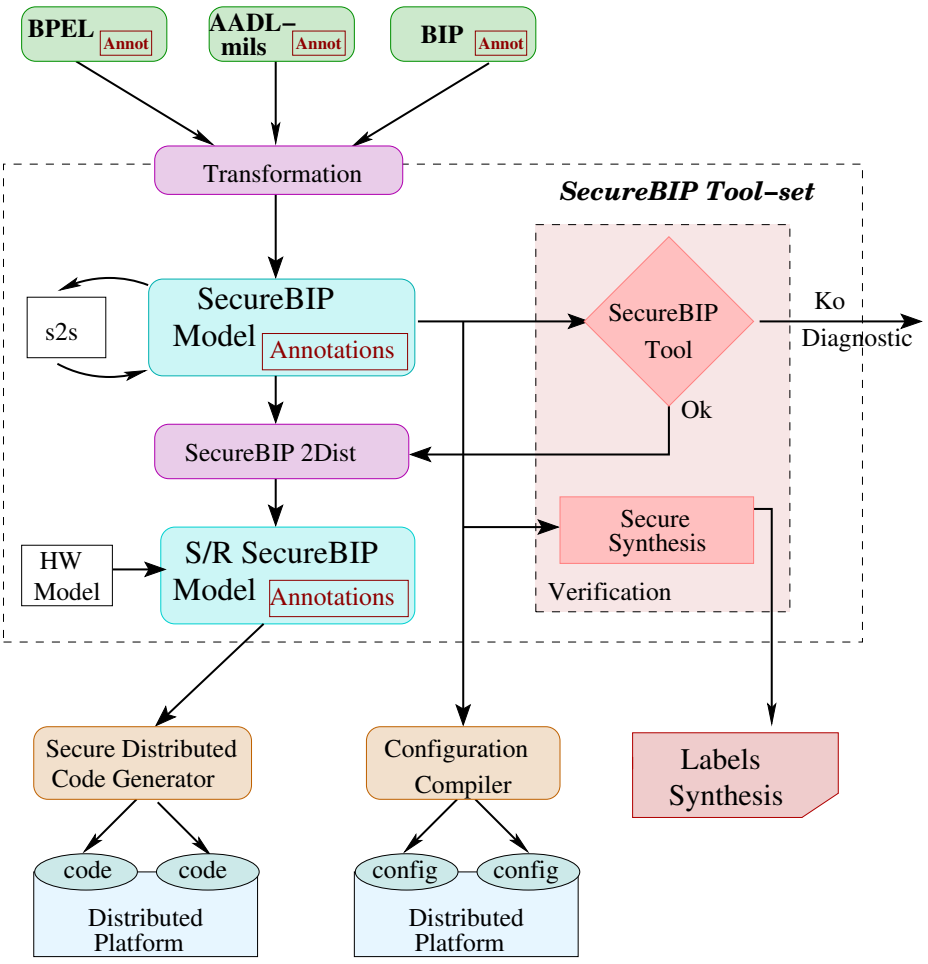

Figure 6.1 – *Securebip* Tool-set overview.

are added to the atomic component. Basic activities such as ⟨*receive.../*⟩, ⟨*replay.../*⟩, ⟨*invoke.../*⟩ are translated into specific transitions triggered by respectively *Receive_** and *Replay_** ports. Their corresponding variables are implicitly attached to the above ports. The ⟨*assign.../*⟩ activity is translated as an internal transition that executes the corresponding assignment.

Structured activities define the overall control flow of transitions in the atomic component, in the usual way. In particular, transition guards are extracted from ⟨*if..., while..., repeatuntil.../*⟩ and ⟨*pick.../*⟩ activities. Nevertheless, as atomic component have behaviour expressed using automata, the parallel execution of ⟨*flow.../*⟩ activities is not fully translated. In order to catch data dependencies, however, their execution is sequentialized within the component.

Finally, we define the connectors and the composition of the atomic components by using the *PartnerLinks* defined for *BPEL* processes. Every ⟨*invoke.../*⟩, ⟨*receive.../*⟩ and ⟨*replay.../*⟩, ⟨*receive.../*⟩ interaction defined over partner links is translated to a connector relating the corresponding components and their respective ports. Let us notice that processes may interact through partner links with external WS, that is, developed in other languages than *BPEL* (such as Java, C, etc). In this case, these WS are represented as atomic components with an implicit behaviour, for arbitrarily sending and receiving data through their connected ports.

Similar translations have been already defined in the literature [**?**]. As the above translation is structural the resulting model remains comprehensive for the WS designer. The representation relies basically on adapting reusable and composable model components that directly maps processes with limited numbers of execution steps. Despite that, some features in *BPEL* language are not considered such as fault/event handling and scopes. Security errors that can be generated by these aspects are not in the scope of this paper.

From MILS-AADL to BIP

D-MILS overview: Multiple Independent Levels of Security (MILS) [**?**] is an approach is a highly assure security architecture based on concepts of controlled information flow and of separation [**?**]. The design and implementation of critical systems following the MILS approach involves two principal phases: the designing of an abstract architecture intended to achieve the stated functional and non-functional properties, and the implementation of that architecture on a robust technology platform. During the first phase, the properties that the system is expected to exhibit are

defined and the contributions to the achievement of those properties by the architectural structure and the behavioral attributes are analyzed and justified.

The architecture defines an information flow pattern that is intended to reflect a policy architecture while some behavioral properties, specific components of the architecture enforce some local policies. The combination of the policy architecture and local policies in a compositional reasoning may establish useful system level properties.

The MILS platform provides a separation kernel based technology [**?**] that establishes and enforces the abstract system architecture according to its configuration data. The properties that the abstract system architecture are intended to satisfy are assured not only on the analysis of its design but also by the correct implementation and deployment of that design. Indeed, the MILS platform generates configuration for the separation kernel that must faithfully implement the specified abstract architecture. Using a configuration compiler that is driven by the architecture and constraints of the target platform synthesizing semantically correct configuration that corresponds to the specified architecture. Based on the MILS approach capacity to implement a single unified policy architecture to a network of separation kernels, the D-MILS projects [**?**] proposes an extension for the use separation kernel networking, where each separation kernel is combined with a new MILS foundational component and communicates with each others using a MILS networking system (MNS) producing the effect of a distributed separation kernel.

To achieve all the requirements correctly for critical applications, an automated approach as presented in 6.2 is followed providing a tool chain that takes as input a declarative model of the system expressed in MILS dialect of the Architecture Analysis and Design Language (AADL) [**?**], facts about the target hardware platform, properties of separately developed system components, designer imposed constraints and system property specifications, and human guidance to the construction of the assurance case. The tool chain components perform several tasks that are (1) parsing of the languages [**?**], (2) transforming input model among the various internal forms [**?**, **?**], (3) analysis and verification [**?**], (4) configuration data synthesis and rendering [**?**], and (5) pattern-based assurance case construction [**?**, **?**]. Outputs of the tool chain include, proofs of specified system properties, configuration data for the D-MILS platform.

The transformation from MILS-AADL relies on a general method for generating *BIP* models from languages with well-defined operational semantics. As depicted in Figure 6.2, this method involves the following

three steps for a given application description written in a language $\mathcal{L}$:

— Translation of atomic components of the source language into *BIP* components. The translation focuses on the definition of adequate interfaces. It encapsulates and reuses data structures and functions of the application description

— Translation of coordination mechanisms between components of the application into connectors and priorities in the target *BIP* model

— Generation of a *BIP* component modeling the operational semantics of $\mathcal{L}$. This component plays the role of an engine coordinating the execution of the application components

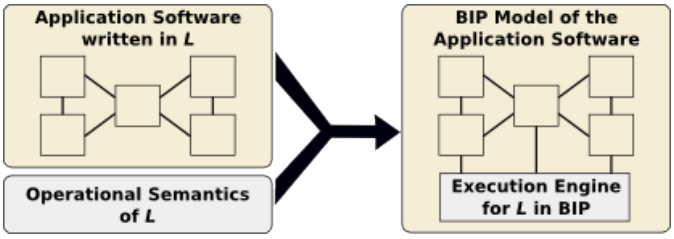

Figure 6.2 – Principles of BIP Language Factory [**?**]

The translation of MILS-AADL components in *BIP* is structural. A detailed explanation of the transformation rules can be found on [**?**]. Here we entail the transformation of main features of the input language into *BIP* :

Packages The *BIP* language provides a package structuring mechanism very similar to the one provided by MILS-AADL. A package in *BIP* allows keeping together a related collection of data, ports, connectors and component types. Packages might have dependencies on each other. An import mechanism allows to re-use definitions from existing packages when constructing new ones.

The package structure of the MILS-AADL specification is therefore fully preserved by translation into *BIP* . In addition, specific *BIP* packages are used to factorize representation for some of the common underlying features of MILS-AADL specification i.e., predefined data types and port types.

Component Types and Implementations In contrast to MILS-AADL, in *BIP* there is no syntactic separation between the component type (the interface) and the component implementation (the behavior or the internal structure). A component type definition in *BIP* defines altogether the

interface of that component (which is a set of typed ports) and its implementation, either as an explicit behavior defined by an extended state machine (or Petri net) or as a composition of already existing components using interactions and priorities.

Table 6.1 summarizes the mapping of different categories of MILS-AADL components into *BIP* .

MILS-AADL	*BIP*
process type + implementation	composite component
thread type + implementation	atomic component
data	data variable in atomic component
processor type + implementation	atomic component
memory type + implementation	atomic component
device type + implementation	atomic component
bus type + implementation	atomic component
network type + implementation	atomic component
node type + implementation	composite component
system type + implementation	composite component

Table 6.1 – Translation Overview of MILS-AADL components

The principles for translation of MILS-AADL components are the following:

— MILS-AADL components are translated in *BIP* as atomic or composite components having the interface depicted in Figure 6.3. With few exceptions, all the MILS-AADL interface features (event and data ports) are structurally mapped to *BIP* ports. Event ports are becoming *BIP* ports without data. Data ports are becoming *BIP* ports associated with a data variable of the corresponding type. In addition, *BIP* components have two additional (basic mode control) ports to be used for explicit activation/deactivation.

— For the case where the result is a composite components in *BIP* , as illustrated in Figure 6.4, it consists of a composition of one atomic *mode controller* sub-component and a number of sub-components (atomic or composite), one for every sub-components in the MILS-AADL description. The composition at *BIP* level is defined according to event and data-flow connections plus additional mode update interactions. For the case of atomic components in *BIP* , they are identical to their associated *mode controller* component.

— The *mode controller* component is responsible for (i) storing and providing access to all data features of the interface and the imple-

mentation of the MILS-AADL component and (ii) implementing the mode control behavior.

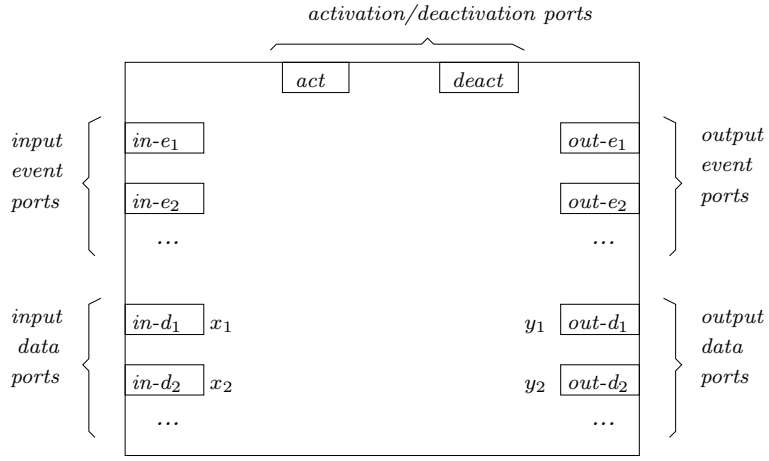

Figure 6.3 – Generic interface of MILS-AADL components in bip

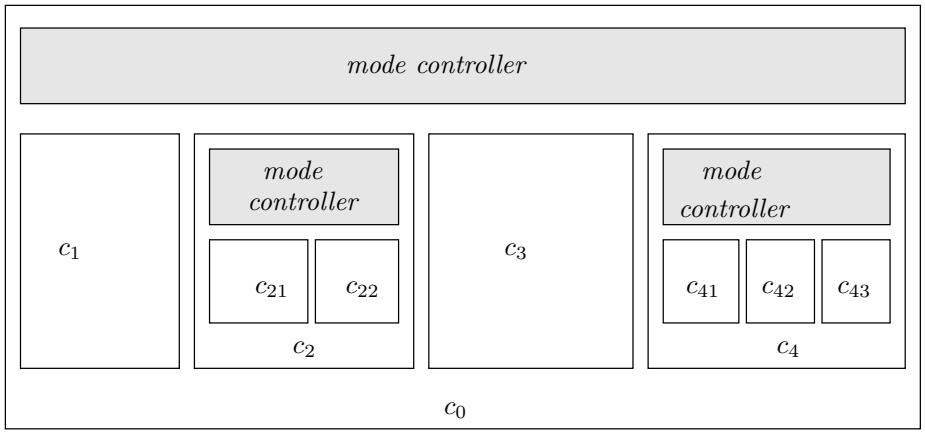

Figure 6.4 – Generic structure of MILS-AADL composite components in bip

BIP systems obtained from translation from MILS-AADL contain multiparty interactions classified into three disjoint categories:

— event-flow interactions: representing event flow communication
— data-flow update interactions: representing data flow computation according to data flow equations

— mode update interactions: representing mode update with activation/deactivation of subcomponents

Interactions are subject to the following generic priority rule: event-flow interactions $\prec$ data flow interactions $\prec$ mode update interactions

Mode Transitions and Hybrid Behavior For every MILS-AADL component, mode transitions and hybrid behavior are translated/represented by the associated mode controller component in *BIP* .

