IMT Institute for Advanced Studies, Lucca

Lucca, Italy

The Signal Calculus: beyond message based coordination for services

PhD Program in Computer Science

XX Cycle

2009

By

Roberto Guanciale

Supervisor

Prof. Gianluigi Ferrari

Abstract

This thesis aims at the definition of foundational techniques driving the design and implementation of a programming middleware supporting the full adoption of a MDD framework for Service Oriented Computing. Our main contribution is the definition of a compositional model for services in the spirit of choreography. Our model takes the form of two-level process calculus that lays at two different levels of abstraction. The local view of coordination is represented by the Signal Calculus, which is tailored to support the formal design of services. The global view of the coordination is supported by the Network Coordination Policies Calculus. The Network Coordination Policies Calculus is the formal machinery we introduced to specify choreography. Distinguished features of our approach are given by the adoption of the event notification paradigm as basic mechanisms to manage service interactions and by the usage of multicast communication. To fill the gap between the local and global views, we relate the semantics of the two calculi by a correctness result that allows us to verify if a design respects a specification. Finally, to highlight the benefit of our approach, we address the issue of defining Long Running Transactions via the Signal Calculus and we provide some sound refactoring rules in the spirit of the MDD approach to software design. The main advantage of these results consists in guaranteeing that the designer can refine the reference implementation without altering the intended meaning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last years, the wide spread of the Internet has made possible entirely new forms of communication among people, companies, and institutions. The open nature of the Internet combined with fast growing of network technologies allowed to design and implement applications (as World Wide Web, peer-to-peer file sharing, and multimedia content distribution to cite a few) that can be provided independently of the location and the underlying platform. Nowadays, the key role played by information technologies suggests to extend these forms of digital communication directly to the information systems of enterprises. However, these new kinds of interactions introduce a wide range of challenging issues:

- Software systems have to be interoperable and rely on heterogeneous networking infrastructures.
- The size of up-to-date distributed applications is orders of magnitude larger than in the past. Moreover, systems are not monolithic entities and require a high level of modularity to support customization and dynamic reconfigurations.
- System maintenance has to became a "lightweight" task, thus allowing applications to be robust to (possibly remote) faults.
- Designers have to make minimal assumptions on external components, thus reducing the risk that changes (be they local or remote) force extensive changes in the application.

One of the emerging approaches in software engineering is the so called Service Oriented Computing and its corresponding architectural style known under the name of Software Oriented Architecture (SOA) [1]. The main building

blocks of SOA applications are *services*. Intuitively, a service is an autonomous agent (or a component) that provides a functionality by interacting over a network. Machine and platform independence are obtained by exploiting standard protocols for interactions. The most widely used form of services are *Web Services* [2; 3; 4; 5]. This kind of services are based on the SOAP [6] protocol stack. Web Services relay on the existing Internet protocols to exchange XML Documents [7]. The adoption of XML allows services to exploit an extensible syntax to exchange structured data. Moreover, the usage of the existing Internet protocols simplifies the adoption of services. In fact, Web Service frameworks can delegate to the existing infrastructures the management of network communications. The technological openness of Web Service specifications allows the design and the implementation of loosely coupled systems capable of minimizing the impact of a change in the business relationships.

A service can be considered as a distributed procedure that can be invoked using the classical remote procedure call style. However, SOA focuses on a more structured scenario [3; 4; 5], in which several service providers are involved to achieve a common goal.

The development of SOA applications cannot be achieved by simply equipping existing programming languages with the RPC style of service invocation. In fact, the methodologies supporting the development of SOA applications must consider and handle some specific issues:

- Since SOA applications are developed on top of a suite of interoperable networks protocols, the development methodology must reduce the effects of architectural changes.
- System complexity cannot be governed by a monolithic structure managing all aspects.
- The definition of the process logic must be clearly separated from the service implementations. Indeed, a service should be easily replaced in order to tackle changes that may be required during the application lifetime.
- Services must be flexible and be invokable by possibly many other services. In a sense, a single service may be part of several applications at the same time.
- To minimize the assumptions on other components, usually services are developed as stateless systems. However, when several services collaborate in a structured scenario, it is necessary to track the distributed progress of the overall computation.

1.1 Developing SOA Applications

It is widely recognized that the complexity of developing of SOA applications can be significantly reduced by the adoption of the Model Driven Development (MDD) approach [8]. MDD suggests to model different aspects of a system using several domain specific languages (DSL). Since the domain of a DSL is confined, the language can be more expressive in its specific domain than general purpose languages. Thus, the designer can manage more efficiently a specific subset of the aspects of a SOA application. The models thus obtained are then repeatedly transformed in order to reflect more concrete aspects. A key feature of the MDD approach is that the abstract specifications can be reused when the software architecture changes.

Also, new design patterns are required since the specification of applications in a SOA context must be clearly separated from the implementations of the services applications consist of. A main challenge is given by the definition of patterns for service coordination. Two main approaches have emerged: *orchestration* and *choreography*. Both styles of coordination describe the so called *business processes*[1] by a global point of view. Thus, the software designer can model structured interactions even if the involved services are stateless. However, the two approaches exploit different strategies.

Orchestration is an executable representation of the coordination required by business processes. The orchestration manages the execution order of services by introducing a controller engine, called *orchestrator*. The standard technology to implement orchestration is WS-BPEL [9].

Choreography is a collaborative style of coordination. It describes the observable behavior of services by constraining the messages exchanged among them. A choreography resembles a contract among parties and provides a specification of the coordination. It must be projected over each participant and locally implemented. The standard to design choreography is WS-CDL [10].

Recently, several vendors have proposed their own solutions to implement and compose services. However, both standards and proprietary solutions are informal specifications, making difficult to provide reasoning techniques. It is worth noting the adoption of a MDD approach requires the introduction of suitable techniques based on formal ground. In particular, we argue that supports are required:

- To statically verify if the implementations respect the coordination poli-

[1] The term *business process* denotes domain dependent functionality of applications or their components. Namely, it represents the structure of inter-company activities required to reach a common intent.

cies.

- To define the core set of primitives of new DSLs, to study their expressiveness and to evaluate if the resulting programming model is suitable for a specific domain.

- To define the semantics of coordination languages, allowing to express unambiguous specifications that represent contracts among parties.

- To verify the correctness of MDD activities like refinements, transformations and refactoring of models, ensuring that the resulting models respect the more abstract specifications.

1.2 Formalizing Service Oriented Computing

Arguably, the strict interplay between formal approaches and standards can provide the way to greatly increase robustness of SOA applications.

In the last few years, several approaches have been proposed to formalize SOA technologies and languages. These approaches extend the techniques developed to deal with concurrent and distributed systems. We briefly classify the existing approaches into two main groups:

Interaction based calculi Several process calculi have been introduced to study the foundation of service orchestration and choreography, see [11; 12; 13; 14; 15; 16; 17; 18; 19] to cite a few. These approaches aim at formalizing various aspects of service coordination, e.g. BPEL [11; 12; 13]. Other approaches [14; 15; 16; 17; 18; 19] advocate the notion of *session* as the correct abstraction mechanism for enclosing an arbitrarily complex interaction between services in order to guarantee e.g., that, during a service conversation, messages are routed as desired. A different programming model for service orchestration has been adopted by Orc [20]. The basic computational entities orchestrated by Orc expressions are sites. A site computation can start other orchestrations, locally store the effects of a computation, and make them visible to clients. Orc provides three basic composition operators, that can be used to model some common workflow patterns, identified by Van der Aalst et al. [21]. Finally, the functional programming paradigm has been extended to support SOA issues. In particular λ_{rec} [22] behavioral types have been used to statically enforce security policies of service orchestration.

Flow based models These approaches are centered around the notion of activities, which are threat as *black box*, namely their behaviors are not specified. Flow languages describe concurrent systems by defining the temporal dependencies among the activities. Well known examples of this approach are YAWL [23], saga [24; 25] and SRML [26] to cite a few. Flow based models permit to describe the core of the business process abstracting from the distribution of activities and communication involved

1.3 Main contributions

The problem of supporting the development of SOA applications (from requirement and design to implementation and maintenance) is at the edge of research in software engineering. Independently from the underlying technological supports, we argue that software engineering methodologies and related programming middleware must properly support the shift from traditional architectural styles of distributed systems to SOA to better accommodate the constraints posed by this new computational paradigms. The present thesis intends to address this issue.

A major contributing of the thesis is the design of an innovative MDD methodology and its supporting middleware relying on a formal programming model. This thesis presents the formal aspects of a research that has nevertheless been applied. In fact, most of the theoretical achievement constitute the basis for the development of a Java API presented in [27] with further refinements and examples of the applicability of our theoretical framework.

Our main contribution is the definition of a compositional model for services that takes inspiration from existing process calculi. To better understand the technical choices at the basis of our approach we focus on choreography based process calculi. Usually, these approaches (e.g. [19; 28]) manage choreography by two different levels of abstraction, referred as *local and global view*. Intuitively, the global view of a system specifies the choreography by constraining the order of service interactions. Instead, the local view designs and models each service by describing its behavior. The formal relation between the two views of a system allows to:

- verify if independently developed services respect a choreography,
- generate the skeleton implementation of services starting from a choreography,
- provide semantic techniques to search from a registry the services that can collaborate to satisfy a choreography.

In the spirit of choreography, our model takes the form of a two-level process calculus laying at two different levels of abstraction. The local view of coordination is represented by the Signal Calculus (*SC*), which is tailored to support the formal design of services. The global view of the coordination is supported by the Network Coordination Policies calculus (*NCP*), which provides the formalism to specify the choreography.

A distinguished feature of our approach is given by the adoption of the event notification paradigm as basic mechanism to manage service interactions. In our proposal, services are *components* that interact by issuing/reacting to events. Components represent the main computational units emboding behavior. In order to provide a design language that abstracts from platform constraints, we exploit the non-brokered event-notification pattern [29]. Each component is responsible to manage the subscriptions for its events, thus not requiring any centralization point.

A further distinguished feature of our approach is the adoption of multicast communication. When a component raises an event, it is notified to all subscribers, transparently to the emitter. Furthermore, notification relies on an asynchronous communication mechanism. Namely, the emitter does not wait for the reception of the raised event by the subscribers.

To highlight the benefits of an event-based programming model with multicast notification, we informally discuss an example. Let us consider a service named "s" that requires a certain amount of resources to provide its functionality. The service maybe invoked by several clients named "c_i" to achieve a common goal. In other words, all clients permit the execution of the service by providing their resources.

The business process is summarized as follows:

1. when the negotiation phase starts, the service notifies to all clients its resource requirements,
2. each client can issue its offer of resources or ignore the service demands,
3. if no client responds to the service demand, negotiation fails and the service functionality is not provided,
4. if the service receives a sufficient amount of resources the negotiation phase terminates, otherwise it restarts,
5. if the negotiation phase successes, the service activates its functionality.

This pattern can be used in several contexts to model a collaborative application. The system can be implemented using the event-notification paradigm as follows. We start classifying the exchanged events into two groups:

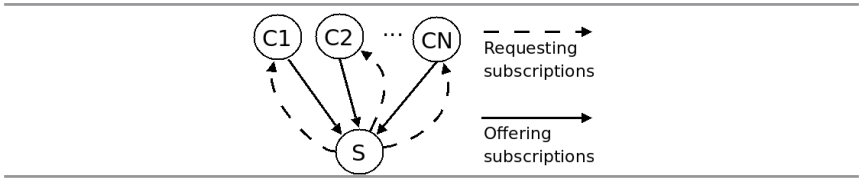


Figure 1.1: Services that collaborate to supply a resource set

- *Offering events:* An "offering event" represents the offer of a new resource from a client. Upon the reception of an offer, the service activates its functionality if the required amount of resources has been reached. Otherwise the service forwards a new request to the clients, restarting the negotiation phase.

- *Requesting events:* A "requesting event" represents that the resources acquired by the service has changed, but the required amount has not been reached. Upon the reception of a request notification, a client can discard service demand or offer a new amount of resources.

Figure 1.1 illustrates the instance of this scenario where three clients are involved. Component inter-connections represent the subscriber relations, which define the multicast routing policy. For example, if the service raises *requesting event* multiple copies are delivered to all involved clients. If a client raises *offering event* only one copy is delivered and targeted to the service.

Event notification and multicast communication can be implemented on top of channel based process calculi (e.g. π-calculus [30]). However, this approach mixes the management of the subscription relations among components with their behaviors. If we focus on our running example we note that, if the number of involved clients changes, the service must be explicitly reprogrammed to support the new situation. Instead, a native multicast model hides communication complexity and permits to program the behavior of the service independently from the number of involved clients.

Several patterns of event notification [29] exist and they mainly differ for the way components subscribe to and filter events. We classify event-notification patterns into three main classes:

- *topic based* event-notification classifies events into topics which subscriber register too

- *content-based* event-notification allows subscriber to filter events in term of the structured data contained into them.

- *type-based* event-notification allows subscribers to register for events that satisfy a given property, expressed as a type.

Our first contribution is the definition of the design language: the Signal Calculus (*SC*). The programming model of *SC* has inspired the design and the development of a programming language (*SCL*) and the implementation of its run-time (*JSCL*). The three forms of event notification discussed above have led to the definition of three *SC* dialects. The topic based dialect is the simplest version of *SC*, allowing us to focus on the main concepts of the calculus. The content based dialect is introduced to extend our framework with session handling. Namely, *SC* permits to track the progress of a process by allowing components to specialize their behaviors for events having a specific structure. This approach to the management of the control flow is inspired by the notion of correlation sets. Finally, the type based dialect has been developed to extend our approach to deal with properties of events. The basic intuition is that the programmer can exploit types to drive the service activation and to model coordination policies.

Our second contribution is the definition of the choreography model: the Network Coordination Policies calculus (*NCP*). Both *NCP* and *SC* support event based interactions and multicast communication. However, they lay at different levels of abstraction and therefore some key differences between the two models exist. While each *SC* component is responsible to manage its subscription, *NCP* models this information by a global point of view, introducing the notion of *network topologies*. Informally, a network topology represents the subscriptions of all components involved by a coordination. Moreover, *NCP* and *SC* involve components in a different way. Computations of *SC* are boxed into components and represent the threads executed by the services. Instead, computations of *NCP* represent global policies by specifying the order of execution of component actions. For example *NCP* can express the policy "*Upon the reception of an event X by the component A, the component B must raise an event Y*".

Although *NCP* is reminiscent of the asynchronous π-calculus, its semantics is centered on network topologies, that are the environment of the computation. In fact, the observational semantics of a *NCP* choreography depends on and can affect the network topology in which it is evaluated. The abstract semantics allows us to reason about the behavior of systems and provides a formal definition of choreography satisfaction. More in detail, we developed a theory of weak

bisimilarity for *NCP* and we studied its compositionality properties.

To fill the gap between the local and global abstraction levels, we relate the semantics of the two calculi by a correctness result: for each *SC* design, there is an *NCP* choreography that reflects all the properties of the design. We establish this result by the introduction of a semantics-based transformation, mapping an *SC* design into an *NCP* choreography. We say that a *SC* design implements the *NCP* choreography if their translations to the global level of abstraction are weakly bisimilar. This is a semantic notion of satisfaction and can be mechanically checked in the finite-state case.

This notion of satisfaction has the main benefit of supporting the development of systems in a Model Driven Development fashion. The designer can define successive *SC* models that implement the system, each of them is obtained refining the previous one to add more details. The conformance of each model with respect to an *NCP* specification can be formally verified.

The *SC* and *NCP* calculi are intended to provide a formal framework on top of which more abstract languages can be formally implemented and verified. We investigated this topic for an existing flow language. In particular we consider the saga calculus [24; 25] to model the control flows of business processes and to handle their transactional aspects. The saga calculus has been also used to provide a formal interpretation of the semantics of the a subset of the Business Process Modeling Notation [31] (BPMN). We show how *SC* can be used to formally represent Long Running Transactions (LRT) designed in saga. The semantic based encoding of saga into *SC* enables designers to specify LRTs using a flow language and to mechanically obtain a reference *SC* design that respects the transactional requirements. Then, the designer can enrich the *SC* model to care about relevant aspects that cannot be described using saga, which is a more abstract than *SC*. The correctness of the refined *SC* models can be checked with respect to the reference implementation, by using our definition of satisfaction.

Finally, our theoretical results have provided the basis to define a methodology to refine *SC* models implementing saga processes. In particular, we studied some refactoring rules that address some crucial issues of the deployment phase, that are the possible alterations that one would like to apply at the *SC* level where they can be more suitably tackled. Arguably, refactoring does not have to alter the intention of the designer, namely, refactoring rules must preserve the intended semantics. Our refactoring rules are proved sound by showing that they preserve weak bisimilarity.

To summarize this thesis has shown process calculi techniques can be adopted and scale-up to give semantic definition of a full-fledged programming methodology and its supporting middleware. This "language complexity" has led to proofs quite involved even if the proof strategy is intuitive and easy to follow.

1.4 Structure of the thesis

The thesis presents a two-level process calculus to model local and global views of service choreography. We study the interplay between the two levels of abstraction to provide effective reasoning techniques

In Chapter 2, we review the technological and theoretical backgrounds that will be used by our framework. We introduce the idea of SOA and we review the main features of the π-calculus. We also introduce the BPMN notation. BPMN provides a compact and graphical notation of transactional aspects of business processes. We also describe the saga calculus, a flow language designed to deal with long running transactions.

In Chapter 3, we present the Signal Calculus family of process calculi. Signal Calculus (*SC*) is the formal machinery we propose to design service coordination using the event notification paradigm. We develop three dialects of *SC* in order to deal with the different styles of event notification: *topic based*, *content based* and *type based*. The *SC* programming model has inspired the definition of the *SCL* programming language and the its reduction semantics has driven the implementation of the language run-time, called *JSCL*.

In Chapter 4, we develop the choreography model of *SC*. In particular, we introduce the syntax and the observational semantics of Network Coordination Policies calculus (*NCP*). We also relate the semantics of *NCP* and *SC*, in order to formalize when an *SC* design respects an *NCP* specification. To emphasize the properties of our approach we first introduce a semantic based transformation that provides a reference *SC* design starting from a saga model. Then, we study the formal properties of some refactoring rules applied to transactional behaviors, verifying that the refactoring transformation preserve weak bisimilarity.

In Chapter 5, we illustrate the usage of our framework through a case study [32] borrowed by the SENSORIA project [33]. We address the problem of developing a service oriented application for an automotive system that involves several services. We focus on implementing the transactional properties of the example and we exploit our refactoring rules to address the deployment phase of the scenario.

In Chapter 6, we describe the Event based Service Coordination (ESC) framework and we focus on the interplay between the actual programming primitives and our theoretical constructions. We briefly introduce the three main components of ESC: the programming language, the run-time that implements the communications and the integrated development environment.

1.5 Origin of the chapters

Many chapter of this thesis are based on already published papers. In particular:

- The design of the *SC* family of process calculi described in Chapter 3 appears in [34; 35]
- The definition of our choreography model described in Chapter 4 is based on [36; 37]
- The refactoring methodology presented in Chapter 4 appears in [38]
- Chapter 5 extends and develops the experimental results presented in [39]
- Chapter 6 is based on the idea presented in [34; 39; 40]

Notice that the above list points out the papers where the results of this thesis have been introduced and described for the first time. However, there are some results which are contained in this thesis and have never written before. Most of these are contained in Chapter 4, which is a significant extension of the results presented in [36; 37; 38]. Moreover, experimental results for heterogeneous networks are reported [41]. Finally, a presentation of some of the results of this thesis targeted for IT professionals has been published in [42].

1.6 Acknowledgments

This research has been supported by the EU FET-GC2 IST-2004-16004 Integrated Project Sensoria and by the Italian FIRB Project Tocai.it.

Chapter 2

Preliminaries

This Chapter reviews the main concepts and notions that will be used throughout the thesis. We first start by introducing the Service Oriented Architecture (SOA) paradigm. We also highlight some challenging issues of the SOA paradigm. Finally, we review the main features of the formal techniques of the basis of our work.

This Chapter is organized as follows. In Section 2.1, we introduce the idea of Service Oriented Architecture (SOA). We summarize the standard technologies and languages developed deal with SOA. We highlight the open issues in the development of service oriented applications. In Section 2.2, we describe the Business Process Model Notation (BPMN), an emerging standard to model business processes through a graphical notation. In Section 2.3, we review the syntax and semantics of the π-calculus, probably the most acknowledged calculus to design and reason about concurrent systems. Finally, in Section 2.4, we describe the SAGA calculus, one of the first proposal to deal with SOA notion of long running transactions.

2.1 Service Oriented Architectures

Usually, the term *service* refers to a system that supports interoperable machine-to-machine interactions over a network. This definition covers many different approaches, but the most widely used form of services are *Web services* [43]. Technically, a Web service is a system that handles XML [7] based messages. Interest in Web services has rapidly grown for several reasons. The exploiting of XML as a platform agnostic formalism to deliver inputs and outputs of services

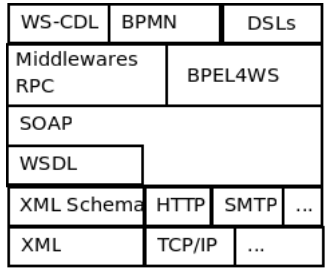


Figure 2.1: Web service stack

allows systems to collaborate independently by their platforms, their implementation languages and their machine architectures. Any programming language able to manage plain text messages can handle XML documents. Moreover, Web services use existing Internet protocols to deliver their documents, thus simplifying the adoption of services and the integration of systems using the existing information technology infrastructures. For example, services that exploit the HTTP [44] protocol can be consumed and deployed using the existing and affordable HTTP clients and servers [45; 46].

Web services rely on the SOAP [6] protocol stack depicted in Figure 2.1. XML is used to share structured data in a machine independent form. The main feature of XML is that it is extensible. In fact, it can be used to create a custom markup languages for specific scopes. The XML Schema [47] is used to specialize XML by defining new *document types*. They are expressed in terms of constraints on the structure and content of documents that should be compliant.

Several types of document has been standardized using XML Schema. For example the Scalable Vector Graphics [48] (SVG) and the Universal Business Language [49] (UBL) define standard file formats to represent two-dimensional graphics and purchase orders respectively.

XML and XML Schema simplify the sharing of data among heterogeneous systems. Before the adoption of XML as standard to share data, a designer had to define special file formats, requiring to write informal specifications and had hoc parsers for all whooshed formats. In the last years several tools has been developed to simplify the management of XML documents and XML schemata. One of their main features is the automatically serialization and deserialization of language data structures to and from XML documents [50; 51].

The Web Service Description Language [52] (WSDL) permits to specifies the interfaces of Web services. It is an XML description language that defines, using

XML Schema, the syntax of documents that can be sent to and received from a web service. Moreover, a WSDL document specifies all network requirements to communicate with the service. It defines which application protocol must be used to transport request and responses and the address of the service. WSDL clearly separates the definition of the structure of the exchanged documents from the application protocol used to deliver them. The formal definition of service interfaces using a standardized language simplifies the usage and the implementation of services. Tools can automatically generate the skeleton source code to invoke and to implement a Web service starting from its WSDL [45]. On the other hand, a web service framework can directly generate the WSDL of a service starting from its implementation.

The simplest way to invoke web services is by resorting the standard Remote Procedure Call (RPC) style. This considers a service as a distributed procedure, in the style of the Object Oriented Programming (OOP) interaction pattern. To activate the procedure, a client must send a well formed request to the service and then wait for the corresponding response, terminating the interaction.

A more complex usage scenario of services is the Software Oriented Architecture [1] (SOA). In the SOA approach several services provided by many companies are involved to achieve a common goal. The collection of structured activities required to accomplish the intent is called *Business Process*. In this scenario the simple RPC style is not suitable anymore. A more complex approach must be exploited to coordinate many services involved into a single business process. SOA suggests two main style of service coordination; *Orchestration* and *Choreography*.

The term orchestration refers to an executable business process that may interact with Web services. This process drives the interaction among services and manages their execution order. The standard adopted to implement service orchestration is Business Process Execution Language for Web Services (WS-BPEL [9]). It is a business language specifically designed to deal with composition of services. The language permits to specify the roles of each of the participants and the logical flow of the messages exchanged from the perspective of the process. WS-BPEL provides programming language constructs (sequence, switch, while, pick) to specify the logic of the message flow. The language is designed to define processes that are executable by an *orchestrator*, which is the application that manages the whole sequence of the service invocations.

The term choreography refers to a more collaborative style of coordination, where each participant describes the part it plays in the interaction. For this reason, choreography requires the *federation* of business parties. The standard language adopted to define service choreography is Web Services Choreography Description Language [10] (WS-CDL). It describes the messages exchanged

among services that participate in a collaborative environment. A key aspect of WS-CDL is that it specifies only the observable behavior of the involved services. It does not address the definition of an executable business process. In fact, there is no single "controlling" process managing the interactions. WS-CDL can be viewed as a layer on top of the existing Web service stack, since the overall description of the behavior must be projected over each participant and locally implemented (exploiting either an RPC style framework or an orchestration engine).

It is worth noticing that, while WS-BPEL describes an executable process from the perspective of one of the partners, WS-CDL takes more of a collaborative and choreographed approach, requiring that each participant guarantees that the message exchanges satisfied the specified business process.

We conclude this start review by highlighting We highlight the main requirements and the main features that must be handled by a service oriented architecture:

- The architecture must clearly separate the process logic from the implementation of the services involved. This separation is usually achieved through an orchestration engine that handles the process flow or by an inter-business contract that specifies the business process in the choreography style. The separation of the coordination logic from the internal service implementation permits to an organization to change services as business requirements change.