The mode controller component stores and provide access to all the data features of the associated MILS-AADL component. In addition to these data variables, the *BIP* component uses one extra boolean variable namely, *active*, to distinguish between active and inactive states.

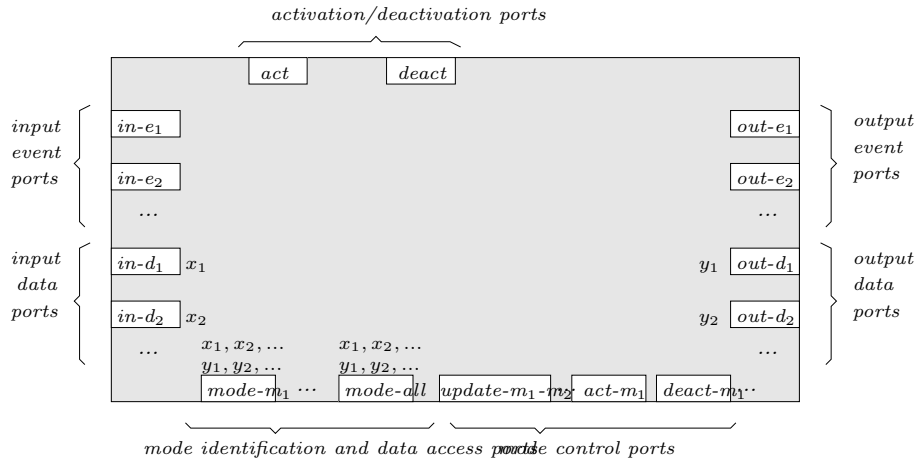

Figure 6.5 – Generic interface of mode controller components in bip

The mode controller component has the interface illustrated in Figure 6.5. The ports belong to different categories that allow either to activate/disactivate the component, receive or send event message or data ports (i.e. in/out event and data ports), mode identification ports that correspond to every component mode, or finally, update ports used to coordinate recursively the mode change of sub-components.

The behavior of the mode controller is described by a Petri net. The transitions are illustrated in Figure 6.6. Each of the previously defined categories of ports do correspond by construction to a corresponding category of transitions.

Finally, let us remark the following limitations of the translation:

— Continuous data flows and hybrid behavior are not supported.

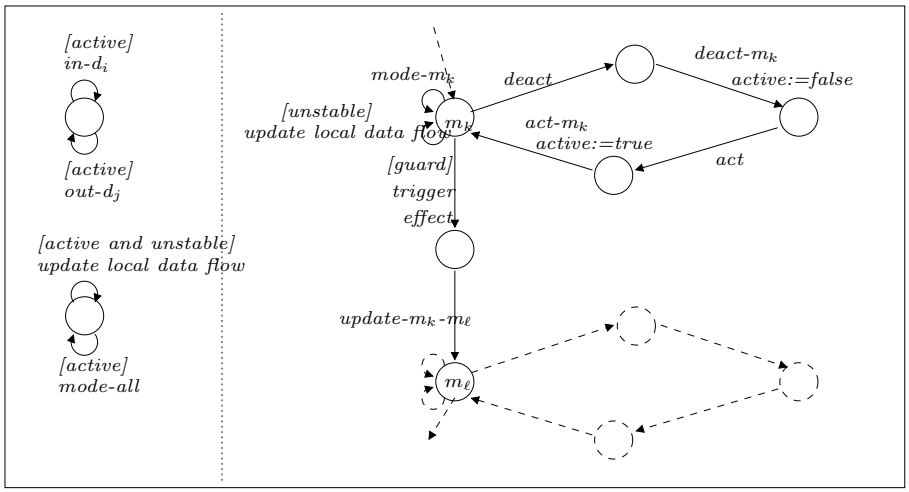

Figure 6.6 – Behaviour of mode controller

— The most general form of mode invariants is not supported. Mode invariants in ADDL-MILS need to be manually rewritten either into the restricted (acceptable) form for *BIP* or into transition urgencies in the mode controller.

Data Types *BIP* supports natively a number of elementary data types including boolean, integer, float and real-time clock. These types can be used to map directly their corresponding ones in MILS-AADL.

In addition, *BIP* allows using arbitrary abstract (opaque) data types in C. Using this mechanism, it is possible to encode and use all other MILS-AADL types, including the various forms of keys (and the associated primitives) as well as the various forms of structured data types (e.g., records, etc).

A full detailed explanation of the different categories of model transitions, Event communications and data flows transformations are given in [**?**].

6.2.2 Verification Tools

Here we present the *secureBIP* and *security synthesis* tools. Both tools allow the verification of the information flow automatically according to different initial annotation configuration.

SecureBIP Tool

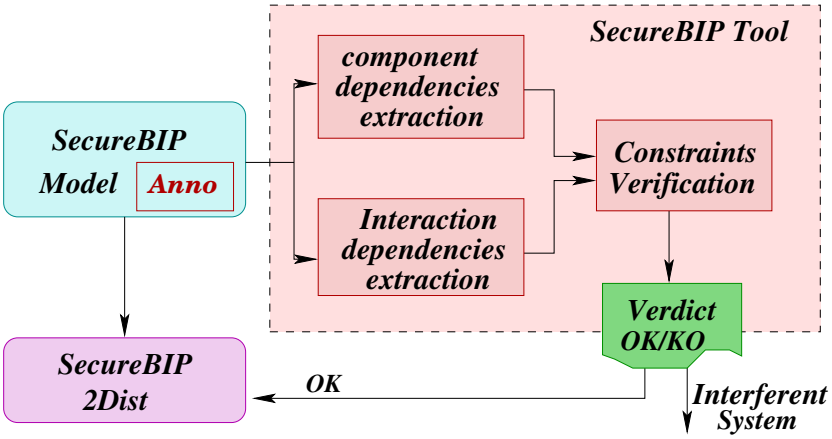

Figure 6.7 – SecureBIP tool.

To control the information security, the component-based architectures facilitate monitoring the flow of information through the explicit connections between atomic components. Security annotations are assigned to data and ports of each component in the *BIP* model. As explained in the Section 6.1.2, the annotation of security levels to different parts of a component is given in the bip file. This tool extracts dependencies in the system model at intra-component level from actions and ports labelling transitions on the atomic components behavior as well as inter-components with the transferred data at interactions, as presented in Figure 6.7. The tool requires that all data and ports are annotated. The security conditions defined in Chapter 4, are then verified.

Security Synthesis

As depicted in Figure 6.8, the synthesis tool takes as input a *secureBIP* model containing an initial annotation for a partial set of data. Then, it builds the dependency graphs of components and runs the synthesis algorithm defined in Chapter 4 at section 18 to produce the complete configuration. If the tool succeeds in generating a configuration for the system variables and ports then the system is considered secure. Moreover, the generated configuration is optimal and the less restrictive security levels are applied while propagating annotations in the system. However, if no configuration is found, the system is not secure and the tool generates a diagnostic containing the location of the security error.

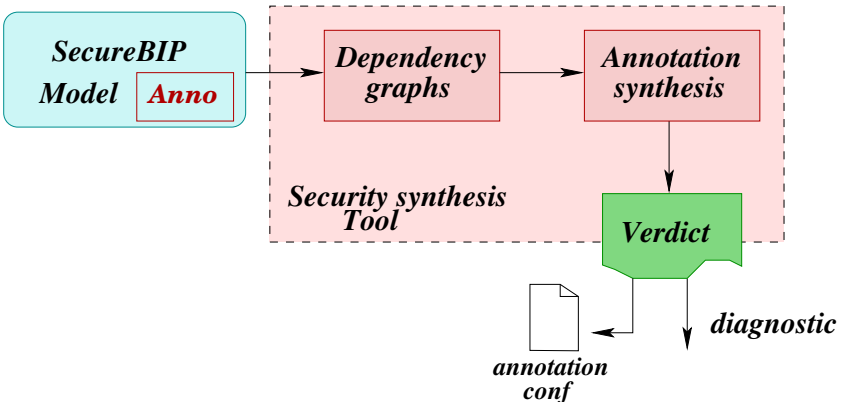

Figure 6.8 – Security annotation synthesis.

6.2.3 Execution

Decentralization and Code Generation

Here we illustrate the complete design flow for generating secure distributed code represented in Figure 6.9. In this architecture, the flow consists on configuring security at two levels, first at the abstract model and second depending on target platform. Hereafter, we first give functional implementation of the distributed *secureBIP* model and then we discuss the different steps and design choices for adding security.

Functional implementation: In decentralized model, the *IP* engine computes the set of enabled ports of each atomic component involved in interactions handled by that *IP*. Then, according to the enabled ports and guards of the interactions, the enabled interactions are computed. The engine indicates a deadlock if no interaction is possible. Then, the data transfer function of the interaction is computed. Finally, the *IP* engine sequentially executes these transitions in the corresponding atomic component to reach the next state. Hence, the *IP* engine here play the role of the coordinator in selecting and executing interactions between the components, taking into account the glue specified in the input component model.

We generate code for the distributed *secureBIP* model automatically where each element (atomic components, connectors, compound components) is implemented as a C++ class. Each of these classes calls functions that constitute the implementation of the *IP* engines. In turn, *IP* engines may call functions of the generated code when needed, for instance when

executing the data transfer function of an interaction.

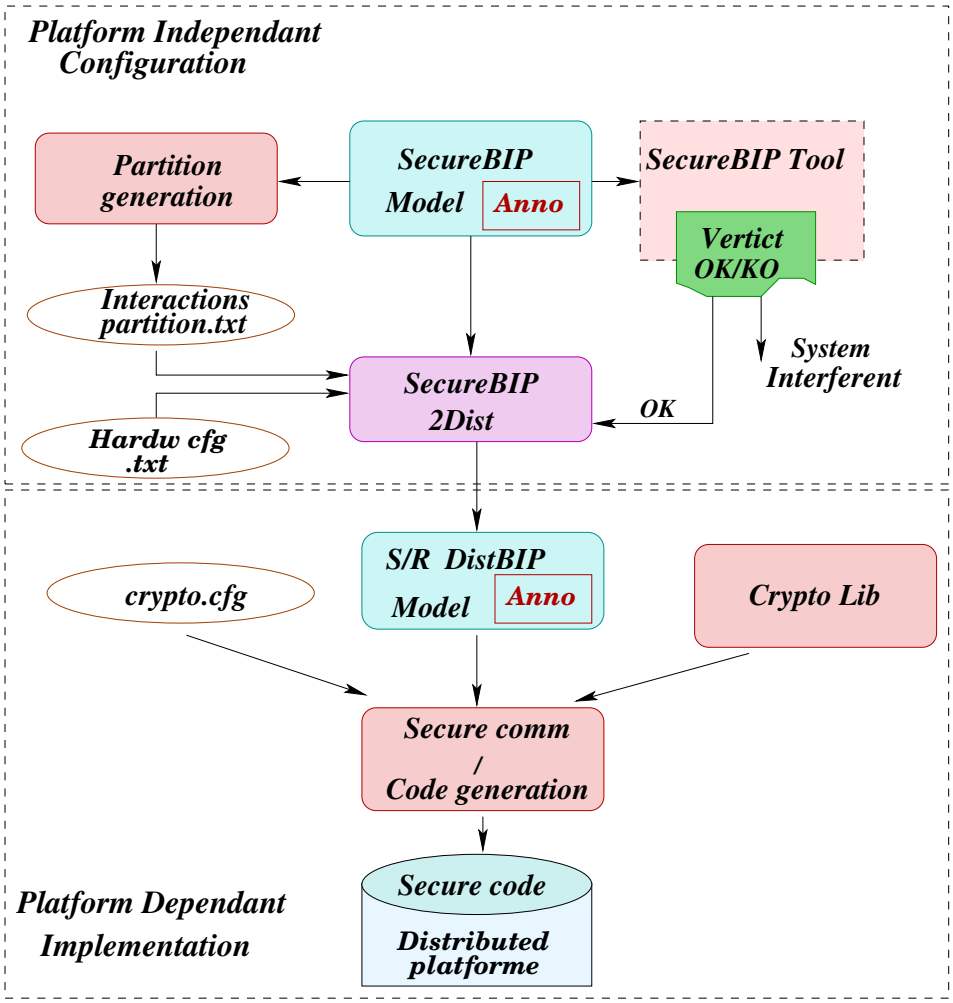

Figure 6.9 – Distributed secure code generation.

Abstract model configuration: Additionally to the system functional model (.bip), the system designer provide a configuration file (Annotations.xml) that contains the DLM annotations, previously presented in Chapter 3 where we define the acts_for relations and the labels to different ports and data in each atomic component. Figure 6.10 presents fragments of the configuration file for the *Whens-App* abstract model. We extend the system model parser to extract labels from Annotations.xml file. Then, we associate annotations to their corresponding ports and data types stored in

the secure component model. Next, the *secureBIP checker* tool browses all atomic components and interactions in the model to extract events dependencies at each local state (incoming and outgoing port labelled transitions) and data dependencies at different transition's and interaction's actions and checks their label consistency. In the case where tool verdict is positive, the tool generates automatically an interaction partition file that describes the set of interactions that each *IP*component would manage. This file is used as input by *secureBIP 2Dist* to generate an annotated S/R model. The *secureBIP 2Dist* generator is modified to encompass modifications in decentralized model as well as rules for annotations propagation.

```
- <config>
  - <acts_for>
      <authority>Event_Creator:Event_receiver1,Event_Receiver2</authority>
    </acts_for>
  - <var_config>
      <variable name="cinfo" component="Event_Creator"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      <variable name="rinfo1" component="Event_receiver1"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      <variable name="rinfo2" component="Event_receiver2"
        label="Event_Creator:Event_receiver1,Event_Receiver2" />
      ...
    </var_config>
  - <port_config>
      <variable name="cinfo" process="Prosumer1" label="Prosumer1:SMG" />
      <variable name="P" process="Prosumer2" label="Prosumer2:SMG" />
      <variable name="P" process="Prosumer3" label="Prosumer3:SMG" />
      ...
    </port_config>
  </config>
```

Figure 6.10 – Configuration file for the abstract model.

Platform dependent configuration: In this part, the system designer provides configuration file that contains the cryptographic mechanisms to be used to ensure confidentiality for data and ports to secure interactions between atomic S/R and IP components. To preserve confidentiality we use encryption. We assume that the generated code is running on trusted hosts where it is safe to generate and store encryption keys. The *Crypto Lib* library contains the different encryption protocols and functions that, following the configuration file, the code generator selects messages to secure at communications using secure TCP/IP sockets.

The configuration states the encryption mechanisms for each defined security level, that is, for variables and ports that need to be secured following the secure abstract annotations. A data security is enforced using

authentication, encryption and signature mechanisms to encrypt and sign the data at socket buffer before sending it. However we consider that encryption does only provide a degree of privacy with variables. Hence, we enforce the security of ports, if it is configured so, by hiding the message identity at sending to enforce privacy of message sender and receiver.This is done by defining a secure session where we encapsulate the sent message such that no information about the sender can be deduced by observing message transfer between components. In this message source and receiver are encrypted under a shared key between sender and receiver component. The *message_index* (common encrypted pass-world shared between sender and receiver) will be used by the receiver to retrieve the sent message. According to domain application, there exist some privacy extensions allowing the identities of the communicating parties to be hidden from third parties.

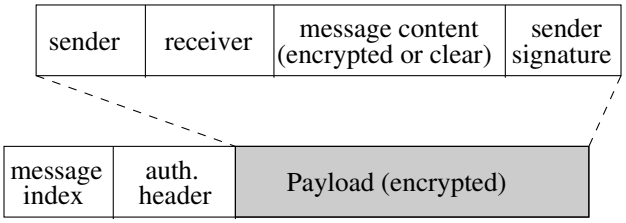

Figure 6.11 – Hide message sender identity.

Following this defined configuration, we automatically generate standalone C++ processes for every S/R components (atomic and *IP*) communicating with secure TCP/IP sockets channels that can be deployed and run on a distributed network. Each C++ process can be run on a host that ensures at least the upper bound security level of annotated data and ports in it. Obviously, it is easier to find a set of hosts that are trusted to run a process of specific security level at most than it is to find a host that can run the whole multi-level system.

Configuration Generation

As a part of the the D-MILs [**?**] project, we developed a tool-sets ensuring that a defined model meets the desired properties at implementation, by stating with configuring the target platform at deployment. A correct configuration is crucial for meeting the security and safety goals of the system. The insurance of security properties, such as non-interference is enforced by the platform where, for instance, two components (subjects)

that are not connected in the original model should not be able to communicate once deployed on the configured platform. For safety properties a low response time may be necessary, which requires both sufficient computational resources and a low-latency network. We consider that the global goals of the system are met once the requirements on the configuration such that the ones above have to be verified.

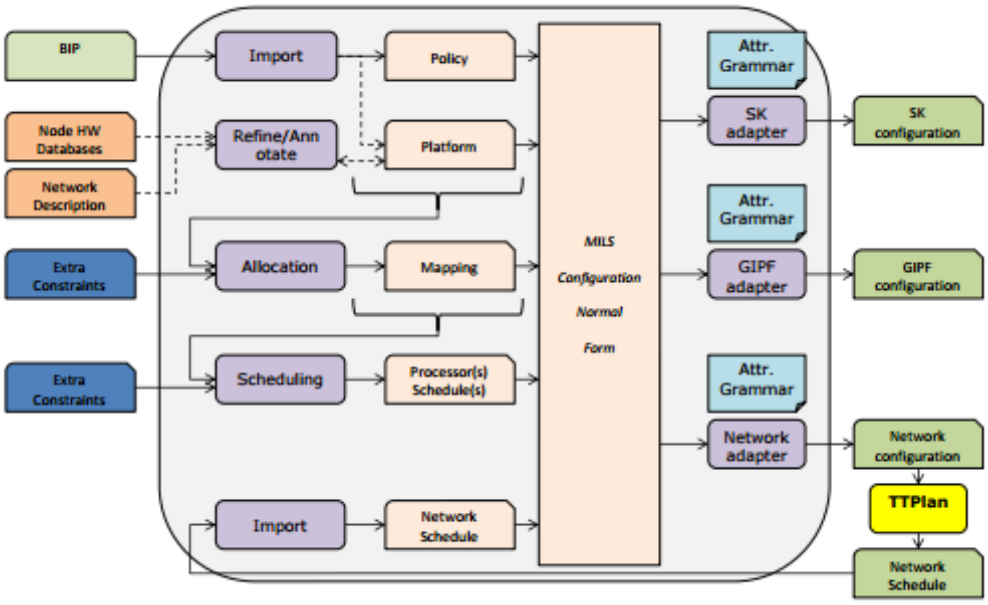

Figure 6.12 – Configuration flow process of the MPCC [**?**].

The system is firstly designed and expressed as a MILS-AADL model [**?**] reflecting a software architecture. When configuring the platform, the MILS-AADL model is translated to a *BIP* model, which abstracts the model and presents it as a set of boxes (components) with some connections between them. This set of interconnected components correspond to the structure to be implemented by the configured platform. The configuration allocates appropriate resources to the different declared components, allowing the execution and communication. When components are located on different nodes these latter require a network configuration.The configuration compiler automatically computes the mapping of components to nodes and generates the configuration.

```
subject implementation comp1.i
    {MPCC: memory(_, [family(ram), size(2)] ) }
```

```
    {MPCC: memory(_, [family(disk), size(10)] ) }
    [...]
end comp1.i
```

Figure 6.12 presents the flow of configuration process. Here we limit our work to the mapping of component's software model to the hardware nodes. First, the configuration compiler begins with a declarative high-level system description expressed in the *BIP* language. The import module defines a policy extractor from *BIP* model into prolog clauses. It is realized and implemented using the *BIP* front-end, where policy-related information were dumped into a policy terms. That is, the policy contains information about the component interfaces, there connections and related annotations if any. The policy representation is structural, expressed in prolog terms which subterms represent the components and the interaction flows between them. Next, according to different annotations on the model (security and safety), a mapping of nodes and virtual links is fully determined. Then, subjects scheduling within nodes as well as the definition and scheduling of virtual links within the network can take place. A fully detailed examples are given in the tech-report [**?**]. Finally, the back-end compiler produces the configuration files for the different separation kernels, components, and for the communication network.

```
system implementation Sys.impl
   {MPCC: deployment(not_same([low,user])) }
   {MPCC: deployment(same([dispatch,user])) }
   subcomponents
     high : subject Hsubject; -- high security network
     low : subject Lsubject; -- low security network
     dispatch: subject Dsubject;
     user: subject Usubject;
   [...]
end Sys.impl;
```

Demonstrative Example

Here we give a simple demonstrative example containing three atomic components a, b and c, communicating between each others through ports. considering the following generated policy from the model:

```
policy_example(4, policy([component(a, subject, [],
            [resource(memory, 6, exclusive),
            resource(cpu, 3, exclusive)], []),
         component(b, subject, [],
            [resource(memory, 5, exclusive),
            resource(cpu, 2, exclusive)], []),
```

```
        component(c, subject, [],
            [resource(memory, 7, exclusive),
            resource(cpu, 2, exclusive)], [])],
        [], [])).
```

Considering the plateform policy containing two nodes $n1$ and $n2$:

```
platform_example(1, platform([device(n1, node, [],
            [resource(memory, 15),
            resource(cpu, 8)], []),
        device(n2, node, [],
            [resource(memory, 15),
            resource(cpu,2)], [])],
        [], []))
```

The generated configuration according to given model policy and plateform is given as follows: two possible solutions are found where, either a and b are executing on the $n1$ node and component c on the node $n2$ or the second solution is to map the components a and c to the $n1$ node and the component b to $n2$ node. These solution are calculated according to the cpu resource needs.

```
?- policy:policy_example(4,Policy), platform:platform_example
    (2,Platform),
     allocate(Policy,Platform,[],Mapping).
Policy = ...
Platform = ...
Mapping = [a-n1, b-n1, c-n2] ;
Policy = ...
Platform = ...
Mapping = [a-n1, b-n2, c-n1] ;
false.
```

However, by adding a constraint on the excution of b and c components to be executed on the same node for instance, no mapping solution is found for the configuration.

```
?- policy:policy_example(4,Policy), platform:platform_example
    (2,Platform),
    allocate(Policy,Platform,[x_same(b,c)],Mapping).
false.
```

6.3 Conclusion

In this chapter, we presented the different implemented tools that defines the approach to track and verify the information flow in the system from model design till implementation and code generation. We also

showed that these tools can be explored by different application domains through the connection and transformations from the set of language factory (MILS-AADL, BPEL, ...) and the *BIP* language. We present in the following chapter an evaluation part, which includes a set of case studies to illustrate with concrete examples the application of the solutions shown in this work, as well as a performance experiments tools.

Chapter 7

Experiments

Contents

7.1 Securing a Web Service Composed System

With the expansion of Web Services (WS) [?] deployed on the enterprise servers, cloud infrastructures and mobile devices, Web Service composition is currently a widely used technique to build complex Internet and enterprise applications. Orchestration languages like *BPEL* [?] allow rapidly developing composed WS by defining a set of activities binding sophisticated services. Nevertheless, advanced security skills and tools are required to ensure critical information security. Indeed, it is important to track data flow and prevent illicit data access by unauthorized services and networks; this task can be challenging when the service is complex or when the composition is hierarchical (the service is composition of composed services and atomic services). For instance, as we presented earlier in Chapter 4, the travel organization WS has to keep a client's destination secret as messages are exchanged between different services like travel agency services and the payment service. Each piece of information depending on the destination, like ticket price, can lead to the secret disclosure if it is not protected. WS security standards [?, ?] provide information flow security solutions for point-to-point inter-service communication but fall short in ensuring end-to-end information flow security in composed services. Furthermore, the *BPEL* language does not state any rules on how to properly apply security mechanisms to services. Generally, developers manually set up their system security configuration parameters which can be tedious and error-prone.

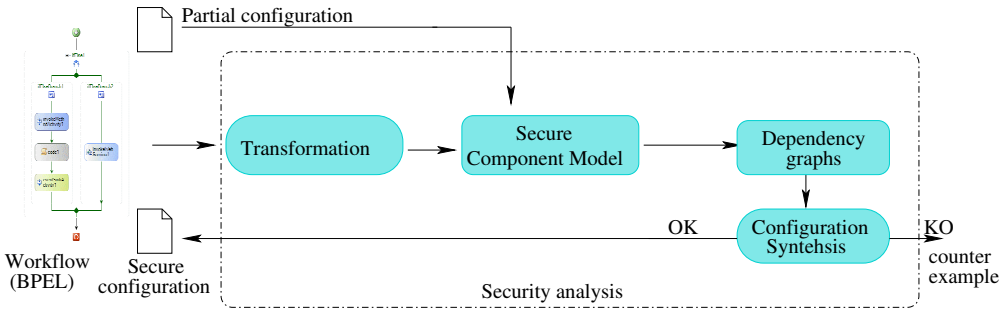

Figure 7.1 – Information flow analysis overview with component-based model

Figure 7.1 shows a workflow overview of an approach used to handle

the information flow security in WS compositions. The service designer describes in *BPEL* his process and defines partial security constraints in a configuration file. The constraints are expressed as authorization rights, that is, a list of services owners and authorized readers for a subset of critical data. The *BPEL* process and the configuration information are then automatically transformed into a component-based framework. This framework was first adapted to abstract distributed WS orchestration to a component-based model where all Web services are transformed into atomic components communicating through interactions by sending and receiving variables and second, to synthesize security configuration for total system variables with respect to security constraints by considering all implicit and explicit data dependencies in the system.

7.1.1 Use-case overview: Smart-Grid System

Here we consider a simplified model of a smart grid system [**?**] managed through Internet network using WS. Smart grid systems usually interconnect a number of cooperating *prosumers*, (that is, *pro*-ducers and con-*sumers*) of electricity on the same shared infrastructure. In principle, every prosumer is able to produce, store and consume energy within the grid. However, its use of the grid has to be negotiated in advance (e.g., on a daily basis) in order to adapt to external conditions (e.g., weather conditions, day-to-day demands,...) as well as to maintain the behaviour of the grid in some optimal parameters (e.g., no peak consumption). Smart grids are subject to requirements related to safety and security e.g., the power consumption / production of a prosumer must remain secret as it actually may reveal sensitive information.