- The business processes should be recursively composed. Namely, the service coordination that implements the business process can itself be provided as a service, allowing processes to be composed by a higher-level process.

- Systems must be able to maintain state across service requests. The language and the underlying infrastructure would provide session handling mechanisms to correlate requests in order to build complex conversations.

- Architectures must deal with exception handling and transactions. In a loosely coupled environment, resources cannot be locked in a transaction that runs over long period of time. More relaxed form of atomicity must be provided and guaranteed.

- Organizations should formalize and agree on standard formats to describe shared concepts

- Changes in the underling network infrastructures should not affect the service implementations and the business processes. Moreover, the deploy of systems should be optimized to take benefit from available network settings.

The developing of distributed and loosely coupled systems like SOA must handle a wide variety of aspects. This task can take benefit from the so called Model Driven Development approach [8]. MDD suggests to implement systems by exploiting several domain specific languages (DSL). Each of them should be suited to describe different aspects of the applications. The models developed by using these languages are then transformed into others exploiting less abstract languages, which implement the previously defined aspects of the system. The designer can then enrich the application by describing the lower abstract details, which will be implemented by lower languages

Adoption a MDD strategy systems are repeatedly transformed so that specific concerns are confined at different stages. In fact, MDD typically starts from a (semi)formal system specification that focuses on the core business process and neglects other aspects (e.g., communication mechanisms or distribution) tackled by subsequent transformations.

Software companies have developed several standards to implement and compose services. However, the majority of the proposals are informal specification, making difficult to provide reasoning techniques. For example, the composition of services and their verification are still manually obtained. A formal approach to deal with service architecture can supply middlewares that simplify the building and the composition of Web Services. Moreover, a formal methodology can provide proof techniques to statically verify local (participant view) and global (choreography view) properties. The adoption of the SOA paradigm allows organizations to reduce the cost of maintaining their IT-structure. This benefits can be further extended with the adoption of formal development techniques, allowing the organizations to focus on proprietary business processes and verify their interactions.

A key aspect of model driven framework is the *refactoring* capability. The refactoring rules address some crucial issues of the development phase, that is the possible alterations that one would like (or has) to apply at the lower abstract level where they can be more suitably tackled. Arguably, refactoring does not have to alter the intention of the designer, namely, refactoring rules must preserve the intended semantics.

2.2 The Business Process Modeling Notation

The Business Process Modeling Notation (BPMN [31]) is an emerging standard to describe business processes through a graphical notation. A key aspect of BPMN is that the notation abstracts from all details that are out of the scope of business process modeling. In other words, BPMN permits to design the dependencies among processes regardless their implementation, data carried in communications, communication paradigm and distribution.

The main building blocks of a BPMN design are the *flow objects*, which represent activities and events involved by the business process. Flow objects are connected by arrows, which define the dependencies among this entities. In this section we briefly introduce by some running examples the key features of the BPMN notations.

The BPMN process depicted in Fig. 2.2a describes a simple business process that specifies the execution of the task A. Circles represent events, something that happens and that have to be treated by the system. All BPMN processes involve two special events: the starting point of the business process, represented by the single edge empty circle on the left, and its termination, represented by the double edge empty circle on the right. The rounded corner box A represents the task to be executed. The model depicted in Fig. 2.2b describes a sequence of two tasks. The arrows connecting the flow elements define their temporal order and their dependencies. In the example, the task B can be executed only after the termination of A. The temporal order of execution, defined by the dependencies among elements, is usually referred to as *forward-flow*.

The model depicted in Fig. 2.2c describes a concurrent process, consisting of activities A and B. When the process starts, both the task A and B can be executed independently. The crossed box represents a synchronization barrier, waiting for the termination of all elements connected by an incoming arrow before to propagate the forward-flow execution. The activation of the task C can start only after the termination of both A and B.

A BPMN process can involve several sub-processes. Each of them must contain its own start and end events and can be used as any other flow element by other business processes. The model depicted in Fig. 2.2d is semantically equivalent to the one described in Figure 2.2c, but the concurrent execution of the tasks A and B has been grouped into a sub-process.

BPMN can also be used to model transactional properties of processes. Since the execution of a business process can continue for long time, the traditional methodologies to ensure atomicity are not suitable. Instead of exploiting locking and rollback mechanisms to fully undo incomplete executions a more relaxed form of atomicity is granted. Each task is equipped with a *compensation* that

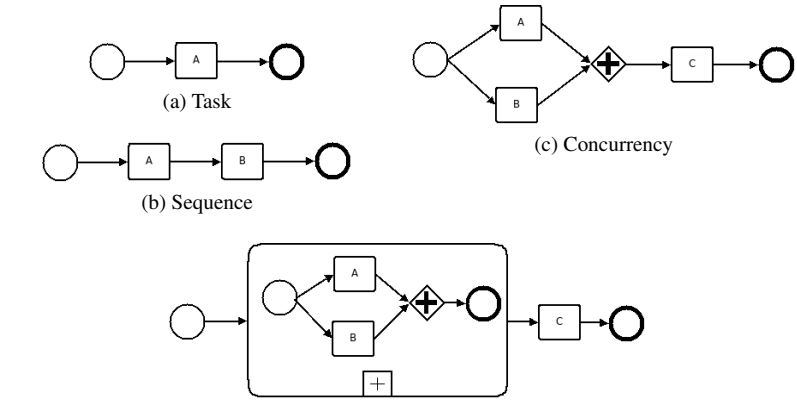
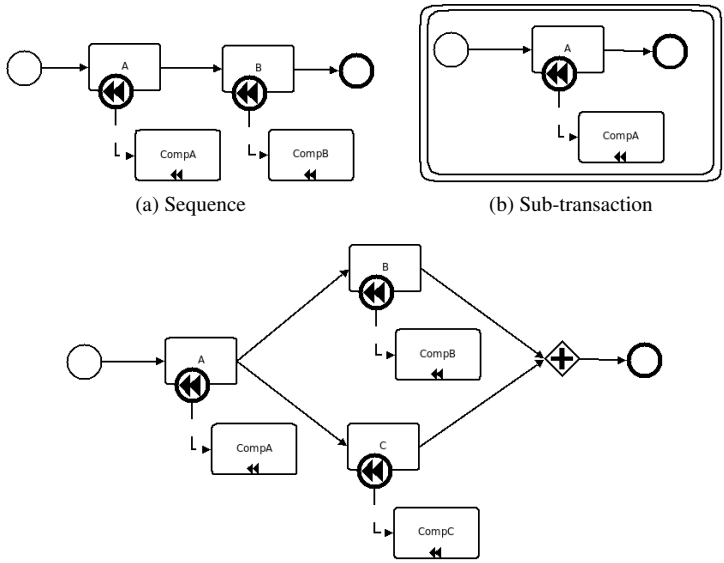


Figure 2.2: Examples of BPMN processes

is responsible to partially recover the task effects if the execution of the whole process cannot be completed. BPMN compensations are represented by tasks connected to the exception events by dotted dashed lines. A key aspect of long running transactions is the order in which compensation must be executed, commonly referred to as *backward-flow*. Intuitively, the backward-flow is the inverse of the temporal order of the corresponding tasks (forward-flow). If the execution of a task fails, the forward-flow has to be interrupted and the backward one starts.

The BPMN specifications depicted in Fig. 2.3a and Fig. 2.3c employ the transactional facilities of BPMN to represent sequential and parallel composition of compensable tasks. Since backward-flow has to be executed reversely to the forward-flow, the parallel composition requires a synchronization of the compensations *CompB* and *CompC*. In particular, if the whole business process fails, the compensation *CompA* can be executed only after that both previous compensations have terminated. BPMN permits to design nested transactions, as depicted in Fig. 2.3b. A double edge box represents a sub-transaction. The start and termination events of sub-transactions are private, namely they are not visible outside the sub-transaction. Intuitively, this scoping mechanism allows sub-transactions to hide any fault of a contained task to external processes.

In summary, a BPMN specification describes the work-flow of a distributed system and its transactional properties by a global point of view. The software architect can abstract from the distribution of the activities, the communication



(a) Sequence

(b) Sub-transaction

(c) Concurrency

Figure 2.3: Examples of BPMN transactional processes

mechanisms and the technologies that will implement each activity. However, converting BPMN models to executable entities is not direct. An implementation of a BPMN model must be enriched with details regarding the communication paradigms and protocols used to coordinate distributed tasks and the data exchanged among them.

2.3 The π-calculus

The π-calculus [30] is probably the most acknowledged formal mechanism to design and reason about concurrent systems. We introduce this calculus because our formal framework takes inspiration from the techniques developer for the π-calculus.

The basic entities of the language are processes and names. Processes are autonomous agents that communicate by exploiting names. The π-calculus extends *ccs* [53] with the ability to dynamically allocate names and exchange them among processes, permitting to describe concurrent systems whose configurations evolve at run-time.

Names can represent communication channels or ports. The set of names N is assumed to be infinite and ranged over by $a, b, c, \ldots$. Names let processes to communicate and exchange information: more precisely, π-calculus processes can exchange names in point-to-point fashion. *Processes* define the behaviors of systems and are denoted by $P, Q, R, \ldots$. *Guards* are special processes that perform an action before to continue their execution, they are ranged by $G, H, \ldots$. The π-calculus syntax is given in Table 2.1.

$P, Q ::=$	$P \mid Q$	$\mid$	$(\nu x)P$	$\mid$	$!G$	$\mid$	G
$H, G ::=$	0	$\mid$	$\bar{x}y.P$	$\mid$	$x(y).P$	$\mid$	$\tau.P$	$\mid$	$G+H$

Table 2.1: π-calculus syntax

The *parallel composition* $P \mid Q$ represents the concurrent execution of P and Q. The *restriction* $(\nu x)P$ declares a new unique name x that will be used by P, this name is distinct from all external names. The *bang* $!G$ represents an unbound concurrent replication of the guard G.

The nil process (0) is a process that cannot perform anything and represents an inactive system. In the process $\pi.P$ (where $\pi \in \{\tau, x(y), \bar{x}y\}$), the prefix π represents the *atomic action* that has to be performed before the execution of

the *continuation process* P. The so called *silent* action τ represents an internal computation. The *input* action $x(y)$ represents the input of a value (a name) on the name x and binds the received value with y. The *output* action $\bar{x}y$ represents the output of the value (the name) y over the name x. The notion of π-calculus name differs from the standard notion of *socket* or *port*; a name is not owned by a process but can be shared among several processes. More precisely several processes can both output and input values on the same name. The *choice* $G+H$ represents a process that either behaves a G or as H. Notice that both input and output can *guard* a continuation process, that is executed only after that the communication occurs. For this reason this version of the calculus is called *synchronous* π-calculus.

The free and bound names of a process (denoted by $fn(P)$ and $bn(P)$ respectively) are naturally defined, according to the only two binders of the language: restriction $(\nu y)P$ and input prefix $x(y).P$, which bind y in P.

2.3.1 Examples

It has been widely recognized that the π-calculus provides suitable mechanisms to model several coordination patterns for services. Here we exploit its capabilities to implement the business processes of Figure 2.2. Let A be a BPMN task, hereafter, we denote with P_A the π-calculus process implementing it. Each process P_A owns a globally known name f_A, that triggers its execution. This assumption is strictly related with SOA core features. Indeed, we model each BPMN task as a service that is reachable through a unique name globally known. We assume the existence of a distinguished name f that is used to represent requests to start the whole business process. We present two possible implementations: one exploiting the orchestration paradigm and the other one the choreography. This will be also useful to highlight the main difference between the two approaches. In the orchestration paradigm, tasks are modeled as passive agents, with no knowledge regarding the collaboration. The collaboration is implemented by an external agent (the "orchestrator") that is informed by all other participants about their termination (we will use the name c_A to deliver the messages informing the termination of the task A). In the choreography paradigm, each agent is directly involved in the coordination, performing itself the required communication to satisfy the collaboration demands. Without loss of generality we model the behavior of a BPMN activities as an internal action (τ), assuming that it does not affect the control flow.

$$
\begin{aligned}
O &\triangleq (\nu c_A, c_B)(P_A \mid P_B \mid D) \\
\textit{where } P_A &\triangleq f_A(y).\tau.\overline{c_A}y \\
\textit{and } P_B &\triangleq f_B(y).\tau.\overline{c_B}y \\
\textit{and } D &\triangleq f(y).\overline{f_A}y \mid c_A(y).\overline{f_B}y
\end{aligned}
$$

(a) Orchestration

$$
\begin{aligned}
C &\triangleq (P_A \mid P_B \mid f(y).\overline{f_A}y) \\
\textit{where } P_A &\triangleq f_A(y).\tau.\overline{f_B}y \\
\textit{and } P_B &\triangleq f_B(y).\tau
\end{aligned}
$$

(b) Choreography

Figure 2.4: π sequential composition

Sequential composition: orchestration

The process O (for *Orchestration*) in Figure 2.4a implements the sequential business process of Figure 2.2b. The system is composed by the three processes P_A, P_B and D, implementing the BPMN tasks A, B and the "orchestrator", respectively. The processes P_A and P_B represent two services which interact with a simple communication protocol; each process waits for a request to start the computation, performs an internal action representing the BPMN task and then communicates to the requester its termination.