In our WS model of the smart grid, the system consists of a finite number of prosumer processes, Pr_i, communicating with a *smart grid* process, *SMG* . Initially, each Pr_i sends its consumption and production plan, (P_i, C_i, B_i), for the next day to the grid. Production P_i, consumption C_i and (storage) battery B_i are expressed using energy units (integer) where $0 \preceq P_i \preceq 2$, $-3 \preceq C_i \preceq 0$ and $-1 \preceq B_i \preceq 1$. The *SMG* validates the plans received by checking that the overall energy flow through the grid implied by these plans does not exceed the power line capacity. This check measures the consumption exceed acknowledgment, *ack*, compared to a bound, that is, *ack*=0 if the $-1 \preceq \sum_{i=1}^{n}(P_i + C_i + B_i) \preceq 4$, otherwise, it returns the difference between the sum of the plans and the consumption bounds. The *SMG* sends back to each Pr_i an ack_i to negotiate updating its own plan, where $ack = \sum_{i=1}^{n} ack_i$. The negotiation terminates when *ack=0* meaning that the energy flow on the grid does not exceed the line capacity. Figure

7.2 shows the system overview with two prosumers that exchange queries with the smart grid.

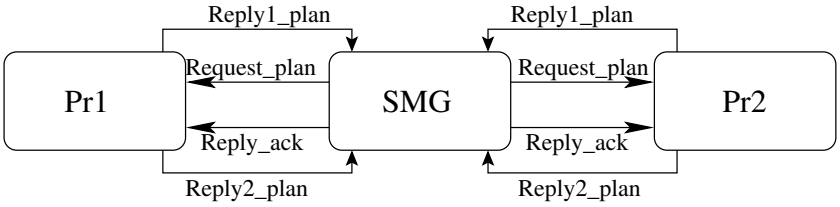

Figure 7.2 – Smart grid application overview

7.1.2 Transformation from BPEL to BIP

Here we give the translation of both *SMG* and *prosumer* components by applying the transformation rules defined in the Chapter 6. The translation of Pr_1 process is given in Figure 7.3 while the translation of the *SMG* process is given in Figure 7.4. The behavior of the atomic components represent the activities given in the *BPEL* processes. First, the prosumer get a request to send his consumption plan to the *SMG* . Here according to an internal threshold, he either send an empty plan or the measured one. The *if* branch in the *BPEL* behavior is translated into a conflicting state where the actions performed from both transitions are different.

In the *SMG* component the parallel executions in the *BPEL* process are translated into sequential transition executions in *BIP* component.

7.1.3 Implementation and Evaluation

The configuration synthesis algorithm described in Chapter 4 is implemented and available for download at `http://www-verimag.imag.fr/~bensaid/secureBIP/`. The user provides the WS composition in *BPEL* and a configuration file (.xml) that contains an *acts_for* relation defining authorities for different processes and the DLM annotations for some process variables. An example of a configuration file is provided in the Appendix. In a first step, the *BPEL* composition is structurally transformed into a component-based model representation in *BIP*[**?**]. The transformation extends an already existing translation of *BPEL* to BIP developed in [**?**] to study functional aspects. In a second step, the synthesis tool takes as input the system model (.bip) and the configuration file (.xml), builds the depen-

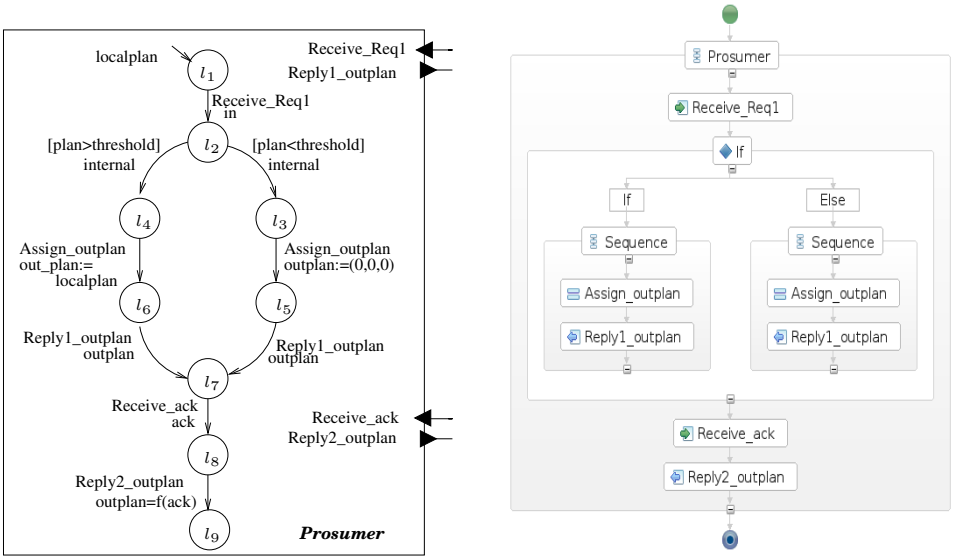

Figure 7.3 – Atomic component representation of a *BPEL* process

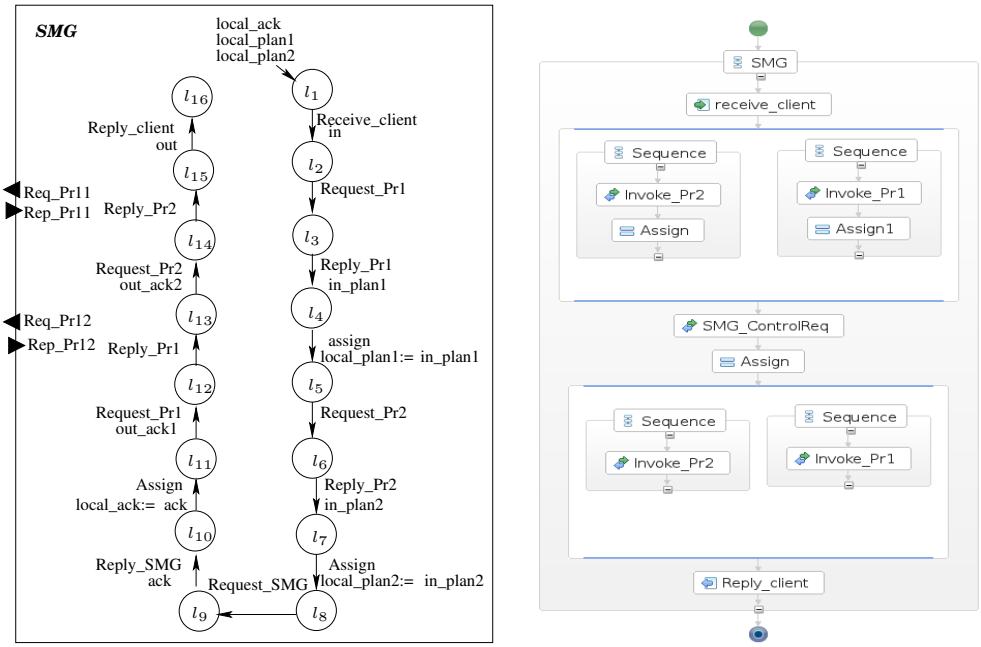

Figure 7.4 – Translation of the *SMG* component

dency graphs of components and runs the synthesis algorithm to produce the complete configuration.

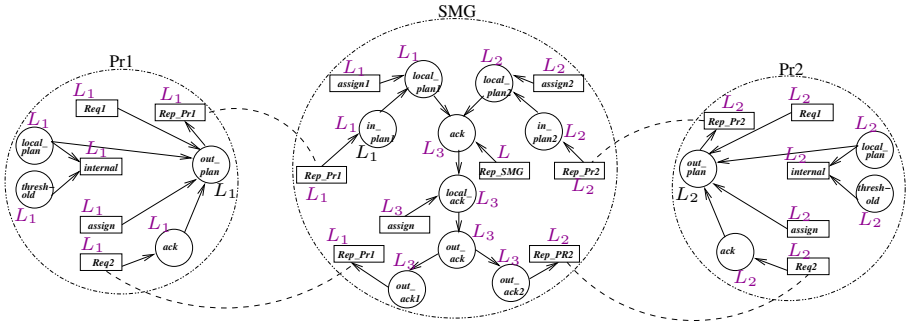

Figure 7.5 – Generated dependency graphs (fragments).

The information flow security requirements that we emphasize here consist first, on ensuring the confidentiality of energy consumption plan for each Pr_i, (which can reveal sensitive competitive information such as its production capacity) and second, ensuring that no prosumer is able to deduce the consumption plan of any other prosumer by observing the received ack information. For instance, consider two prosumers such that one of them, Pr_1, sends an extreme consumption plan $(0, -3, -1)$ to the *SMG* while the second, Pr_2, sends $(0, -3, 0)$ as a consumption plan. The *SMG* first calculates the acknowledgment message that is ack=3 then sends ack_1=1, ack_2=2 messages to respectively Pr_1 and Pr_2. Assume now that Pr_2 sends back a new consumption plan $(1, -2, 1)$ and gets back ack_2=0. By only observing other $ack1$ message sent to Pr_1, the Pr_2 can deduce that the consumption plan of Pr_1 is equal to (0,-3,-1).

The security annotation model we adopted in this part is the DLM (Decentralized Label Model). The designer input configuration file includes an *acts_for* relation as well as some annotated variables. Here we presented an example of a configuration file of the smart grid system. In this xml file we define $\langle authority/\rangle$ to different system components representing the *acts_for* relation. Moreover, we specify by $\langle var\_config/\rangle$ the annotations of variables from different atomic components (processes).

```
<?xml version="1.0"?>
<config>
   <acts_for>
     <authority>SMG: Prosumer1, Prosumer2, Prosumer3</authority
         >
   </acts_for>
   <var_config>
```

```
    <variable var="outplan" process="Prosumer1"
      label="Prosumer1:SMG"></variable>
    <variable var="outplan" process="Prosumer2"
      label="Prosumer2:SMG"></variable>
    <variable var="outplan" process="Prosumer3"
      label="Prosumer3:SMG"></variable>
  </var_config>
</config>
```

For applying our approach to check system security, the designer introduces initially his partial security policy by tagging intuitively some variables that he considers sensitive in system model with security annotations. He also provides an *acts_for* diagram for all model components where he gives authorities to some of them to act for others. In this system the *SMG* component can only acts for both Pr_1 and Pr_2. To ensure confidentiality of prosumers plan, the system administrator annotates *out_plan*1 with $L_1 = \{Pr_1 : SMG\}$ label and *out_plan*2 with $L_2 = \{Pr_2 : SMG\}$ label. Obviously, $L_1 \not\leq L_2$ and $L_2 \not\leq L_1$ are indicating that both prosumers represent separate security domains that can only communicate with the *SMG* component. Then, the tool automatically generates the dependency graph of the transformed smart grid system. Presented in Figure 7.5, the dependency graph is build over ports (rectangles) and data variables (circles) locally at each atomic component(big circles), where arrows intra-circles represent dependencies between ports and data in the same atomic component while arrow inter-circles represent inter-components dependencies. The application of Algorithm 1 to the system dependency graph detects an illegal information flow in the system and generates an error in the *out_plan* node for both prosumers. Indeed, the label propagation in the system creates at *ack* node of the *SMG* component a new label $L_3 = L_1 \sqcup L_2$. Obviously, label $L_3 = \{Pr_1 : SMG\,; Pr_2 : SMG\}$ is more restrictive than both labels L_1 and L_2. Since the *ack* node depends on *out_ack*1 in Pr_1 and *out_ack*2 in Pr_2, then it is labelled with L_3 in both prosumers which causes security level inconsistency at *out_plan* nodes. Algorithm 1 generates an inconsistent security level error between both *out_plan* and *ack* nodes. Here, the system designer has to redefine the initial configuration, for instance, by given more privilege to prosumers to act for *SMG* component and enforce variable *ack* to higher security level $L_3 = \{SMG : SMG\}$. In this case, and with the authority that each prosumer gain, flow can go from L_3 to L_1 and L_2.

7.2 Securing a MILS-AADL Component-Based System

7.2.1 Use-Case overview: Starlight System

We show the usability of this approach by applying it on the Starlight example taken from the literature [?]. The Starlight system was developed by the Australian Defense Science and Technology Organization allowing to establish simultaneous connections to high-level (classified) and low-level networks.

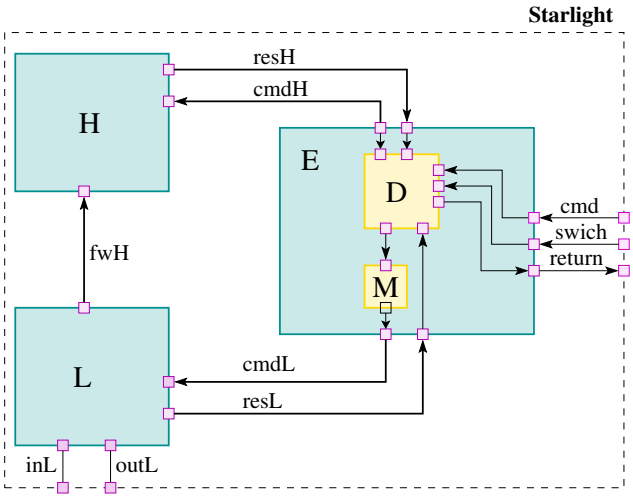

Figure 7.6 – Starlight example overview [?].

As depicted in Figure 7.7 showing the architecture of the Starlight [?], the system contains three atomic components: a high-level server (H), a low-level server (L) each connected to respectively a high and low level network and a switch (E). The switch component is used as a dispatcher that allows to control the user keyboard commands to either flow to the H or the L server. The low-level server can be used either to browse the external world, send messages or have data sent to the high-level server for later use. Based on an internal computation, the D component representing the starlight devise, receives commands from the user and dispatches them to the H or L server. The dispatcher D is extended with a monitor component M that filters commands sent to the L server and stops the communication in case the dispatcher fails. According to the system commands input, a switch_to_low and a switch_to_high events are activated. The prop-

erty that we intend to ensure in this example consists mainly on ensuring that any command sent between a switch_to_high (or the beginning) and a switch_to_low event should not be visible to the low-level subject. Since we consider that the observation of the communications between H and D components cant reveal sensitive information about the sent commands, in this example we don't consider the event non-interference.

7.2.2 Transformation from MILS-AADL to BIP

Figure 7.7 shows a representation of the MILS-AADL starlight model. A full description of the Starlight system in the MILS-AADL language is given in the appendix. The different components (subjects) do communicate with each other through either event data ports (◭) where variables exchange is possible or event ports (⋀). The behavior in the atomic components is specified with modes change that defines a state change at executions.

A complete transformation of the MILS-AADL model to *BIP* language containing six atomic components is given in the appendix. Figure 7.8 shows a simplified representation of the starlight model in *BIP* . Here we only consider three atomic components *L_subject*, *H_subject* and *D_subject*. The *D_subject* receives the input commands from users through *input* port in an integer form, then, according to the received variables (either it is odd or not) the command is routed to one of the servers.

7.2.3 Configuration Generation

As previously mentionned, the security level of events reperesented by the interactions between different components are considered low-level since the observation of the communications between the *D_subject* switcher and the *H_subject* server wan't revial any information related to the command sent by the user for that server. Hence, we concentrate here on the verification of data security by tracking the data information flow. The constraints that we can add to enforce security is that both *L_subject* and *H_subject* are not executed on the same platform node and specially the *H_subject* has to run on a trusted node. A full detailed configurations of the full Starlight model is given in the [**?**].

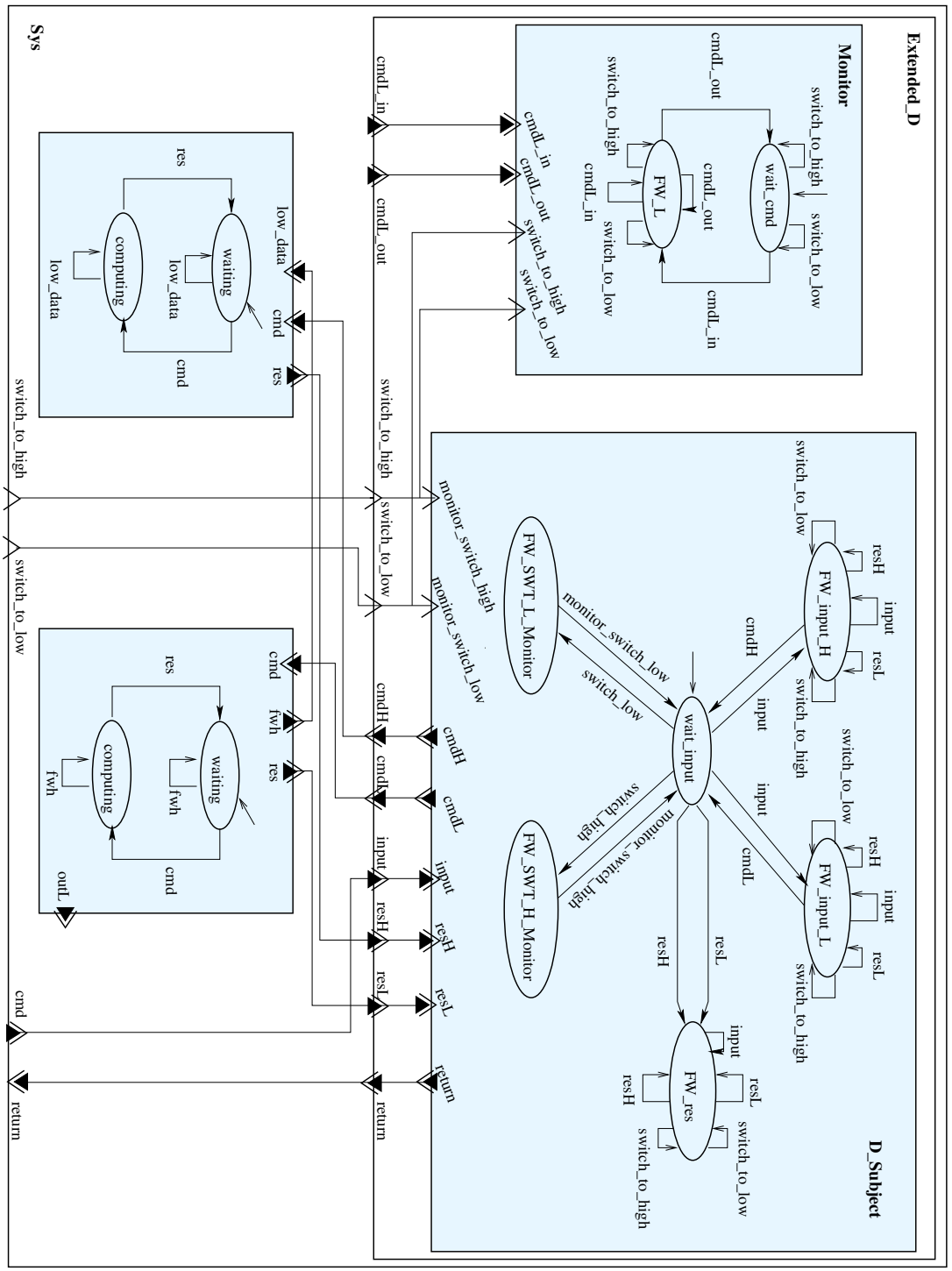

Figure 7.7 – Starlight example overview

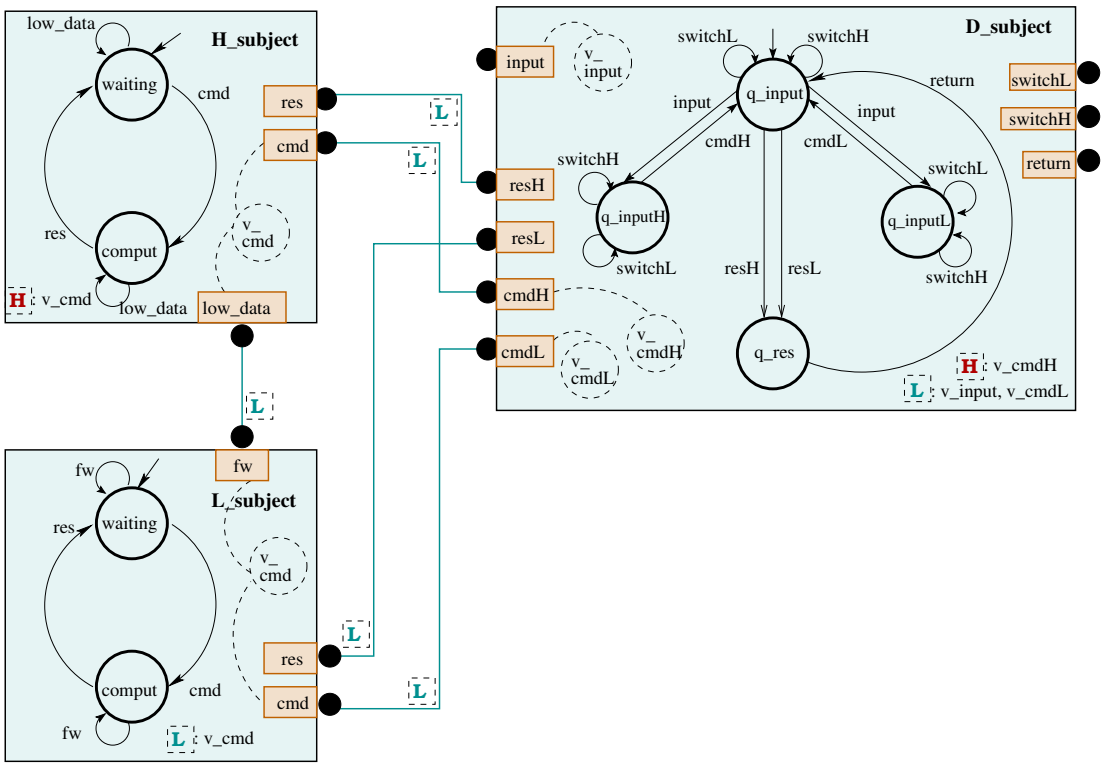

Figure 7.8 – A simplified representation of the translated Starlight system.

Application	n	P	X	σ_0	t
SMG	4	26	24	3	1.82
	13	98	87	12	1.94
	25	194	171	24	2.01
	101	802	703	100	2.82
Whens-App	3	13	6	2	1.13
	13	53	26	12	1.45
	25	101	50	24	1.7
	101	405	204	100	2.11

Table 7.1 – Model size and configuration time (in s) for smart grid application with one initial security label by each prosumer.

7.3 Evaluation

In this chapter, we evaluate the implemented tools and generated code in terms of configuration (annotation) generation and additional cost of the execution. The evaluation is performed on an Intel Code 2Duo 2GHz with 4GB RAM memory running Linux Ubuntu. For generation of the certificates and encryption we use OpenSSL library which contains tested C libraries and here we use X.509 certificates for signature and an asymmetric encryption algorithm (RSA) with 2048bit key size.

The *secureBIP* Tool and the annotation synthesis generator were developed in Java 1.6. The *secureBIP* tool includes a 986 LoC where the secure synthesis tool includes 680 LoC. We extended the code generator first at decentralization to introduce the modification in the generated model and second while adding cryptographic mechanisms to secure interactions between different components.

Compilation and Annotation Generation

A platform dependent configuration is introduced according to the propagated annotation in the distributed model using the configuration file where we specify authentication and encryption mechanisms.

As an evaluation of the compositional approach performance, table 7.1 presents some experiments over configuration time t for different variation of the number of prosumer components, n, in the smart grid system for a given number of variables X, ports P and with initial labels number, σ_0. Here we can notice that our configuration synthesis does not introduce an overhead even by increasing the number of system components.

	Centralized				Decentralized					
Application	**n**	**X**	**P**	σ	**n**	**X**	**P**	σ	γ**(Strd)**	γ**(encry)**
Travel-Reservation	4	20	34	54	19	56	82	138	706	582
Starlight	5	35	29	64	7	40	78	138	1105	1052
SMG	4	24	26	50	6	62	60	122	1013	939
Whens-App	3	6	13	19	11	20	36	56	974	907

Table 7.2 – Model size, configuration and number of executed interactions (without encryption(strd) and encrypted (encry) in 60s for the different applications.

Here, the compilation time depends on the size of the model and the number of variables and ports. However, it is still acceptable, because the generation of the configuration is done only once for the deployment.

Performance of Distributed Implementation

As an evaluation of the decentralization approach performance, table 7.2 presents some experiments over number of interactions executed in 60s of different use-cases presented throughout this thesis. The Table gives information about the number of variables (X), ports (P), components (n) and annotations (σ) in the centralized model and its decentralization. It also shows a comparison between the number of executed interactions in the decentralized model with (encryp) and without (strd) cryptographic mechanisms. The use of encryption method for signing and encrypting messages and sessions between components clearly has an impact on the performance of the application and the number of executed interactions. However, by comparing different applications, the number of encrypted sessions and variables in the *Travel_Reservation* application is the highest compared to the other, hence the number of its encrypted interaction is the lowest in the results. For all the experiments, we are using an asymmetric encryption. The performance of the resulting system can be improved if we choose to use an asymmetric encryption instead of the asymmetric currently used. This optimization is orthogonal to our work.

7.4 Conclusion

In this chapter, we illustrated the implementation and use of the different tools by two examples, which differ in their types, their models oriented components. These examples were used to demonstrate the usefulness and feasibility of our approach and its application to specific cases. However, as useful as it is, any security mechanism drives usually an additional cost in terms of performance which we showed in the experiments in distributed implementation. On the other hand, it is important to show that the tools that we have achieved allow to really ensure non-interference for the entire distributed system.

Chapter 8

Conclusion and Perspectives

In this chapter, we conclude the thesis by describing the main achievements, the future work directions and its perspectives.

Achievements

Building secure software systems is a tedious and complex task to achieve especially when these systems are in large scale. In particular, the verification of non-interference requires an information flow control, which is not always easy for large systems and that can be difficult to apply by developers. As exposed along the manuscript, our belief is that the best way to ensure an end-to-end security in distributed systems is to follow a rigorous approach that ensures the correct and secure building of systems at an abstract level from the early step of defining specifications until implementation and code generation.