It is easy to see that the coordination is totally performed by the "orchestrator" D. It waits for a request of activation of the whole business process on the global name f. When it is activated, it sends a message on the name f_A to activate the process implementing the task A. Concurrently, the orchestrator waits for the termination of the activated process, by performing an input on the name c_A. When this notification is received, a message on the name f_B is sent, to activate the implementation of the task B. The orchestrator does not regard about messages on the name c_B, since there are no activities in the business process depending by the task B. Both processes P_A and P_B have no knowledge of the whole business process. They should be involved in other collaboration and used by other "orchestrator" Notice that names c_A and c_B are restricted, to represent that notifications of the termination of processes P_A and P_B are responses that are visible only to the requester (the orchestrator).

Sequential composition: choreography

The process C (for *Choreography*) in Figure 2.4b implements the sequential business process of Figure 2.2b. The system is obtained starting from the implementations of the two tasks P_A and P_B and a "intermediate process". The intermediate process $f(y).\overline{f_A}y$ is used only to start the process P_A when the whole business process is activated. The intuitive idea is that the "intermediate process" acts as

$$
\begin{array}{rll}
O \triangleq & (\nu c_A, c_B, c_C)(P_A \mid P_B \mid P_C \mid D) & \\
\textit{where } P_i \triangleq & f_i(y).\tau.\overline{c_i}y & i \in \{A,B,C\} \\
\textit{and } D \triangleq & f(y).\overline{f_A}y \mid \overline{f_B}y \mid c_A(y).c_B(y).\overline{f_C}y &
\end{array}
$$

(a) Orchestration

$$
\begin{array}{rl}
C \triangleq & (P_A \mid P_B \mid P_C \mid f(y).(\overline{f_A}y \mid \overline{f_A}y)) \\
\textit{where } P_A \triangleq & f_A(y).\tau.\overline{f_c}y \\
\textit{and } P_B \triangleq & f_B(y).\tau.\overline{f_c}y \\
\textit{and } P_C \triangleq & f_C(y).f_C(y).\tau
\end{array}
$$

(b) Choreography

Figure 2.5: π parallel composition

proxy of request. Notice that processes P_A and P_B *hard-wire* in their behavior the collaboration in which they are involved. In particular, the process P_A, after its termination, sends a message directly on the name f_B to start the execution of the task B. Moreover the process P_B does not send any message after its termination, since there are no dependent activities on the business process.

Parallel composition: orchestration

The π process O in Figure 2.5a implements the business process described in 2.2c: As done above, we restrict the names c_A, c_B and c_C to allow the orchestrator D to directly manage task termination. All tasks are implemented in the same way, with the exception of the name used to receive the requests and to notify termination.

The orchestrator D waits for the request to execute the whole business process, then it sends concurrently the notification to start on the names f_A and f_B, to permit the independent executions of both implementations of A and B. When both tasks terminate their execution and the orchestrator receives the corresponding notifications and, therefore, sends the request of activation to the implementation of the last task C.

Parallel composition: choreography

The π process C in Figure 2.5b implements the sequential business process of Figure 2.2c: The system is obtained starting from the implementations of the

three tasks (P_A, P_B and P_C) and a "intermediate process". The intermediate process ($f(y).(\overline{f_A}y \mid \overline{f_A}y)$) plays the role of started: it activates processes P_A and P_B when the whole business process is activated. The intuitive idea is that the "intermediate process" acts as proxy of requests and represent the start event of BPMN. Notice that the processes P_A, P_B and P_C hard-wire in their behavior the collaboration in which they are involved. In particular, P_A, and P_B directly inform the process P_C of their termination (by sending a message on the name f_C). Moreover, process P_C works as synchronization barrier, waiting for two notifications on its activation channel f_C. Notice that process P_C does not send any message after its termination. In fact, there are no further dependent activities on the business process.

2.3.2 Reduction semantics

We now review the reduction semantics of π-calculus. First, we introduce a *structural congruence* (denoted by $\equiv$), The structural congruence is the smallest congruence relation over processes that satisfies the rules presented in Table 2.2.

- $(G, 0, +)$ and $(P, 0, |)$ are commutative monoids:

$$G+H \equiv H+G \quad G+(H+I) \equiv (H+G)+I \quad G+0 \equiv G$$

$$P \mid Q \equiv Q \mid P \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \quad P \mid 0 \equiv P$$

- $!G \equiv G \mid !G$
- $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$ and if $x \notin fn(P)$ then $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$

Table 2.2: π structural conguence rules

The *reduction relation* $P \rightarrow P'$ describes the evolution of the process P to P' by a single computational step. The reduction relation is defined by the inferences rules of Table 2.3.

We briefly comment on the key aspects of the semantics of the π-calculus. If a non-deterministic choice involves non-internal actions, the behavior of the process is determined by the context in which it is executed:

- $0+P$ always behaves as P

$$\frac{P \to P'}{P \mid Q \to P' \mid Q}\,(PAR) \qquad \frac{P \to P'}{(\nu x)P \to (\nu x)P'}\,(RES)$$

$$\frac{Q \equiv P \to P' \equiv Q'}{Q \to Q'}\,(STRUCT) \qquad \frac{}{\tau.P + G \to P}\,(TAU)$$

$$\frac{}{(x(y).P + G) \mid (\bar{x}z.Q + H) \to \{z/y\}P \mid Q}\,(COM)$$

Table 2.3: π reduction rules

- $\tau.P + \tau.Q$ performs an internal choice. The process can choose independently by the context one of the two branches, executing in both cases an internal action.
- $x(z).P + y(z).Q$, where $x \neq y$, cannot autonomously choose whether to activate the input on the name x or the input on the name y. In fact, the rule (COM) can be applied only when the process is executed in a context having at least an output for one of this names. For example, if the process is executed concurrently with $\bar{x}w.R$, only the branch waiting on the name x is enabled. Hence, the parallel composition will perform a communication and will be reduced to $\{w/z\}P \mid R$.

The interplay between the reduction relation and the structural congruence permits to express the so called *scope extrusion*: the capability of a process to discover, through the input primitive, a name that was restricted by a different process. For example, the parallel composition $((\nu y)\bar{x}y.P) \mid x(z).Q$ contains a process restricting the name y. To enable the communication among the two processes, the input must "enter" inside the restriction, using α-renaming and the structural congruence:

$$\frac{y \notin fn(x(z).Q)\bar{x}y.P \mid x(z).Q \to P \mid \{y/z\}Q}{((\nu y)\bar{x}y.P) \mid x(z).Q \equiv (\nu y)(\bar{x}y.P \mid x(z).Q) \to (\nu y)(P \mid \{y/z\}Q)}$$

2.3.3 The asynchronous π-calculus

Languages using the message passing paradigm can be classified in asynchronous and synchronous languages. In the synchronous approach the delivery and the

reception of a message are treat like happening at the same time. In the asynchronous approach the message is sent by an agent, that can perform other task. From the standpoint of language design, the difference between the two paradigms is rather important. Asynchronous communications are closer to the standard distributed system infrastructures and are easier to be implemented. The version of the π-calculus described in the previous section is called *synchronous* π-calculus, since the semantics permits to express processes (e.g. $\bar{x}t.P$) that send a message and whose continuation (P) is executed only after the reception of the message by a consumer. The *asynchronous* version of the π-calculus (in the following π_a) can be easily defined as a subset of the synchronous language that respects the following requirements:

1. outputs have no continuation,
2. output actions cannot be involved in a non-deterministic choice.

More precisely, the syntax of the π_a calculus is defined in Table 2.4.

$$\begin{array}{lll} P,Q ::= & \bar{x}y \mid P \mid Q \mid (\nu x)P \mid !G \mid G \\ H,G ::= & 0 \mid x(y).P \mid \tau.P \mid G+H \end{array}$$

Table 2.4: Asynchronous π calculus: syntax

The reduction semantics of the π_a calculus can be obtained by the synchronous one, since the new language is obtained by only syntactic constraints.

2.3.4 Asynchronous π-calculus: behavioral semantics

In the literature, several abstract behavioral semantics for the asynchronous π-calculus have been proposed [54]. We now briefly describe the "direct HT-transition system" [55], since it is the basic building block of the abstract semantics we will develop in this thesis (Section 4.1).

The direct HT semantics is obtained by a labeled transition system, whose labels are of the form

$$\alpha ::= \tau \mid \bar{x}y \mid \bar{x}(y) \mid x(y)$$

Label τ models the unobservable actions. $\bar{x}y$ is a *free* output action and represents the communication of y over x. $\bar{x}(y)$ is a *bound* output and represents the delivery of a message containing a value (y) that has been restricted. Finally, $x(y)$ is the

input action, and represents the reception of a message on the name x carrying the value y. All names of actions are free, with the exception of the value of the bound output. $n(\alpha)$ is used to denote the union of free and bound names of the action α.

$$\frac{P \equiv P' \xrightarrow{\alpha} Q' \equiv Q}{P \xrightarrow{\alpha} Q}\ (cong)$$

$$\frac{}{P \xrightarrow{x(y)}_0 P \mid \bar{x}y}\ (in_0) \qquad \frac{}{x(y).P \xrightarrow{x(z)}_1 \{z/y\}Q}\ (in_1)$$

$$\frac{}{\tau.P \xrightarrow{\tau} P}\ (\tau) \qquad \frac{}{\bar{x}y \xrightarrow{\bar{x}y} 0}\ (out)$$

$$\frac{P \xrightarrow{\bar{x}y} P' x \neq y}{(\nu y)P \xrightarrow{\bar{x}(y)} P'}\ (out_{ex}) \qquad \frac{P \xrightarrow{\alpha} P' x \notin n(\alpha)}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ (\nu)$$

$$\frac{P \xrightarrow{\bar{x}y} P' Q \xrightarrow{x(y)}_1 Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}\ (sync) \qquad \frac{P \xrightarrow{\bar{x}(y)} P' Q \xrightarrow{x(y)}_1 Q' y \notin fn(Q)}{P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q')}\ (sync_{ex})$$

$$\frac{P \xrightarrow{\alpha} P' bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ (comp) \qquad \frac{G \xrightarrow{\alpha} P}{!G \xrightarrow{\alpha} P \mid !Q}\ (rep)$$

Table 2.5: HT labelled transition system

The direct HT transition system is defined by the rules in Table 2.5, where the relation $\equiv$ is the syntactic identity modulo α-conversion. As usual, we omit the symmetric rules for *sync*, $sync_{ex}$, *comp* and *sum* We comment on the most interesting rules. Rule in_1 models input actions; notice that the behavior is defined in the *early instantiation* style, since the rule can be applied for any name (z) that can be received by the process. Rule in_0 permits to perform an input, simply storing the received message for subsequent usages, thus allowing to arbitrarily delay the communication. Rule *sync* allows the communication of a non-restricted name. Rules $synx_{ex}$ and out_{ex} model the scope extrusion of the name y. Notice that the rules *sync* and $sync_{ex}$ depend on the input transitions $\xrightarrow{x(y)}_1$ corresponding to input guards, while the rule in_0 cannot be involved into

the communication.

The notion of *weak* transition system is defined in the standard way as follows:

$$P \stackrel{\tau}{\Rightarrow} Q \quad \text{iff } P \stackrel{\tau}{\rightarrow}^{*} Q \qquad P \stackrel{\alpha}{\Rightarrow} Q \quad \text{iff } P \stackrel{\tau}{\Rightarrow} . \stackrel{\alpha}{\rightarrow} . \stackrel{\tau}{\Rightarrow} Q \; and \alpha \neq \tau$$

The weak transition systems abstract from the interactions performed by processes, hiding they internal actions.

2.3.5 Bisimulation semantics

We now introduce the abstract notion of bisimilarity. The behavioral semantics is important since it not only describes how a set of processes interact with each other, but also permits to reason about isolated subsystems. Bisimulation allows one to check for properties that have to be satisfied by the implementation of a system against its design expressed in a high-level language. Sometimes the implementation is slightly modified in order to verify a subset of the system requirements, e.g. by inserting the implementation in a suitable controlled context or environment, where it can be formally shown that, by construction, only properties of interest can lead to violation of the design. Honda Amadio etc [54; 55] have studied bisimulation for asynchronous π-calculus. We now introduce the direct HT bisimulation. In the Honda-Tokoro vision, in the *bisimulation game*, any process can act as a *buffer* that reads any possible message and stores it, thus effectively not consuming the message. This is done by the rule in_0 of the labeled transition of Figure 2.5. On the other hand, processes that actually can consume a message are not observed at all in the bisimulation. The intuitions is that, in an asynchronous settings, an observer has not direct way of knowing if a message he sent has been received or not.