Following an MDS (Model Driven Security) approach, we proposed the *secureBIP* , a framework for information flow verification. The security specifications are defined very early in the design phase using *secureBIP* , in parallel to the functional specifications. In this way, security is not introduced in an "ad-hoc" manner at the final phase. In the design phase, the model obtained meet the various requirements of the chosen security policy. Indeed, the system designer intervenes only in the design phase since the various transformations as well as the code generation is done automatically. In this way, the developer does not need to be expert in security, some preliminary knowledge is sufficient.

In this work, we handled two types of non-interference, event and data-

flow non-interference, where we consider that preserving the safety of data flow in a system does not necessarily preserve safe observability equivalence on system's public behavior. Different annotations models are orthogonal to the *secureBIP* framework as presented in state of the art. Sensitive information are tracked in the entire system using annotation defining different security levels. We defined a set of syntactic conditions allowing to automate the verification of non-interference.

We also proposed two tools for transitive non-interference verification, the *secureBIP* and *security synthesis*. The verification is done in practical manner where using the *secureBIP* tool, we annotate the entire variables and ports of the model and then according to the defined set of syntactic constraints, we check the satisfaction of the property. While using the *security synthesis*, we partially annotate the model and then by extracting its compositional dependency graphs we apply a synthesis algorithm that computes the less restrictive secure configuration of the model if it exists. The overall approach is based on propagating annotations on the model following data dependency-graphs and respecting the local security constraints defined with for *secureBIP* model.

The main challenge while implementing distributed software systems is to ensure the end-to-end security. With the complexity increase of such systems, this task may be error prone and demand security skills. The solution we proposed in this thesis is a secure-by-construction approach that consider as input an initially centralized system satisfying the set of security constraints and produces a functionally equivalent distributed model where these security constraints are preserved. The approach consists on automatically transforming the centralized model by splitting the multiparty interactions into binary send/receive interactions based on message passing. The approach is constructive, practical and implementable on distributed platforms, where a program designer has only to focus on specifying a secure centralized multiparty model and then the compiler automatically generates a secure send/receive model.

Following the annotation configuration on the generated decentralized model, we secure the communications and data transfer inter-components by introducing cryptographic atomic components. Hence we generate an intermediate model from which we generate concrete implementation targeting distributed platform architecture. The generated code is considered secure with efficient performance, where only communications and

exchanged variables that need to be secure are encrypted during the model execution.

Perspectives

In the above contributions, several amelioration and extensions are sought to enable covering additional aspects and hence ensure more generality for the proposed approaches.

Information-flow Verification

The verification of non-interference can be extended and investigated in diverse directions.

— The syntactic conditions that we presented in Chapter 4 may be considered restrictive for some systems. Hence, as a future work we consider to further relax them by applying a program dependency graphs based analysis over the compositional model. It is interesting to consider such graphs to handle the transitions dependencies (e.g, causal and conflicting transitions) with consideration of an execution history. Such analysis can provide finer dependencies amongst ports annotations at the conflicting transitions. The use of a runtime verification technique can also dynamically refine dependencies on a runtime-execution history.

— Most of the declassification techniques are not adequate for real systems or are given without rigorous and formal proofs. Thanks to the compositionnality of the present model we can introduce declassification mechanisms to our model. Declassification has been studied for sequential interactive programs with inputs and outputs [**?**], nevertheless, its extension to distributed concurrent component-based models such as Web Services is less understood.

— The transitive non-interference properties that we handle in this thesis may be also considered restrictive especially for interactive systems where the security flow can only go from the less restrictive level to the more restrictive one. The intransitive non-interference is more relaxed version of non-interference that is interesting to analyze using the *secureBIP* framework. According to a specific security policy that defines how information can flow in the system according to a sequence of security levels. Here again using a runtime verification associated a post static analysis phase using program dependency graphs allows to put in place an efficient methode to

verify the intransitive non-interference property. Such analysis can be practical to handle the intransitive non-interference thanks to the compositionality of *secureBIP* model.

Non-interference for real-time systems

The growing importance of real-time systems verification lead us naturally to necessary question of whether verification methodes and proof thechniques developped to handle non-interference in untimed setting can be generalized for timed systems in order to capture not only logical information flow (such is the case with *secureBIP*) but also tiled dependent non-interference (e.g, timing covert channels [**?**]). Previous works [**?**] has reformulated some bisimulation-based definitions in a discrete time settings [**?**], while others [**?**] introduced state-based and trace-based non-interference using timed automata.

We consider that the *secureBIP* , as an extension for the BIP framework, represents a complete flateform that isadequate to handle such issues. Indeed, the BIP framework provides a real-time semantics for BIP models [**?**] where each interactions are annotated with timing constraints expressed using as range of times defined over clocks and urgencies. The ranges of time indicates when an interaction can be executed while the urgency indicates whether the model authorizes the time to progress before executing the interaction. The behavior of a timed BIP framework is described with timed automata. Hence the the analysis of timed non-interference can be reformulated in compositional manner.

Besides, ensuring security in destributed real time systems represents a huge challenge where before scheduling an interaction once has to ensure that no conflicting interaction with an earlier deadline is enabled. Hence, Hence, based on an extension of the timed-BIP framework we can assure a combination of distribution, time and security constraints in single framework which can be intresting in efficiently building complexe destributed systems.

List of Figures

List of Tables

.1 Starlight Use-case

.1.1 MILS-AADL Original Model

```
-- MILS-AADL Model of Starlight

-- Property to ensure:
--     any command sent between a switch_to_high (or the
    beginning)
--     and a switch_to_low event should not be visible to the
--     low-level subject.

-- Overall MILS-AADL Model of Starlight

constants
  computation: function int -> int;
  is_high: function int -> bool;

system Sys
  features
    cmd: in event data port int;
    switch_to_high: in event port;
    switch_to_low: in event port;
    return: out event data port int;
    outL: out data port int;

end Sys;

system implementation Sys.impl
  subcomponents
    E : subject Extended_D;
    High : subject Hsubject; -- high security network
    Low : subject Lsubject; -- low security network
  connections
    event data port cmd -> E.input;
    event port switch_to_high -> E.switch_to_high;
    event port switch_to_low -> E.switch_to_low;
    event data port E.return -> return;
    event data port E.cmdL -> Low.cmd;
    event data port E.cmdH -> High.cmd;
    event data port Low.res -> E.resL;
    event data port High.res -> E.resH;
    event data port Low.fwH -> High.low_data;
```

```
    data port Low.outL -> outL;

end Sys.impl;

-------------------------------------------------------
-- Subjects descriptions
-------------------------------------------------------

subject Extended_D
  features
    switch_to_high: in event port;
    switch_to_low:  in event port;
    input:    in event data port int;
    return: out event data port int;
    cmdH:   out event data port int;
    cmdL:   out event data port int;
    resH:    in event data port int;
    resL:    in event data port int;

end Extended_D;

subject implementation Extended_D.i
  subcomponents
    D: thread Dsubject;
    M: thread Monitor;
  connections
    event port switch_to_high -> D.switch_to_high;
    event port D.monitor_switch_to_high -> M.switch_to_high;

    event port switch_to_low -> D.switch_to_low;
    event port D.monitor_switch_to_low -> M.switch_to_low;

    event data port input -> D.input;
    event data port D.return -> return;
    event data port D.cmdH -> cmdH;
    event data port D.cmdL -> M.cmdL_in;
    event data port M.cmdL_out -> cmdL;
    event data port resL -> D.resL;
    event data port resH -> D.resH;

end Extended_D.i;


-- D:
thread Dsubject
  features
    switch_to_high: in event port;    -- from user
    switch_to_low: in event port;     -- from user
    monitor_switch_to_high: out event port;     -- to monitor
```

```
    monitor_switch_to_low:  out event port;    -- to monitor
    input:    in event data port int;  -- cmd input from user
    return: out event data port int;  -- to user
    cmdH:    out event data port int;  -- to High subject
    cmdL:    out event data port int;  -- to Low subject
    resH:     in event data port int;  -- from High subject
    resL:     in event data port int;  -- from Low subject

end Dsubject;

thread implementation Dsubject.impl
  subcomponents
    i: data int default 0;       -- storing the last input from
       the user
    r: data int default 0;       -- storing the next result for
       the user
    mode_high: data bool default true; -- if false the current
       mode is low, otherwise high
    clk : data clock;
  modes
    wait_input: initial mode;
    FW_INPUT_H: mode while clk <= 1;  -- received input to
       forward
    FW_INPUT_L: mode while clk <= 1;  -- received input to
       forward
    FW_SWT_H_MONITOR: mode while clk <= 1;  -- forward switch
       to monitor
    FW_SWT_L_MONITOR: mode while clk <= 1;  -- forward switch
       to monitor
    FW_RES: mode while clk <= 1;  -- received result to forward
  transitions
     -- switch between security modes
    wait_input -[switch_to_high then mode_high := true ]->
       FW_SWT_H_MONITOR;
    wait_input -[switch_to_low  then mode_high := false]->
       FW_SWT_L_MONITOR;
    -- forward switch to monito
    FW_SWT_L_MONITOR -[monitor_switch_to_low  ]-> wait_input;
    FW_SWT_H_MONITOR -[monitor_switch_to_high ]-> wait_input;
    -- otherwise, forwards the command (through cmdH)
    wait_input -[input when mode_high = true  then clk := 0; i
       := data(input)]-> FW_INPUT_H;
    wait_input -[input when mode_high = false then clk := 0; i
       := data(input)]-> FW_INPUT_L;
    -- forward incoming results to the user
    wait_input -[resH then clk := 0; r := data(resH)]-> FW_RES;
    wait_input -[resL then clk := 0; r := data(resL)]-> FW_RES;
    wait_input -[]-> wait_input;
```

```
    -- forward commands
    FW_INPUT_H -[cmdH(i)]-> wait_input;
    FW_INPUT_H -[]-> FW_INPUT_H;
    -- FW_INPUT_H -[switch_to_low  ]-> FW_INPUT_H;
    -- FW_INPUT_H -[switch_to_high ]-> FW_INPUT_H;
    -- FW_INPUT_H -[resH ]-> FW_INPUT_H;
    -- FW_INPUT_H -[resL ]-> FW_INPUT_H;
    -- FW_INPUT_H -[input]-> FW_INPUT_H;

    FW_INPUT_L -[cmdL(i)]-> wait_input;
    FW_INPUT_L -[]-> FW_INPUT_L;
    -- FW_INPUT_L -[switch_to_low  ]-> FW_INPUT_L;
    -- FW_INPUT_L -[switch_to_high ]-> FW_INPUT_L;
    -- FW_INPUT_L -[resH ]-> FW_INPUT_L;
    -- FW_INPUT_L -[resL ]-> FW_INPUT_L;
    -- FW_INPUT_L -[input]-> FW_INPUT_L;

    FW_RES -[return(r)]-> wait_input;
    FW_RES -[]-> FW_RES;
    -- FW_RES -[switch_to_low  ]-> FW_RES;
    -- FW_RES -[switch_to_high ]-> FW_RES;
    -- FW_RES -[resH ]-> FW_RES;
    -- FW_RES -[resL ]-> FW_RES;
    -- FW_RES -[input]-> FW_RES;

end Dsubject.impl;

thread Monitor
  features
    switch_to_high: in event port;
    switch_to_low: in event port;
    cmdL_in: in event data port int;
    cmdL_out: out event data port int;

end Monitor;

thread implementation Monitor.i
  subcomponents
    i: data int default 0;
    mode_high: data bool default true; -- if false the current
        mode is low, otherwise high
    clk: data clock;
  modes
    wait_cmd: initial mode;
    FW_L: mode while clk <= 1;
  transitions
    wait_cmd -[switch_to_low  then mode_high := false]->
        wait_cmd;
```

```
    wait_cmd -[switch_to_high then mode_high := true ]->
        wait_cmd;
    wait_cmd -[cmdL_in when mode_high = false then clk := 0; i
        := data(cmdL_in)]-> FW_L;
    wait_cmd -[cmdL_in when mode_high = true]-> wait_cmd;
    wait_cmd -[]-> wait_cmd;

    FW_L -[cmdL_out(i)]-> wait_cmd;
    FW_L -[]-> FW_L;
    -- ignore cmds during operation
    -- FW_L -[switch_to_low  ]-> FW_L;
    -- FW_L -[switch_to_high ]-> FW_L;
    -- FW_L -[cmdL_in]-> FW_L;

end Monitor.i;

-- end extension --


--  low-level subject
subject Lsubject
  features
    cmd: in event data port int;
    res: out event data port int;
    outL: out data port int default 0; -- 0 is assumed as ok as
        init
    fwH: out event data port int;

end Lsubject;

subject implementation Lsubject.impl
  subcomponents
    i: data int default 0;
    clk: data clock;
  modes
    WAITING: initial mode;
    COMPUTING: mode while clk <= 1;
  transitions
    WAITING -[cmd then clk :=0; outL := data(cmd); i := data(
        cmd)]-> COMPUTING;
    WAITING -[fwH]-> WAITING;
    WAITING -[]-> WAITING;

    COMPUTING -[res(computation(i))] -> WAITING;
    COMPUTING -[fwH]-> COMPUTING;
    COMPUTING -[]-> COMPUTING;
    -- COMPUTING -[cmd]-> COMPUTING;

end Lsubject.impl;
```

```
-- high-level subject
subject Hsubject
  features
    cmd: in event data port int;
    res: out event data port int;
    low_data: in event data port int;

end Hsubject;

subject implementation Hsubject.impl
  subcomponents
    i: data int default 0;
    clk: data clock;
  modes
    WAITING: initial mode;
    COMPUTING: mode while clk <= 1;
  transitions
    WAITING    -[cmd then clk := 0; i := data(cmd)]-> COMPUTING;
    WAITING    -[low_data]-> WAITING;
    WAITING    -[]-> WAITING;
    COMPUTING -[res(computation(i))]-> WAITING;
    COMPUTING -[low_data]-> COMPUTING;
    COMPUTING -[]-> COMPUTING;

end Hsubject.impl;
```

.1.2 Translated Starlight Model into BIP

```
package starlight3
 // mode access/identification port type
    port type mode_access()
 // in/out event port types & related connectors

    port type in_event()
    port type out_event()

 connector type out_in
     (out_event out, in_event in)
       define out in
       on out in provided (true)
   end

 connector type out_in_mode
     (out_event out, in_event in, mode_access mode)
```

```
      define out in mode
      on out in mode provided (true)
end

connector type in_mode
    (in_event in, mode_access mode)
      export port in_event _in()
      define in mode
      on in mode provided (true)
end

connector type in_in_mode
    (in_event in1, in_event in2, mode_access mode)
           export port in_event _in()
           define in1 in2 mode
           on in1 in2 mode provided (true)
end


connector type out_mode
    (out_event out, mode_access mode)
      export port out_event _out()
      define out mode
      on out mode provided (true)
end

// act/deact event port types

   port type act_event()
   port type deact_event()
connector type act_mode
    (act_event act, mode_access mode)
      define act mode
      on act mode provided (true)
end

connector type deact_mode
    (deact_event deact, mode_access mode)
      define deact mode
      on deact mode provided (true)
end


/*
 * (Event_)Data_Port types and connectors
 *
 */

port type in_data_int ( int value )
```

```
port type out_data_int ( int value )

connector type out_in_int
   (out_data_int out, in_data_int in)
            define out in
            on out in
                 provided (true)
                 up { }
                 down { in.value = out.value; }
end

connector type out_in_int_mode
   (out_data_int out, in_data_int in, mode_access mode)
      define out in mode
      on out in mode
           provided (true)
           up { }
           down { in.value = out.value; }
end

connector type in_int_mode
   (in_data_int in, mode_access mode)
      data int value
      export port in_data_int _in(value)
      define in mode
      on in mode
           provided (true)
           up { }
           down { in.value = value; }
end

connector type out_int_mode
  (out_data_int out, mode_access mode)
      data int value
      export port out_data_int _out(value)
      define out mode
      on out mode
           provided (true)
           up { value = out.value; }
           down { }
end


/*
 * Dsubject_impl
 *
 */

atom type Dsubject_impl()
```

```
data int bip_active


data int i

 data int mode_high


data int r
data int v_input
data int v_resH
data int v_resL
data int v_cmdL
data int v_cmdH
data int v_return

// clock clk

// event \ data ports
export port in_data_int input(v_input)
export port in_data_int resH(v_resH)
export port in_data_int resL(v_resL)
export port out_data_int cmdL(v_cmdL)
export port out_data_int cmdH(v_cmdH)
export port out_data_int return(v_return)
export port in_event switch_to_high()
export port in_event switch_to_low()




//  inner trigger ports

//  mode identification ports
export port mode_access bip_run()

export port mode_access wait_input()
export port mode_access FW_INPUT_H()
export port mode_access FW_RES()
export port mode_access FW_INPUT_L()

// modes
place  q_wait_input,q_FW_INPUT_H,q_FW_RES,q_FW_INPUT_L

// initialization
initial to  q_wait_input do { bip_active = true; i = 0; r = 0;
    mode_high = true; }
```

```
 // data flow transitions

 // mode transitions
 on switch_to_high from q_wait_input to q_wait_input
    provided ( bip_active ) do {mode_high = true;  }
 on switch_to_low from q_wait_input to q_wait_input
    provided ( bip_active ) do {mode_high = false;  }

 @SuppressWarning(nondeterminism)
 on input from q_wait_input to q_FW_INPUT_H
   provided ( bip_active && mode_high == true ) do { i =
       v_input;  }
 @SuppressWarning(nondeterminism)
 on input from q_wait_input to q_FW_INPUT_L
   provided ( bip_active && mode_high == false ) do { i =
       v_input;  }

 on resH from q_wait_input to q_FW_RES
    provided ( bip_active ) do { r = v_resH;  }
 on resL from q_wait_input to q_FW_RES
    provided ( bip_active ) do { r = v_resL;  }
 on cmdH from q_FW_INPUT_H to q_wait_input
    provided ( bip_active )
 on switch_to_low from q_FW_INPUT_H to q_FW_INPUT_H
    provided ( bip_active ) do {mode_high = false;  }
 on switch_to_high from q_FW_INPUT_H to q_FW_INPUT_H
    provided ( bip_active ) do {mode_high = true;  }
 on cmdL from q_FW_INPUT_L to q_wait_input
    provided ( bip_active )
 on switch_to_low from q_FW_INPUT_L to q_FW_INPUT_L
    provided ( bip_active ) do {mode_high = false;  }
 on switch_to_high from q_FW_INPUT_L to q_FW_INPUT_L
    provided ( bip_active ) do {mode_high = true;  }
 on return from q_FW_RES to q_wait_input
    provided ( bip_active )
 on switch_to_low from q_FW_RES to q_FW_RES
    provided ( bip_active ) do {mode_high = false;  }
 on switch_to_high from q_FW_RES to q_FW_RES
    provided ( bip_active ) do {mode_high = true;  }

 // data ports activation transitions


end

/*
 * Lsubject_impl
 *
```

```
*/

atom type Lsubject_impl()


 data int bip_active

// data
data int i
data int v_outL
data int v_cmd
data int v_res
data int v_fwH

// clock clk

// event \ data ports
export port out_data_int outL(v_outL)
export port in_data_int cmd(v_cmd)
export port out_data_int res(v_res)
export port out_data_int fwH(v_fwH)




//  inner trigger ports

//  mode identification ports
export port mode_access bip_run()

export port mode_access WAITING()
export port mode_access COMPUTING()

// modes
place  q_WAITING,q_COMPUTING

// initialization
initial to q_WAITING do { bip_active = true; i = 0; }

// data flow transitions

// mode transitions
on cmd from q_WAITING to q_COMPUTING
   provided ( bip_active ) do { v_outL = v_cmd; i = v_cmd;  }



on fwH from q_WAITING to q_WAITING
   provided ( bip_active )
```

```
on res from q_COMPUTING to q_WAITING
    provided ( bip_active )
on fwH from q_COMPUTING to q_COMPUTING
    provided ( bip_active )

on bip_run from q_WAITING to q_WAITING

on bip_run from q_COMPUTING to q_COMPUTING



end

/*
 * Extended_Dispatch_i_controller
 *
 */

atom type Extended_Dispatch_i_controller()


 data int bip_active


// data

// event \ data ports


//  inner trigger ports

//  mode identification ports
export port mode_access bip_run()


// modes
place q_bip_run

// initialization
initial to q_bip_run do { bip_active = true; }

// data flow transitions

// mode transitions

// data ports activation transitions


// mode identification transitions
```

```
on bip_run from q_bip_run to q_bip_run



end

/*
 * Hsubject_impl
 *
 */

atom type Hsubject_impl()


 data int bip_active

// data
data int i
data int v_cmd
data int v_low_data
data int v_res

// clock clk

// event \ data ports
export port in_data_int cmd(v_cmd)
export port in_data_int low_data(v_low_data)
export port out_data_int res(v_res)



//  inner trigger ports

//  mode identification ports
export port mode_access bip_run()

export port mode_access WAITING()
export port mode_access COMPUTING()

// modes
place  q_WAITING,q_COMPUTING

// initialization
initial to  q_WAITING do { bip_active = true; i = 0; }

// data flow transitions

// mode transitions
on cmd from q_WAITING to q_COMPUTING
```

```
    provided ( bip_active ) do { i = v_cmd;  }
 on low_data from q_WAITING to q_WAITING
    provided ( bip_active )
 on res from q_COMPUTING to q_WAITING
    provided ( bip_active )
 on low_data from q_COMPUTING to q_COMPUTING
    provided ( bip_active )




 end

/*
 * starlight_impl_controller
 *
 */

 atom type starlight_impl_controller()


 data int bip_active



 //  inner trigger ports

 //  mode identification ports
 export port mode_access bip_run()


 // modes
 place q_bip_run, q_bip_suspended

 // initialization
 initial to q_bip_run do { bip_active = true; }

 // data flow transitions

 // mode transitions

 // data ports activation transitions

 // mode identification transitions
 on bip_run from q_bip_run to q_bip_run



 end
```

```
/*
 * Monitor_impl
 *
 */

atom type Monitor_impl()


 data int bip_active

// data
data int i

 data int mode_high

data int v_cmdL_in
data int v_cmdL_out

// clock clk

// event \ data ports
export port in_data_int cmdL_in(v_cmdL_in)
export port out_data_int cmdL_out(v_cmdL_out)
export port in_event switch_to_high()
export port in_event switch_to_low()



//  inner trigger ports

//  mode identification ports
export port mode_access bip_run()
export port mode_access bip_suspend()
export port mode_access FW_L()
export port mode_access wait_cmd()

// modes
place  q_FW_L, q_wait_cmd

// initialization
initial to  q_wait_cmd do { bip_active = true; i = 0;
    mode_high = true; }

// data flow transitions

// mode transitions
on switch_to_low from q_wait_cmd to q_wait_cmd
   provided ( bip_active ) do {mode_high = false;  }
```

```
 on switch_to_high from q_wait_cmd to q_wait_cmd
    provided ( bip_active ) do {mode_high = true;  }

 @SuppressWarning(nondeterminism)
 on cmdL_in from q_wait_cmd to q_FW_L
   provided ( bip_active && mode_high == false ) do { i =
       v_cmdL_in;  }
 @SuppressWarning(nondeterminism)
 on cmdL_in from q_wait_cmd to q_wait_cmd
   provided ( bip_active && mode_high == true )

 on cmdL_out from q_FW_L to q_wait_cmd
    provided ( bip_active )
 on switch_to_low from q_FW_L to q_FW_L
    provided ( bip_active ) do {mode_high = false;  }
 on switch_to_high from q_FW_L to q_FW_L
    provided ( bip_active ) do {mode_high = true;  }

 // data ports activation transitions


 end



connector type in_in_mode_mode
     (in_event in1, in_event in2, mode_access mode1,
        mode_access mode2)
      define in1 in2 mode1 mode2
end



connector type in_int_mode_mode
     (in_data_int in, mode_access mode, mode_access mode2)
     data int value
     define in mode mode2
        on in mode mode2
           provided (true)
           up { }
           down { in.value = value; }
end


 connector type out_int_mode_mode
   (out_data_int out, mode_access mode, mode_access mode2)
       data int value
       define out mode mode2
       on out mode mode2
```

```
            provided (true)
            // up { value = out.value; }
            // down { }
end

connector type out_in_int_mode_mode
    (out_data_int out, in_data_int in, mode_access mode,
        mode_access mode2)
        define out in mode mode2
        on out in mode mode2
            provided (true)
            up { }
            down { in.value = out.value; }
end



compound type main()

    component Extended_Dispatch_i_controller
        dispatch_i_controller()
    component starlight_impl_controller starlight_controller()

    component Monitor_impl monitor()
    component Dsubject_impl subDispatch()


// mode controller


 // subcomponents
    component Lsubject_impl low_net()
    component Hsubject_impl high_net()


// data flows
   connector out_int_mode
        x_low_net_outL_controller_bip_run(low_net.outL,
            starlight_controller.bip_run)


 // event connections
 connector in_in_mode_mode
        x_dispatch_switch_to_high_controller_bip_run
      (subDispatch.switch_to_high, monitor.switch_to_high,
          dispatch_i_controller.bip_run, starlight_controller.
          bip_run)

 connector in_in_mode_mode
```

```
        x_dispatch_switch_to_low_controller_bip_run
    (subDispatch.switch_to_low, monitor.switch_to_low,
        dispatch_i_controller.bip_run, starlight_controller.
        bip_run)



// event data connections

connector in_int_mode_mode
        x_dispatch_input_controller_bip_run(subDispatch.input,
            dispatch_i_controller.bip_run, starlight_controller.
            bip_run)

connector out_int_mode_mode
        x_dispatch_return_controller_bip_run(subDispatch.return
            , dispatch_i_controller.bip_run,
            starlight_controller.bip_run)

connector out_in_int_mode_mode
        x_dispatch_cmdL_low_net_cmd_controller_bip_run(monitor.
            cmdL_out, low_net.cmd, dispatch_i_controller.
            bip_run , starlight_controller.bip_run)

connector out_in_int_mode_mode
        x_dispatch_cmdH_high_net_cmd_controller_bip_run(
            subDispatch.cmdH, high_net.cmd ,
            dispatch_i_controller.bip_run, starlight_controller.
            bip_run)

connector out_in_int_mode_mode
        x_low_net_res_dispatch_resL_controller_bip_run(low_net.
            res, subDispatch.resL, dispatch_i_controller.bip_run
             , starlight_controller.bip_run)


connector out_in_int_mode_mode
        x_high_net_res_dispatch_resH_controller_bip_run(
            high_net.res, subDispatch.resH,
            dispatch_i_controller.bip_run, starlight_controller.
            bip_run)

connector out_in_int_mode
        x_low_net_fwH_high_net_low_data_controller_bip_run(
            low_net.fwH, high_net.low_data, starlight_controller
            .bip_run)


connector out_in_int_mode
```

```
        x_subDispatch_cmdL_monitor_cmdL_in_controller_bip_run(
            subDispatch.cmdL, monitor.cmdL_in,
            dispatch_i_controller.bip_run)

end
end
```

Bibliography

[1] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Model-driven Information Flow Security for Component-Based Systems. In *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014*, pages 1–20, 2014.