Definition 1 *The Direct HT bisimulation ($\sim$) is the largest symmetric relation on π_a-terms such that if $P \sim Q$ then:*

- *if $P \stackrel{\alpha}{\rightarrow} P'$, $\alpha \in \{\tau, \bar{x}y, \bar{x}(y)\}$ and $bn(\alpha) \cap fn(Q) = \emptyset$ implies that $Q \stackrel{\alpha}{\rightarrow} Q'$ and $P' \sim Q'$.*
- *if $P \stackrel{x(y)}{\longrightarrow}_0 P'$ then $Q \stackrel{x(y)}{\longrightarrow}_0 Q'$ and $P' \sim Q'$.*

The notion of weak bisimulation ($\approx$) is obtained substituting in the above definition the transition relation with the weak one.

The π-calculus bisimulation can be used to check properties of systems. The standard methodology consists of checking if an implementation (modeled as a

π-calculus process) is bisimilar respect to a specification (sometimes modeled as a *magic* π-calculus process). In asynchronous distributed systems is usually preferred the check of weak bisimilarity, since it permits to compare processes regardless their internal actions.

In Section 2.3.1 and 2.3.1 we presented two π-calculus implementations of the BPMN business process depicted in Fig. 2.2b. Checking the weak bisimilarity of the two processes ($O \approx C$) ensures that the two systems perform the same behavior. In this case both processes have finite states and the proof is easy. Clearly, the bisimilarity check is not always decidable if the processes have infinite states. We refer to [56] for a comprehensive discussion about this topic.

2.4 The saga calculus

We highlighted that enforcing transactional properties of service composition is a challenging issue. Saga [24; 25] is basically the most accepted proposal to deal with long running transactions. Also in this case we stress the role formal techniques may have to standardize the semantics of error recoveries and verify the correctness of the implementation of a process.

The main building blocks of saga are activities, which are short-term transactions that can exploit traditional ACID techniques to ensure their atomicity. Atomic activities are uniquely identified by a name $A, B, \ldots \in \mathcal{A}_{ct}$. Each activity A can have a corresponding activity B, called *compensation*, that can be executed to undo its effects. The pair containing the atomic activity and its compensation is called *step*.

Activities, with their compensation, can be composed to build complex *processes* and *transaction* (also called *saga*). Informally, the difference between a process and a transaction is that the latter is considered atomic: it can be successfully executed (*committed*) or no effect can be observed if it fails (*aborted*).

$$
\begin{array}{l}
X ::= 0 \mid A \mid A \div B \\
P ::= X \mid P;P \mid P|P \mid S \\
S ::= \{|P|\}
\end{array}
$$

Table 2.6: Syntax of steps (X), processes (P) and sagas (S)

The syntax of the saga calculus is defined in Table 2.6. A saga *step* X is an atomic task to be performed. The step 0 represents an inactive task, which cannot

raise errors. The step $A \div B$ represents a compensate task, where A is the atomic activity and B the compensation that can undo the effects produced by A. The step A models an atomic activity having no compensation.

Steps can be composed to build processes. The operators ";" and "|" permit to represent the sequential and the parallel composition of processes, respectively. Notice that sagas, which represent transactions, can be nested. Intuitively, this nesting mechanism allows sub-transactions to hide any fault of a contained activity to external processes. The root level of a hierarchy of sagas is referred as the *top-level saga*.

2.4.1 Big step semantics

The intuition underlying the operational semantics of saga calculus is that partial execution of a saga must be compensated, to guarantee atomicity. The possible behaviors of the sequential composition of activities clarify this idea. Let $A_1, \ldots, A_n$ be activities (having compensations $B_1, \ldots, B_n$); their sequential composition can

- execute entirely the sequence $A_1; \ldots; A_n$, if the whole saga successfully terminates
- execute the compensated sequence $A_1; \ldots; A_j; B_j, \ldots, ; B_1$ for $j < n$, if the activity A_{j+1} fails, recovering the activities already completed $(A_1; \ldots; A_j)$ by executing in reverse order the corresponding compensations $(B_j; \ldots; B_1)$

In literature, the temporal order of execution of activities is usually called to as *forward-flow*, while the order in which compensation must be executed is commonly referred to as *backward-flow*.

Now we review a simplified version of the operation semantics of saga. We focus on executions of a saga that can only commit or compensate all completed steps. This requires that all compensations always terminate with success (relaxing this assumption requires to model executions of sagas that can abnormally terminate.) The full semantic of the saga calculus can be found in [24]. The possible results of the execution of a saga belong to the set $\mathcal{R} = \{\boxdot, \boxtimes\}$, where $\boxdot$ represents the successful execution, while $\boxtimes$ stands for the occurrence of a failure and the full compensation of all terminated activities. We will use $\square$ to denote an arbitrary element of $\mathcal{R}$.

The semantic of saga is given up to a structural congruence over terms, which is defined by the axioms of Table 2.7.

Since sagas abstract from the inner structure of the atomic activities, the semantics of the calculus exploits a context Γ that expresses if an atomic activities

$A \div 0 \equiv A$	$0;P \equiv P;0 \equiv P$	$(P;Q);R \equiv P;(Q;R)$
$P\|0 \equiv P$	$P\|Q \equiv Q\|P$	$P\|(Q\|R) \equiv (P\|Q)\|R$

Table 2.7: saga structural congruence rules

successful terminates or fails. Formally, $\Gamma : \mathcal{A}_{ct} \rightarrow \mathcal{R}$ is a partial function that maps activities to "commit/fail".

The semantics of top-level sagas is defined by the relation $\Gamma : S \xrightarrow{\alpha} \Box$, given by inference rules displayed in Table 2.8. Intuitively, the transition $\Gamma : S \xrightarrow{\alpha} \Box$ states that the saga S produces the result $\Box$ if its atomic activities terminate as prescribed by Γ. Observations α are defined as follows:

$$\alpha ::= 0 \quad | \quad A \quad | \quad \alpha;\alpha' \quad | \quad \alpha|\alpha'$$

An observation α represents uncompensated processes, and $\alpha|\alpha'$ represents all possible interleaving executions of α and α'.

The auxiliary relation $\Gamma : \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle$ defines the behavior of a process P within a saga that already have installed the compensation β, which syntactically stands for a process without compensations. When a process is executed inside a saga, it can commit or fail, but it can also change the installed compensations.

We now comment on the rules in Table 2.8. The rule $(ZERO)$ states that an inactive process can only commit, without changing the installed compensations. The rule $(ACT - S)$ describes the successful execution of the main activity of $A \div B$; since A is the last executed activity, the compensation B is installed in front of β, so that A will be the first activity to be compensated if the next one fails. The rule $(ACT - C)$ describes the failure of the execution, inferred from the context gamma, of the step $A \div B$. The rule activates the execution of the compensation β that was already installed. As described above, we present a simplified version of the saga semantics that require that all compensations never fail ($\Gamma : \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle$). Since A is an aborting atomic activity, A had no effect. For this reason its compensation B is not activated and the observed flow is the one obtained by executing the compensations (β). The rule $(SEQ - S)$ describes the execution of a sequential composition when the first process P commits. The continuation process Q is executed taking into account the compensations produced by P; the result of the sequential composition is obtained according to the execution of Q. The rule $(SEQ - C)$ describes the execution of a sequential composition when the first process compensates. In this case, the next process Q is not executed.

$$\frac{}{\Gamma : \langle 0, \beta \rangle \xrightarrow{0} \langle \boxdot, \beta \rangle} \ (ZERO)$$

$$\frac{\Gamma(A) = \boxdot}{\Gamma : \langle A \div B, \beta \rangle \xrightarrow{A} \langle \boxdot, B; \beta \rangle} \ (ACT - S) \qquad \frac{\Gamma(A) = \boxtimes \quad \Gamma : \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle}{\Gamma : \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle} \ (ACT - C)$$

$$\frac{\Gamma : \langle P, \beta \rangle \xrightarrow{\alpha} \langle \boxdot, \beta'' \rangle \quad \Gamma : \langle Q, \beta'' \rangle \xrightarrow{\alpha'} \langle \square, \beta' \rangle}{\Gamma : \langle P; Q, \beta \rangle \xrightarrow{\alpha;\alpha'} \langle \square, \beta' \rangle} \ (SEQ - S)$$

$$\frac{\Gamma : \langle P, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}{\Gamma : \langle P; Q, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle} \ (SEQ - C)$$

$$\frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, \beta' \rangle \quad \Gamma : \langle P', 0 \rangle \xrightarrow{\alpha'} \langle \boxdot, \beta'' \rangle}{\Gamma : \langle P | Q, \beta \rangle \xrightarrow{\alpha|\alpha'} \langle \boxdot, \beta' | \beta''; \beta \rangle} \ (PAR - S)$$

$$\frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle \quad \Gamma : \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \boxtimes, 0 \rangle \quad \Gamma : \langle \beta, 0 \rangle \xrightarrow{\alpha''} \langle \boxdot, \beta'' \rangle}{\Gamma : \langle P | Q, \beta \rangle \xrightarrow{\alpha|\alpha';\alpha''} \langle \boxtimes, 0 \rangle} \ (PAR - C1)$$

$$\frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle \Gamma : \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \boxdot, \beta' \rangle \Gamma : \langle \beta'; \beta, 0 \rangle \xrightarrow{\alpha''} \langle \boxdot, 0 \rangle}{\Gamma : \langle P | Q, \beta \rangle \xrightarrow{\alpha|\alpha';\alpha''} \langle \boxtimes, 0 \rangle} \ (PAR - C2)$$

$$\frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, \beta' \rangle}{\Gamma : \langle \{|P|\}, \beta \rangle \xrightarrow{\alpha} \langle \boxdot, \beta'; \beta \rangle} \ (SAGA - S) \qquad \frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}{\Gamma : \langle \{|P|\}, \beta \rangle \xrightarrow{\alpha} \langle \boxdot, \beta' \rangle} \ (SAGA - C)$$

$$\frac{\Gamma : \langle P, 0 \rangle \xrightarrow{\alpha} \langle \square, \beta \rangle}{\Gamma : \{|P|\} \xrightarrow{\alpha} \square} \ (SAGA - TOP)$$

Table 2.8: Semantics of sagas

A key aspect of saga is its treatment of compensations of parallel activities. Usually, flow languages [57] state that all compensations must be executed in the reverse order of the completed corresponding activities. However, this requirement is too strong to be suitable for SOA. Enforcing the execution order of the compensations to reflect the order of the main activities executed necessarily requires synchronizations among all performed actions. In the saga approach, the compensation of parallel activities can be executed independently.

The rule $(PAR-S)$ describes the successful execution of the parallel composition, when both processes commit. Since parallel branches and their recoveries are executed independently, the parallel composition of the compensations is installed on the head of ready compensations. Finally the observed flow is obtained by the possible interleaving of the flow of P and Q. The rule $(PAR-C1)$ describes the behavior of the parallel composition when both processes compensate. The execution is stopped and the compensation β installed. Notice that the compensation is started only after both parallel branches terminate their failure recovery, namely a synchronization of the backward-flow is performed. The rule $(PAR-C2)$ describes the execution of a parallel composition when one of the parallel branches commits and the other one fails and compensates. The execution of the successful branches is aborted by executing its compensation β' before activating the original compensation. Notice that the aborting procedure for the successful terminated branches is activated only after that the fault branches have totally recovered their execution.

The rule $(SAGA-S)$ and $(SAGA-C)$ describe the execution of a nested saga. Independently by the termination status of the boxed process P, the saga always commits. This permits to hide the failures of the activities in P to any external process. The successful completion of $\{|P|\}$, rule $(SAGA-S)$, is analogous to the case of a successful execution of P. In stead, if P fails the abort is hidden to the parent, but the observed flow corresponds to the execution of P and the installed compensation β is not modified. Finally, the rule $(SAGA-TOP)$ describes the execution of a top-level saga in terms of the contained process with an empty installed compensation.

2.4.2 Examples

The saga calculus can be used to formally describe BPMN work-flows. We illustrate the semantics of the saga calculus by showing the possible executions of saga processes modeling the BPMN designs depicted in Figure 2.3.

$$\{|A \div C_A; B \div C_B|\}$$

Figure 2.6: The saga representing the BPMN design 2.3a

Sequential composition

The saga encoding of the BPMN process 2.3a is illustrated in Figure 2.6. The saga is simply obtained by the sequential composition of two steps, each of them containing an atomic activity (e.g. A) and the corresponding compensation (e.g. C_A). The forward-flow prescribes that the execution of the activity B can start only after the termination of activity A. Since the activities and compensations are not described in the saga calculus, we use the context to infer if the activities success or fail. We recall that compensations are assumed to always success.