[2] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. A robust framework for securing composed web services. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 105–122, 2015.

[3] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. Building secure-by-construction distributed component-based systems. Technical Report TR-2014-6, Verimag Research Report, 2014.

[4] Jianjun Shen, Sihan Qing, Qingni Shen, and Liping Li. Covert channel identification founded on information flow analysis. In *Computational Intelligence and Security (CIS'05)*, volume 3802 of *LNCS*, pages 381–387. Springer, 2005.

[5] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[6] Fredrik. Seehusen and Ketil Stølen. *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, chapter A Method for Model-driven Information Flow Security, pages 199–229. IGI Global, 2012.

[7] Takoua Abdellatif, Lilia Sfaxi, Riadh Robbana, and Yassine Lakhnech. Automating information flow control in component-based distributed systems. In *14th International ACM Sigsoft Symposium on Component Based Software Engineering (CBSE'11)*, pages 73–82. ACM, 2011.

[8] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, pages 504–513, 1977.

[9] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and Multics interpretation, 1976.

[10] J. K. Biba. Integrity considerations for secure computer systems, 1977.

[11] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. *SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[12] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 293–308. USENIX Association, 2008.

[13] P Efstathopoulos, M Krohn, S Vandebogart, C Frey, D Ziegler, E Kohler, D Mazières, F Kaashoek, and R Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17, 2005.

[14] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9, 2000.

[15] Jean Quilbeuf. *Distributed Implementations of Component-based Systems with Prioritized Multiparty Interactions. Application to the BIP Framework.* Theses, Université de Grenoble, September 2013.

[16] Mohamad Jaber. *Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP.* Theses, Université Joseph-Fourier - Grenoble I, October 2010.

[17] Ananda Basu, Philippe Bidinger, Marius Bozga, and Joseph Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems - FORTE*.

[18] K. Mani Chandy. *Parallel Program Design: A Foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[19] Rajive Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Trans. Software Eng.*, 15:1053–1065, 1989.

[20] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, pages 632–646, 1984.

[21] Rajive Bagrodia. A distributed algorithm to implement n-party rendevouz. In *Proceedings of the Seventh Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 138–152, London, UK, UK, 1987. Springer-Verlag.

[22] Chandy K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.

[23] Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 2012.

[24] Heiko Mantel. *A uniform framework for the formal specification and verification of information flow security.* PhD thesis, Saarland University, 2004.

[25] D. Sutherland, editor. *Proceedings of the 9th Natinal Computer Security Conference*. ACM, 1986.

[26] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 94–, Washington, DC, USA, 1997. IEEE Computer Society.

[27] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, 1990.

[28] Jeremy Jacob. Security specifications. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 18-21, 1988*, pages 14–23, 1988.

[29] Dale M. Johnson and F. Javier Thayer. Security and the composition of machines. In *First IEEE Computer Security Foundations Workshop - CSFW'88, Franconia, New Hampshire, USA, June 12-15, 1988, Proceedings*, pages 72–89, 1988.

[30] Joshua D. Guttman and Mark E. Nadel. What needs securing. In *First IEEE Computer Security Foundations Workshop - CSFW'88, Franconia, New Hampshire, USA, June 12-15, 1988, Proceedings*, pages 34–57, 1988.

[31] John McLean. Security models and information flow. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*, pages 180–189, 1990.

[32] Colin O'Halloran. A calculus of information flow. In *ESORICS 90 - First European Symposium on Research in Computer Security, October 24-26, 1990, Toulouse, France*, pages 147–159, 1990.

[33] Pierre Bieber and Frédéric Cuppens. A logical view of secure dependencies. *Journal of Computer Security*, 1(1):99–130, 1992.

[34] Ira S. Moskowitz and Oliver Costich. A classical automata approach to noninterference type problems. In *5th IEEE Computer Security Foundations Workshop - CSFW'92, Franconia, New Hampshire, USA, June 16-18, 1992, Proceedings*, pages 2–8, 1992.

[35] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, SRI International, 1992.

[36] Riccardo Focardi and Roberto Gorrieri. Classification of Security Properties (Part I: Information Flow). In *Revised lectures of IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design (FOSAD'00)*, volume 2171 of *LNCS*, pages 331–396. Springer, 2001.

[37] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, pages 114–127, 1995.

[38] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Security and Privacy (SP'97)*, pages 94–102. IEEE Computer Society, 1997.

[39] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW 1999, Mordano, Italy, June 28-30, 1999*, pages 228–238, 1999.

[40] Riccardo Focardi and Sabina Rossi. Information flow security in dynamic contexts. *Journal of Computer Security*, 14(1):65–110, 2006.

[41] N. Zeldovich and Stanford University. Computer Science Dept. *Securing untrustworthy software using information flow control*. 2007.

[42] Bowen Alpern and Fred B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[43] McCullough Darl. Specifications for multi-level security and a hook-up. In *Security and Privacy, 1987 IEEE Symposium on*, 1987.

[44] Guttman Joshua and M. Nadel. What needs securing? In *Proceedings of the IEEE Computer Security Foundations Workshop*, 1987.

[45] Jonathan K. Millen. Unwinding forward correctability. *J. Comput. Secur.*, 3(1):35–54, January 1995.

[46] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974.

[47] G. Scott Graham and Peter J. Denning. Protection: Principles and practice. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[48] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, August 1976.

[49] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[50] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *17th IEEE workshop on Computer Security Foundations (CSFW'04)*, pages 172–, 2004.

[51] M Krohn, A Yip, M Brodsky, N Cliffer, M F Kaashoek, E Kohler, and R Morris. Information flow control for standard OS abstractions. *Proceedings of twentyfirst ACM SIGOPS symposium on Operating systems principles SOSP 07*, 41(6):321, 2007.

[52] Simone Fischer-Hübner. *IT-security and Privacy: Design and Use of Privacy-enhancing Security Mechanisms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[53] N Zeldovich, S Boyd-Wickizer, E Kohler, and D Mazi. Making Information Flow Explicit in HiStar. *Network*, pages:263–278, 2006.

[54] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: from uml models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15:39–91, 2006.

[55] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Inf. Comput.*, 206(2-4):378–401, 2008.

[56] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.

[57] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *16th ACM conference on Computer and Communications Security (CCS'09)*, pages 432–441. ACM, 2009.

[58] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 321–334, New York, NY, USA, 2009. ACM.

[59] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based design using the BIP framework. *IEEE Software, Special Edition – Software Components beyond Programming – from Routines to Services*, 28(3):41–48, 2011.

[60] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. IEEE Computer Society Press, 2006.

[61] Rafael Accorsi and Andreas Lehmann. Automatic information flow analysis of business process models. In *10th International Conference on Business Process Management (BPM'12)*, volume 7481 of *LNCS*, pages 172–187. Springer, 2012.

[62] Simone Frau, Roberto Gorrieri, and Carlo Ferigato. Petri net security checker: Structural non-interference at work. In *Formal Aspects in Security and Trust, 5th International Workshop (FAST'08), Revised Lectures*, volume 5491 of *LNCS*, pages 210–225. Springer, 2009.

[63] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. pages 26–60, New York, NY, USA, January 1990. ACM.

[64] David Binkley. Computing amorphous program slices using dependence graphs. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 519–525, New York, NY, USA, 1999. ACM.

[65] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating fast code from concurrent program dependence graphs. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, New York, NY, USA, 2004. ACM.

[66] W. Baxter and H. R. Bauer, III. The program dependence graph and vectorization. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 1–11, New York, NY, USA, 1989. ACM.

[67] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. volume 9, pages 177–184, New York, NY, USA, April 1984. ACM.

[68] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 384–396, New York, NY, USA, 1993. ACM.

[69] Dieter Hutter and Melanie Volkamer. Information flow control to secure dynamic web service composition. In *Security in Pervasive Computing (SPC'06)*, volume 3934 of *LNCS*, pages 196–210. Springer, 2006.

[70] `http://www.cs.cornell.edu/jif/`.

[71] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. *SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[72] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition.* Packt Publishing 2006, 2006.

[73] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1.* IBM, 2003.

[74] Specification of mils-aadl. Technical Report D2.1, Version 2.0, D-MILS Project, July 2014.

[75] Emmanouela Stachtiari, Anakreon Mentis, and Panagiotis Katsaros. Rigorous analysis of service composability by embedding WS-BPEL into the BIP component framework. In *2012 IEEE 19th International Conference on Web Services*, pages 319–326, 2012.

[76] Boettcher Carolyn, DeLong Rance, Rushby John, and Sifre Wilmar. The mils component integration approach to secure information sharing. In *27thAIAA/IEEE Digital Avionics Systems Conference*, St. Paul, MN, Oct. 2008.

[77] D-MILS Project. `http://http://www.d-mils.org/`.

[78] J. M. Rushby. Design and verification of secure systems. *SIGOPS Oper. Syst. Rev.*, pages 12–21, 1981.

[79] D2.2 translation of mils-aadl into formal architectural modeling framework. Technical Report Tech. Rep.D2.2, Version 1.2, D-MILS Project, Feb. 2014.

[80] Intermediate languages and semantics transformations for distributed mils part 1. Technical Report Tech. Rep.D3.2, Version 1.2, D-MILS Project, Feb. 2014.

[81] Intermediate languages and semantics transformations for distributed mils part 2. Technical Report Tech. Rep.D3.3, Version 1.0, D-MILS Project, July. 2014.

[82] Compositional verification techniques and tools for distributed milspart 1. Technical Report Tech. Rep.D4.4, Version 1.0, D-MILS Project, July. 2014.

[83] Distributed mils platform configuration compiler. Technical Report Tech. Rep. D5.2, Version 0.2, D-MILS Project, Mar. 2014.

[84] Integration of formal evidence and expression in mils assurance case. Technical Report Tech. Rep. D4.3, Version 0.7, D-MILS Project, Mar. 2015.

[85] Compositional assurance cases and arguments for distributed mils. Technical Report Tech. Rep. D4.2, Version 1.0, D-MILS Project, Apr. 2014.

[86] BIP language factory: `http://www-verimag.imag.fr/Language-Factory.html`.

[87] Specification of MILS-AADL. Technical Report D2.1, Version 2.0, D-MILS Project, July 2014.: `http://www.d-mils.org/page/results/`.

[88] Configuration correctness and semantics preserving transformations. Technical Report Tech. Rep. D5.3, Version 1.1, D-MILS Project, Nov. 2014.

[89] A.E. Walsh. *UDDI, SOAP, and WSDL: The Web Services Specification Reference Book*. Prentice Hall, 2002.

[90] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Securing SOAP e-services. *International Journal of Information Security*, 1(2):100–115, 2002.

[91] Giovanni Della-Libera, Martin Gudgin, Phillip Hallam-Baker, Maryann Hondo, Hans Granqvist, Chris Kaler, Hiroshi Maruyama, Michael McIntosh, Anthony Nadalin, Nataraj Nagaratnam, Rob Philpott, Hemma Prafullchandra, John Shewchuk, Doug Walter, and Riaz Zolfonoon. Web services security policy language (WS-SECURITYPOLICY). Technical report, 2005.

[92] Dagmar Koss, Florian Sellmayr, Steffen Bauereiss, Denis Bytschkow, Pragya Gupta, and Bernhard Schaetz. Establishing a Smart Grid

Node Architecture and Demonstrator in an Office Environment Using the SOA Approach. In *First International Workshop on Software Engineering Challenges for the Smart Grid, SE4SG*, pages 8–14, 2012.

[93] Mark S. Anderson, Christopher James North, John Edmund Griffin, Robert Brunyee Milner, John D. Yesberg, and Kenneth Kwok-Hei Yiu. Starlight: Interactive link. In *ACSAC*, pages 55–63. IEEE Computer Society, 1996.

[94] Floor Koornneef and Coen van Gulijk, editors. *Computer Safety, Reliability, and Security - SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings*, volume 9338 of *Lecture Notes in Computer Science*. Springer, 2015.

[95] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *22nd IEEE Computer Security Foundations Symposium, CSF 2009*, pages 43–59, 2009.

[96] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 25–32, New York, NY, USA, 2000. ACM.

[97] R. Focardi, R. Gorrieri, and F. Martinelli. Real-time information flow analysis. *IEEE J.Sel. A. Commun.*, 21(1):20–35, September 2006.

[98] Roberto Barbuti and Luca Tesei. A decidable notion of timed noninterference. *Fundam. Inform.*, 54(2-3):137–150, 2003.

[99] Tesnim Abdellatif, Jacques Combaz, and Joseph Sifakis. Model-based implementation of real-time applications. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 229–238, New York, NY, USA, 2010. ACM.