Suppose that the activity A fails its execution, namely $\Gamma(A) = \boxtimes$. The execution trace of the saga is obtained by exploiting the saga operational semantics and is reported in Figure 2.7a. Since A is an aborting atomic activity, A had no effect on the system. For this reason its compensation C_A is not activated. The forward-flow is stopped and the whole observed execution is empty .

Suppose now that the activity A successfully terminates and B fails, namely $\Gamma(A) = \boxdot$ and $\Gamma(B) = \boxtimes$. The trace of the saga is reported in Figure 2.7b. Initially, the activity A is successfully executed and its compensation pushed on the stack of compensations. When the activity B fails, the backward-flow is activated, executing the installed compensations. The whole trace of execution is the sequence $A; C_A$.

Finally, suppose that both activities successfully terminate, namely $\Gamma(A) = \boxdot$ and $B(\boxdot)$. The trace of the sagais reported in Figure 2.7c. The whole saga successes and $A; B$ is observed. Notice that the last compensation installed is $C_B; C_A$. If this saga is composed with a process that fails, the compensation C_B should be executed before C_A.

Parallel composition

The BPMN process 2.3c is encoded by the saga reported in Figure 2.8. The saga composes the three atomic activities and their compensations. Suppose that all activities successfully terminate, namely $\Gamma(A) = \Gamma(B) = \Gamma(C) = \boxdot$ The trace of the saga is reported in Figure 2.9a. The whole saga successes, observing $A; (B|C)$. The trace does not care about which parallel branch is first executed or

$$\dfrac{\dfrac{\dfrac{\Gamma(A) = \boxtimes \quad \Gamma : \langle 0,0\rangle \xrightarrow{0} \langle \boxdot, 0\rangle}{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{0} \langle \boxtimes, 0\rangle}\,(ACT-C)}{\Gamma : \langle A \div C_A; B \div C_B, 0\rangle \xrightarrow{0} \langle \boxtimes, 0\rangle}\,(SEC-C)}{\Gamma : \{|A \div C_A; B \div C_B|\} \xrightarrow{0} \boxtimes}\,(SAGA-TOP)$$

(a) Failure of A

$$\dfrac{\dfrac{\dfrac{\Gamma(A) = \boxdot \quad \Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle}{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle}\,(ACT-S) \quad \dfrac{\Gamma(B) = \boxtimes \quad \Gamma : \langle C_A, 0\rangle \xrightarrow{C_A} \langle \boxdot, 0\rangle}{\Gamma : \langle B \div C_B, C_A\rangle \xrightarrow{C_A} \langle \boxtimes, 0\rangle}\,(ACT-C)}{\Gamma : \langle A \div C_A; B \div C_B, 0\rangle \xrightarrow{A;C_A} \langle \boxtimes, 0\rangle}\,(SEC-C)}{\Gamma : \{|A \div C_A; B \div C_B|\} \xrightarrow{A;C_A} \boxtimes}\,(SAGA-TOP)$$

(b) Success of A, Failure of B

$$\dfrac{\dfrac{\dfrac{\Gamma(A) = \boxdot \quad \Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle}{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle}\,(ACT-S) \quad \dfrac{\Gamma(B) = \boxdot \quad \Gamma : \langle B \div C_B, C_A\rangle \xrightarrow{B} \langle \boxdot, C_B; C_A\rangle}{\Gamma : \langle B \div C_B, C_A\rangle \xrightarrow{B} \langle \boxdot, C_B; C_A\rangle}\,(ACT-S)}{\Gamma : \langle A \div C_A; B \div C_B, 0\rangle \xrightarrow{A;B} \langle \boxdot, C_B; C_A\rangle}\,(SEC-S)}{\Gamma : \{|A \div C_A; B \div C_B|\} \xrightarrow{A;B} \boxdot}\,(SAGA-TOP)$$

(c) Successes of A and B

Figure 2.7: Example traces of the saga sequential composition

$$\{|A \div C_A; (B \div C_B | C \div C_C)|\}$$

Figure 2.8: The saga representing the BPMN design 2.3c

terminated. In fact, the last compensation installed is $(C_B|C_C);C_A$, meaning that if a further failure occurs, the compensations of B and C can be independently executed, but they must terminate before activating the compensation of A.

Now, let us assume that only one branch fails, e.g. C, namely $\Gamma(A) = \Gamma(B) = \boxdot$ and $\Gamma(C) = \boxtimes$. The trace of the saga is reported in Figure 2.9b. The two parallel branches are executed independently. The execution of the branch B is not stopped by a failure on the branch C, respecting the naive semantics. After that both branches terminates their forward-flow, the compensation C_B is activated. Notice that the compensation C_A can start only after that the parallel composition has terminated its recovery, synchronizing the backward-flow.

Finally, suppose that the activity A successfully terminates, while both B and C fail, namely $\Gamma(A) = \boxdot$ and $\Gamma(B) = \Gamma(C) = \boxtimes$. The trace of the saga is reported in Figure 2.9c. The two parallel branches are executed independently. The execution of the branch B is not stopped by a failure on the branch C and vice versa. Notice that the compensations C_B and C_C are not executed, since we assume that the activities B and C support ACID transactions and then their failure does not have any side effect. After that both branches terminates, failing, the compensation C_A is executed, to undo the effects of the committed activity A.

(a) Successes of A, B and C

$$\dfrac{\dfrac{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle \qquad \dfrac{\Gamma : \langle B \div C_B, 0\rangle \xrightarrow{B} \langle \boxdot, C_B\rangle \; \Gamma : \langle C \div C_C, 0\rangle \xrightarrow{C} \langle \boxdot, C_C\rangle}{\Gamma : \langle B \div C_B | C \div C_C, C_A\rangle \xrightarrow{B|C} \langle \boxdot, (C_B|C_C); C_A\rangle}\,(PAR-S)}{\Gamma : \langle A \div C_A; (B \div C_B | C \div C_C), 0\rangle \xrightarrow{A;(B|C)} \langle \boxdot, (C_B|C_C); C_A\rangle}\,(SEC-S)}{\Gamma : \{|A \div C_A; (B \div C_B | C \div C_C)|\} \xrightarrow{A;(B|C)} \boxdot}\,(SAGA-TOP)$$

(b) Successes of A and B, failure of C

$$\dfrac{\dfrac{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle \qquad \dfrac{\Gamma : \langle B \div C_B, 0\rangle \xrightarrow{B} \langle \boxdot, C_B\rangle \; \Gamma : \langle C \div C_C, 0\rangle \xrightarrow{0} \langle \boxtimes, 0\rangle \; \Gamma : \langle C_B; C_A, 0\rangle \xrightarrow{C_B;C_A} \langle \boxdot, 0\rangle}{\Gamma : \langle B \div C_B | C \div C_C, C_A\rangle \xrightarrow{B;C_B;C_A} \langle \boxtimes, 0\rangle}\,(PAR-C2)}{\Gamma : \langle A \div C_A; (B \div C_B | C \div C_C), 0\rangle \xrightarrow{A;B;C_B;C_A} \langle \boxtimes, 0\rangle}\,(SEC-C)}{\Gamma : \{|A \div C_A; (B \div C_B | C \div C_C)|\} \xrightarrow{A;B;C_B;C_A} \boxtimes}\,(SAGA-TOP)$$

(c) Successes of A, failure of B and C

$$\dfrac{\dfrac{\Gamma : \langle A \div C_A, 0\rangle \xrightarrow{A} \langle \boxdot, C_A\rangle \qquad \dfrac{\Gamma : \langle B \div C_B, 0\rangle \xrightarrow{0} \langle \boxtimes, 0\rangle \; \Gamma : \langle C \div C_C, 0\rangle \xrightarrow{0} \langle \boxtimes, 0\rangle \; \Gamma : \langle C_A, 0\rangle \xrightarrow{C_A} \langle \boxdot, 0\rangle}{\Gamma : \langle B \div C_B | C \div C_C, C_A\rangle \xrightarrow{C_A} \langle \boxtimes, 0\rangle}\,(PAR-C1)}{\Gamma : \langle A \div C_A; (B \div C_B | C \div C_C), 0\rangle \xrightarrow{A;C_A} \langle \boxtimes, 0\rangle}\,(SEC-C)}{\Gamma : \{|A \div C_A; (B \div C_B | C \div C_C)|\} \xrightarrow{A;C_A} \boxtimes}\,(SAGA-TOP)$$

Figure 2.9: Example traces of the saga parallel composition

Chapter 3

The *SC* family of process calculi

In this Chapter we introduce the Signal Calculus (*SC*). The *SC* calculus has been designed and developed to provide formal machineries supporting service oriented applications. Our main goal is to provide a small set of basic primitives to drive the implementation of a middleware (See Chapter 6) tailored to program service coordination policies. However, the *SC* programming paradigm is suitable to deal with other contexts, such as grid and clouds computing [58].

The starting point of our work is the event notification paradigm. The building blocks of *SC* are called *components*, which interact by issuing/reacting to *events*. A component represents a "simple" service interacting through an asynchronous signal passing mechanism, inspired from the asynchronous π-calculus [30]. Components are the basic computational units: they perform internal operations and can be composed and distributed over a network. Each component is identified by unique *name*, which, intuitively, can be through of as the URI of the published service.

The adoption of the event notification paradigm, for managing service coordination has two main advantages

- it is a well known programming model
- it permits the distribution of coordination activities and of the underlying computational infrastructure.

In the literature event based architectures can be classified as *brokered* and *non-brokered* [29]. The two approaches differ for the way they notify events. The

brokered pattern exploits a centralized support that is responsible for implementing the subscriptions and the notification forwarding. Such solution is closely related to the ideas of tuple space based systems [59]. Instead, the non-brokered approach suggests that each component acts both as publisher or subscriber for other components and the delivering of signals is implemented through *peer-to-peer* like structures. The *SC* calculus adopts the non-brokered approach since it better fits with service oriented paradigm.

In the following we assume as given the set of names of the components involved into a system with no assumption on the mechanisms adopted to retrieve them (e.g. UDDI service directories [60]).

Communications can be performed by the rising and the handling of events. They are tagged with a *meta type* representing the class of events they belong to. Such *meta type* information is often referred to, in the literature (e.g. [61]), with the term *topic*.

Each component provides an *event flow*, namely the collection of component names that must be notified about the emitted events. Hence, flows define the component view of the coordination policies. A key feature of *SC* is that it provides a multicast notification structure. When an event is raised by a component, several envelopes are generated to notify all components in the flow. Each envelope contains the event itself and the address of the target component. Envelopes are called *signals*.

Each component own a set of signal handlers associated to topics. This handlers, called *reactions*, are responsible of the management of the reception of an event notification. Indeed, the reception of a signal acts like a trigger that activates the execution of a new behavior described by the compatible reaction within the component.

The *SC* component *interface* is defined by its *reactions* and *flows*. The language primitives allow one to *dynamically* modify the topology of the coordination by adding new flows and reactions to components, namely the component interface can by updated at run-time. Finally, components are structured to build a *network* of services. A network provides the facility to transport signals containing the events exchanged among components.

The structure of the chapter is organized as follows: In Section 3.1 we present the syntax and the reduction semantics of the simplest version of the calculus. Our goal is to introduce smoothly the main concepts of *SC*. In Section 3.2 we extend the syntax and the semantics of the calculus to manage sessions. We also provide some examples to highlight the expressiveness of the resulting calculus. In Section 3.3 we present a type-based approach to event notification. We equip our calculus with simple types for events. Intuitively, types drive the coordination of components. We conclude in Section 3.4 by reviewing the distinguished

$R ::= 0 \mid \tau \gg B \mid R_1 | R_2$

(a) Reactions

$F ::= 0 \mid \tau \rightsquigarrow \vec{a} \mid F | F$

(b) Flows

$B ::= 0 \mid \texttt{rupd}(R); B \mid \texttt{fupd}(F); B \mid \texttt{out}\langle\tau\rangle; B \mid \varepsilon; B \mid B \mid B$

(c) Behaviors

$N ::= \emptyset \mid a[B]_F^R \mid N \parallel N \mid \langle\tau\rangle @ a$

(d) Networks

Table 3.1: *SC* syntax

features of *SC* comparing our calculus with other proposals.

3.1 Signal Calculus: *SC*

The Signal Calculus (*SC*) is a process calculus specifically designed to describe coordination of services distributed over a network. We start introducing the syntax of the calculus together with some notational machinery. We assume a finite set of topics ranged over by $\tau_1, ..., \tau_k \in \mathcal{T}$ and a finite set of component names ranged by $a, b, c... \in \mathcal{A}$. We use $\vec{a}$ to denote a set of names $a_1, ..., a_n$.

The calculus is centered around the notion of *component*, written as $a[B]_F^R$ and representing a service uniquely identified by a name a, the public address of the service, having internal behavior B, reaction R and outgoing connection F called *flow*.

Reactions (R) are described by the syntax in Table 3.1a. A reaction is a multiset ($R_1|R_2$), possibly empty (0), of unit reactions. A unit reaction $\tau \gg B$ triggers the execution of the behavior B upon reception of a signal tagged by the topic τ. The syntax of behaviors will be given later. The reactions R_1 and R_2 are called *sub-reactions* of the reaction composition $R_1|R_2$.

Each component has a flow (Table 3.1b) that describes addresses of events. A flow is a set ($F_1|F_2$), possibly empty (0), of unit flows. A unit flow $\tau \rightsquigarrow \vec{a}$ describes the set of component names $\vec{a}$ where raised events having τ as topic have to be delivered.

Reactions and flows are defined up-to a structural congruence ($\equiv$). Indeed,

we assume that $(F, |, 0)$ and $(R, |, 0)$ are commutative monoids, meaning that:

$$F|0 \equiv F \quad F|F' \equiv F'|F \quad F|(F'|F'') \equiv (F|F')|F''$$
$$R|0 \equiv R \quad R|R' \equiv R'|R \quad R|(R'|R'') \equiv (R|R')|R''$$

Moreover we assume that $\tau \rightsquigarrow \vec{a}|\tau \rightsquigarrow \vec{b} \equiv \tau \rightsquigarrow \vec{a} \cup \vec{b}$. Notice that this assumptions implies that $F|F \equiv F$. This structural congruence allows us to freely rearrange reactions and flow.

Component behaviors (B) are defined in Table 3.1c. Intuitively, the syntax of behaviors defines the programming interface that the middleware should provide. A reaction update $\texttt{rupd}(\tau);B$ extends the reaction part of the component interface, providing the ability to change the reactive part of a component. Similarly, a flow update $\texttt{fupd}(F);B$ extends the component flows. An asynchronous event emission $\texttt{out}\langle\tau\rangle;B$ first spawns into the network a set of envelopes containing the event, one for each component name declared in the flow having topic τ, and then activates B. The behavior $\varepsilon;B$ represent an internal action performed by the component (at the end of its execution, the component activate the continuation behavior B). Finally, the inactive behavior 0 and the parallel composition $B \mid B$ have the standard meanings. We assume that $(B, \mid, 0)$ is a commutative monoid, allowing to rearrange the parallel composition. We omit the trailing occurrences of 0.

Networks (N) describe the distribution of components and carry signals exchanged among them. Networks are defined in Table 3.1d. A component is defined as $a[B]^R_F$. A signal envelope $\langle s\rangle@a$ describes a message containing the signal having the topic τ whose target component is the component named a. The empty network $\emptyset$ and the parallel composition have the standard meanings. In the following we will use $\prod_{b_i \in \vec{b}} \langle\tau\rangle@b_i$, with $\vec{b}$ a finite set of component names, to represent the parallel composition of messages having topic τ. Hereafter, we assume that components are uniquely identified by their names.

Definition 2 *A* SC *network is well formed if the names of the components it declares are all different.*

3.1.1 Reduction Semantics

SC operational semantics is defined in the reduction style and states how components, at each step, communicate and update their interface. The reduction relation over *SC* networks ($\rightarrow$) is defined by the rules in Table 3.3. We also introduce a *projection* function $F \downarrow_\tau$ that returns the set of component names linked by the flow F through the topic τ. The projection is defined in Table 3.2.

$$0 \downarrow_\tau = \emptyset \qquad \tau \rightsquigarrow \vec{a} \downarrow_\tau = \vec{a} \qquad \tau' \rightsquigarrow \vec{a} \downarrow_\tau = \emptyset \qquad F|F' \downarrow_\tau = F \downarrow_\tau \cup F' \downarrow_\tau$$

Table 3.2: *SC* flow projection function

Rule (SKIP) describes the execution of an internal action, namely an action that has no side effect on the system. Rule (RUPD) extends the component reactions with a further unit reaction (the parameter of the primitive). Rule (FUPD) extends the component flows with a unit flow. Rule (OUT) first takes the set of component names $\vec{a}$ that are linked to the component for the conversation topic τ and then spawns into theEach client can raise its offer or ignore the demand. network an envelope for each component name in the set. This rule implements the multicast feature of *SC*. Rule (IN) allows a signal envelope to react with the component whose name is specified inside the envelope. Notice that signal emission rule (OUT) and signal receiving rule (IN) do not consume, respectively, the flow and the reaction of the component. Namely, flows and reactions are *persistent*. In Section 3.1.2 we describe how exploit these features to implement a simple form of recursion using *SC*. Finally, rules (STRUCT) and (PAR) are standard. In the following, we use $N \rightarrow^* N_1$ to represent a network N that is reduced to N_1 after a finite number of steps.

3.1.2 Examples

We now present some examples to better explain the *SC* programming model. These examples focus on the management of the control flow of composition of components. For this reason, we will model behaviors that does not affect the coordination via the internal action ε.

Multicast notifications

The following example highlights how the multicast notification of *SC* simplifies the design of component coordination. Let us consider a component s that requires a set of resources to provide a certain functionality. This component is exploited by several clients c_i, with $i = 1,..,n$, to achieve a common goal. Namely, all clients collaborate to permit the activation of the service supplied by s, providing the required resources. The process is summarized as follows:

1. the bid phase starts, the service notifies to all clients its requirement of

$$\frac{}{a\,[\varepsilon; B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R}\ (SKIP)$$

$$\frac{}{a\,[\mathtt{rupd}\,(R_1); B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^{R|R_1}}\ (RUPD)$$

$$\frac{}{a\,[\mathtt{fupd}\,(F_1); B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_{F|F_1}^R}\ (FUPD)$$

$$\frac{F \downarrow_\tau = \vec{b}}{a\,[\mathtt{out}\,\langle\tau\rangle; B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R \parallel \prod_{b_i \in \vec{b}} \langle\tau\rangle @ b_i}\ (OUT)$$

$$\frac{}{\langle\tau\rangle @ a \parallel a\,[B_1]_F^{R|\tau \gg B_2} \rightarrow a\,[B_1 \mid B_2]_F^{R|\tau \gg B_2}}\ (IN)$$

$$\frac{N \rightarrow N_1}{N \parallel N_2 \rightarrow N_1 \parallel N_2}\ (PAR) \qquad \frac{N \equiv N_1 \rightarrow N_2 \equiv N_3}{N \rightarrow N_3}\ (STRUCT)$$

Table 3.3: *SC* reduction rules

resources,

2. each client can raise its offer of resources or ignore the service demand,
3. if no client responds to the service demand, the bid fails and the functionality is not provided,
4. if the service receives a sufficient amount of resources the bid phase terminates, otherwise it restarts,
5. if the bid phase successes, the service activates its functionality.

The network N in Figure 3.1 illustrates the instance of this scenario where three clients are involved. We use the topics τ_o to represent the bid of a new resource from a client. The occurrence of an event τ_r represents that the resources acquired by the service has changed, but the required amount has not been reached. We do not model quantitative aspects related to the requirements

$$
\begin{array}{ll}
N = & s\,[\mathtt{out}\langle\tau_r\rangle]^{\tau_o \gg \mathtt{out}\langle\tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \parallel C(1) \parallel C(2) \parallel C(3) \\
\textit{where } C(i) = & c_i\,[0]^{\tau_r \gg \mathtt{out}\langle\tau_o\rangle \mid \tau_r \gg 0}_{\tau_o \rightsquigarrow \{s\}}
\end{array}
$$

Figure 3.1: *SC* components that collaborate to supply a resource set

of the service and the availabilities of the clients. We also abstract from the functionality supplied, which is simply modeled as a behavior B. Initially, the service issues an event to notify its demand of resources. The component s uses the projection function to discover the three resource providers and then spawns into the network the three corresponding envelopes:

$$
\frac{(\tau_r \rightsquigarrow \{c_1,c_2,c_3\})\downarrow_{\tau_r} = \{c_1,c_2,c_3\}}{s\,[\mathtt{out}\langle\tau_r\rangle]^{\tau_o \gg \mathtt{out}\langle\tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \rightarrow s\,[0]^{\tau_o \gg \mathtt{out}\langle\tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \parallel \langle\tau_r\rangle@c_1 \parallel \langle\tau_r\rangle@c_2 \parallel \langle\tau_r\rangle@c_3}
$$

Upon the reception of a resource request, a client non-deterministically activates one of its two reactions. The execution of $\tau_r \gg 0$ models that the client don't want offer resources, so it simply consume the received envelope. The execution of $\tau_r \gg \mathtt{out}\langle\tau_o\rangle$ represents that the client is able to offer a resource. Anyhow, the client reactions are not consumed and persist on its interface:

$$
c_i\,[0]^{\tau_r \gg \mathtt{out}\langle\tau_o\rangle \mid \tau_r \gg 0}_{\tau_o \rightsquigarrow \{s\}} \parallel \langle\tau_r\rangle@c_i \rightarrow c_i\,[\mathtt{out}\langle\tau_o\rangle]^{\tau_r \gg \mathtt{out}\langle\tau_o\rangle \mid \tau_r \gg 0}_{\tau_o \rightsquigarrow \{s\}}
$$

Client raise events tau_o to notify their agreement to provide a resource. Since each client is connected only to the component s, the envelope spawned is only one:

$$
\frac{(\tau_o \rightsquigarrow \{s\})\downarrow_{\tau_r} = \{s\}}{c_i\,[\mathtt{out}\langle\tau_o\rangle]^{\tau_r \gg \mathtt{out}\langle\tau_o\rangle \mid \tau_r \gg 0}_{\tau_o \rightsquigarrow \{s\}} \rightarrow c_i\,[0]^{\tau_r \gg \mathtt{out}\langle\tau_o\rangle \mid \tau_r \gg 0}_{\tau_o \rightsquigarrow \{s\}} \parallel \langle\tau_o\rangle@s}
$$

Upon the reception of a resource bid, the service non-deterministically activates one of its two reactions. The execution of $\tau_o \gg B$ models that the required amount of resources has been reached, then the functionality B can be provided:

$$
s\,[0]^{\tau_o \gg \mathtt{out}\langle\tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}} \parallel \langle\tau_o\rangle@s \rightarrow s\,[B]^{\tau_o \gg \mathtt{out}\langle\tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3\}}
$$

The execution of $\tau_o \gg \mathtt{out}\langle\tau_r\rangle$ represents that the service still need more resources. It forwards a new request to the clients, restarting the bid phase. Now,

$$N_{seq} \triangleq e\,[B_e]^{f \gg \mathtt{out}\langle f\rangle}_{f \rightsquigarrow a} \parallel a\,[B_a]^{f \gg \varepsilon;\mathtt{out}\langle f\rangle}_{f \rightsquigarrow b} \parallel b\,[B_b]^{f \gg \varepsilon;\mathtt{out}\langle f\rangle}_{F_b}$$

Figure 3.2: *SC* implementation of the BPMN design 2.2b

assume that the involved clients are four. The model of the service s must be substituted with

$$s\,[\mathtt{out}\langle \tau_r\rangle]^{\tau_o \gg \mathtt{out}\langle \tau_r\rangle \mid \tau_o \gg B}_{\tau_r \rightsquigarrow \{c_1,c_2,c_3,c_4\}}$$

Notice that only the flows of the component has been changed, connecting the service to the new component c_4. In fact, the *SC* multicast notification of events and the component flows permits hide communication complexity and permit to program the behavior of the service independently from the number of involved clients.

Sequential composition

We now provide an implementation of the BPMN business process 2.2b. We map each BPMN task to a distinguished *SC* component. Notice that this is an arbitrary choice and that is not prescribed by the BPMN model, since BPMN does not regard the distribution of task over the network. Moreover, we must directly implement the communications and synchronization required to guarantee the dependencies of BPMN flow. Clearly, the *SC* programming model is less abstract than the BPMN one. We now provide a naive implementation of the business process. The mapping from BPMN designs (formally saga processes) to *SC* models will be given in Section 4.3.

We implement each BPMN task A of the business process via an *SC* component a_A. Without loss of generality, we assume that task names are all different. This will also ensure that the corresponding network is well formed. The forward-flow is propagated through *forward* events, which are signals having topic f. When a component that implements a BPMN task receives a *forward* signal, it assumes that all previous stages have been completed. Hence, the component starts its internal computation and, upon termination, raises a *forward* event to inform other components. We also adopt a special component e to implement the entry point of the business process. The *SC* network N_{seq} in Figure 3.2 represents the business process depicted in Figure 2.2b.

The *SC* component e simply implements the entry point of the business process. It waits for the request of activation via a notification of a *forward* event.

The component simply delivers the notification to the components in the business process that have no dependencies. Hence, the flow is $f \rightsquigarrow a$.

The *SC* components a and b implement the BPMN tasks A and B, respectively. The behavior of these components is straightforward. These components wait for the termination of all their dependencies ($f \succcurlyeq \ldots$), perform an internal action (ε), representing the execution of the corresponding task, and finally notify their termination ($\mathtt{out}\langle f \rangle$). We remark that we focus on the managing of the control flow and that we do not deal with the action performed by the BPMN tasks. Hence, we assume that the a BPMN task does not have any side effect and we model it as an internal action. Notice that the component a and b change only for their flows. This allow us to highlight how component can be exploited as building block that are composed into complex coordination using their peer to peer flow structure.

Finally, the active behaviors B_e, B_a and B_b represent active threads executed inside the component,

In order to have a snapshot of the execution, we send a request to the entry point e of the business process. Namely, we spawn into the network an envelope $\langle f \rangle @ e$. The entry point can consume the pending envelope, activating the proper behavior:

$$\frac{}{\langle f \rangle @ e \parallel e\,[B_e]_{f \rightsquigarrow a}^{f \succcurlyeq \mathtt{out}\langle f \rangle} \rightarrow e\,[B_e \mid \mathtt{out}\langle f \rangle]_{f \rightsquigarrow a}^{f \succcurlyeq \mathtt{out}\langle f \rangle}} \; (IN)$$

Now, the entry point raises an *forward* event ($\mathtt{out}\langle f \rangle$) to notify the request to the next task in the forward-flow. The component examines its flows (($f \rightsquigarrow \{a\}$) $\downarrow_f$) to discover its subscribers ($\{\mathrm{a}\}$) and then spawn into the network the corresponding envelopes:

$$\frac{(f \rightsquigarrow \{a\}) \downarrow_f = \{a\}}{e\,[B_e \mid \mathtt{out}\langle f \rangle]_{f \rightsquigarrow a}^{f \succcurlyeq \mathtt{out}\langle f \rangle} \rightarrow e\,[B_e]_{f \rightsquigarrow a}^{f \succcurlyeq \mathtt{out}\langle f \rangle} \parallel \langle f \rangle @ a} \; (OUT)$$

The component a can consume the envelope. Notice that this kind of notification characterizes that all previous stages of the business process have completed their executions. Since the component a has no predecessor, the envelope signals that the work-flow started:

$$\frac{}{\langle f \rangle @ a \parallel a\,[B_a]_{f \rightsquigarrow b}^{f \succcurlyeq \varepsilon;\mathtt{out}\langle f \rangle} \rightarrow a\,[B_a \mid \varepsilon;\mathtt{out}\langle f \rangle]_{f \rightsquigarrow b}^{f \succcurlyeq \varepsilon;\mathtt{out}\langle f \rangle}} \; (IN)$$

The internal action (ε) performed by the component a represents the execution of the BPMN task A. Notice that the execution of an internal action preserves the

overall structure of the network:

$$\frac{}{a\,[B_a \mid \varepsilon;\mathtt{out}\langle f\rangle]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{f\rightsquigarrow b} \longrightarrow a\,[B_a \mid \mathtt{out}\langle f\rangle]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{f\rightsquigarrow b}}\ (SKIP)$$

After the termination of the task A the component raises a *forward* event, to inform the other components involved in the business process about its successful termination. As usual, the event notification exploits the component flow to discover the actual subscribers:

$$\frac{(f \rightsquigarrow \{b\})\downarrow_f = \{b\}}{a\,[B_a \mid \mathtt{out}\langle f\rangle]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{f\rightsquigarrow b} \longrightarrow a\,[B_a]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{f\rightsquigarrow b} \parallel \langle f\rangle@b}\ (OUT)$$

The component b can consume the pending notification,

which ensures that all previous stages of the work-flow (A) have completed their execution:

$$\frac{}{\langle f\rangle@b \parallel b\,[B_b]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{F_b} \longrightarrow b\,[B_b \mid \varepsilon;\mathtt{out}\langle f\rangle]^{f\gg\varepsilon;\mathtt{out}\langle f\rangle}_{F_b}}\ (IN)$$

Finally, the component b executes the internal action corresponding to the BPMN task B. Notice that the component raises an event to notify its termination. This notification will be delivered according to the flow of the component (F_b).

Consumer and Producer

Let us assume that a producer p and a consumer c have access to a shared data space. The consumer c can get resource only after the producer p has produced it. This design pattern can be suitably applied in a wide range of systems, including SOA multi-threaded applications, Grid and Cloud computing.

The problem can be specified as displayed in Figure 3.3a. The component p starts its execution performing the (internal) behavior that modifies the state of the data space that has to be read by c. When the data have been modified, the producer p raises an event ($\mathtt{out}\langle produced\rangle$) in order to inform c that the required resources are now available. Upon the notification of the event, the consumer c automatically starts its execution and takes the resource in the data space performing its internal behavior. After its termination, the consumer raises a *response* event ($\mathtt{out}\langle consumed\rangle$) in order to inform p that it can produce a new resource. As usual, we omit to model the internal actions performed by the components to produce and consume the data. These operations does not affect the coordination and then can be modeled as internal actions ε.

$$N_p \triangleq \quad p\,[\varepsilon;\mathtt{out}\langle produced\rangle]^{consumed \gg \varepsilon;\mathtt{out}\langle produced\rangle}_{produced \rightsquigarrow c}$$

$$N_c \triangleq \quad c\,[0]^{produced \gg \varepsilon;\mathtt{out}\langle consumed\rangle}_{consumed \rightsquigarrow p}$$

$$Net \triangleq \quad N_p \parallel N_c$$

(a) Statically linked components

$$N'_p \triangleq \quad p \left[\begin{array}{l} \mathtt{fupd}\,(produced \rightsquigarrow c)\,; \\ \mathtt{rupd}\,(consumed \gg \varepsilon;\mathtt{out}\langle produced\rangle)\,; \\ \varepsilon;\mathtt{out}\langle produced\rangle \end{array}\right]^0_0$$

$$N'_c \triangleq \quad c\,[\mathtt{fupd}\,(consumed \rightsquigarrow p)\,;\mathtt{rupd}\,(produced \gg \varepsilon;\mathtt{out}\langle consumed\rangle)]^0_0$$

$$Net' \triangleq \quad N'_p \parallel N'_c$$

(b) Components connecting at the start up phase

Figure 3.3: Producer and consumer in *SC*

We provide an outline of the execution of the example given in Figure 3.3a. As a shorthand, we write τ_p for the topic *produced* and τ_c for *consumed*. Initially, only the component p contains an active behavior. It emits the event τ_p, spawning into the network an envelope for the consumer c:

$$\frac{(\tau_p \rightsquigarrow c)\downarrow_{\tau_p} = \{c\}}{p\,[\varepsilon;\mathtt{out}\langle\tau_p\rangle]^{\tau_c \gg \varepsilon;\mathtt{out}\langle\tau_p\rangle}_{\tau_p \rightsquigarrow c} \longrightarrow^* p\,[0]^{\tau_c \gg \varepsilon;\mathtt{out}\langle\tau_p\rangle}_{\tau_p \rightsquigarrow c} \parallel \langle\tau\rangle@c}$$

The envelope can be handled by the consumer, activating the behavior of the corresponding reaction:

$$\langle\tau\rangle@c \parallel c\,[0]^{\tau_p \gg \varepsilon;\mathtt{out}\langle\tau_c\rangle}_{\tau_c \rightsquigarrow p} \longrightarrow c\,[\varepsilon;\mathtt{out}\langle\tau_c\rangle]^{\tau_p \gg \varepsilon;\mathtt{out}\langle\tau_c\rangle}_{\tau_c \rightsquigarrow p}$$

In a similarly way, the consumer c sends an envelope to the producer p, thus activating the proper internal behavior:

$$p\,[0]^{\tau_c \gg \varepsilon;\mathtt{out}\langle\tau_p\rangle}_{\tau_p \rightsquigarrow c} \parallel c\,[\varepsilon;\mathtt{out}\langle\tau_c\rangle]^{\tau_p \gg \varepsilon;\mathtt{out}\langle\tau_c\rangle}_{\tau_c \rightsquigarrow p} \longrightarrow^* p\,[\varepsilon;\mathtt{out}\langle\tau_p\rangle]^{\tau_c \gg \varepsilon;\mathtt{out}\langle\tau_p\rangle}_{\tau_p \rightsquigarrow c} \parallel c\,[0]^{\tau_p \gg \varepsilon;\mathtt{out}\langle\tau_c\rangle}_{\tau_c \rightsquigarrow p}$$

The producer-consumer example given above exploits a static linkage. However, dynamic configuration can be easily handled with our framework. For example, the consumer and the producer can be dynamically linked together (e.g.

$a\left[\mathtt{out}\langle\tau_+\rangle\right]^{\tau_+ \gg B_1 \mid \tau_+ \gg B_2}_{\tau_+ \rightsquigarrow a}$

(a) Non-deterministic choice

$a\left[\mathtt{out}\langle\tau_!\rangle\right]^{\tau_! \gg \mathtt{out}\langle\tau_!\rangle \mid B}_{\tau_! \rightsquigarrow a}$

(b) Unbound replication of B

Figure 3.4: *SC* encoding of primitives

at the start up phase) using the reaction update and flow update primitives (Figure 3.3b). Now, both components have an active internal behavior. The producer can update its flows by adding the link to the consumer for signals of topic τ_p, as follows:

$$p\left[\begin{array}{l}\mathtt{fupd}\left(\tau_p \rightsquigarrow c\right);\\ \mathtt{rupd}\left(\tau_c \gg \varepsilon; \mathtt{out}\langle\tau_p\rangle\right);\\ \varepsilon; \mathtt{out}\langle\tau_p\rangle\end{array}\right]^0_0 \rightarrow p\left[\begin{array}{l}\mathtt{rupd}\left(\tau_c \gg \varepsilon; \mathtt{out}\langle\tau_p\rangle\right);\\ \varepsilon; \mathtt{out}\langle\tau_p\rangle\end{array}\right]^0_{\tau_p \rightsquigarrow c}$$

Then we apply the reduction rule for the reaction update of the producer:

$$p\left[\mathtt{rupd}\left(\tau_c \gg \varepsilon; \mathtt{out}\langle\tau_p\rangle\right); \varepsilon; \mathtt{out}\langle\tau_p\rangle\right]^0_{\tau_p \rightsquigarrow c} \rightarrow p\left[\varepsilon; \mathtt{out}\langle\tau_p\rangle\right]^{\tau_c \gg \varepsilon; \mathtt{out}\langle\tau_p\rangle}_{\tau_p \rightsquigarrow c}$$

After these reductions the producer component has created a link to the consumer for signals of topic τ_p and can receive signals of topic τ_c. In a similar way, the consumer updates its reactions and flows.

Non-deterministic choice

The *SC* calculus is not directly equipped with an operator to handle non-deterministic choice. However, it can be model as reported in Figure 3.4a. Assume that the topic τ_+ is only known by the component a, which notifies the topic to itself:

$$a\left[\mathtt{out}\langle\tau_+\rangle\right]^{\tau_+ \gg B_1 \mid \tau_+ \gg B_2}_{\tau_+ \rightsquigarrow a} \rightarrow a\left[0\right]^{\tau_+ \gg B_1 \mid \tau_+ \gg B_2}_{\tau_+ \rightsquigarrow a} \parallel \langle\tau_+\rangle @ a$$

Now, both reactions are able to consume the spawned envelope $\langle\tau_+\rangle @ a$. Operationally, this results in a non-deterministic choice; only one of them will be activated.

Replication

SC does not provide a primitive to handle unbound replication of a behavior (e.g. the π-calculus $!G$ construct). This feature can be encoded exploiting the

persistence of reactions (the reduction rule (IN) does not consume the reaction that is involved in the communication). For instance, the component depicted in Figure 3.4b replicates the behavior B. We have the following reductions:

$$
\begin{array}{lll}
a\,[\mathtt{out}\langle\tau_!\rangle]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} & \longrightarrow a\,[0]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} \parallel \langle\tau_!\rangle@a & \longrightarrow \\
a\,[\mathtt{out}\langle\tau_!\rangle \mid B]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} & \longrightarrow a\,[B]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} \parallel \langle\tau_!\rangle@a & \longrightarrow \\
a\,[\mathtt{out}\langle\tau_!\rangle \mid B \mid B]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} & \longrightarrow a\,[B \mid B]^{\tau_!\gg\mathtt{out}\langle\tau_!\rangle\ \mid B}_{\tau_!\rightsquigarrow a} \parallel \langle\tau_!\rangle@a & \longrightarrow^*
\end{array}
$$

3.2 Managing Sessions

The *SC* calculus allows modeling a wide range of coordination policies for service-oriented applications. However, high-level abstractions for programming such policies are still required. In particular, information associated to signals is not structured and topics cannot be created dynamically. Furthermore, the notion of session is missing: components cannot keep track of concurrent event notifications. We now introduce a first extension of the basic *SC* calculus to manage sessions. Sessions are a sort of "virtual communication links" among publishers and subscribers. In our approach, a session identifies the scope within an event is significant: partners that are not in this scope cannot react to events of the session. Furthermore, our session handling mechanism can deal with multi-party sessions in a natural way.

First, we extend the notion of events. Events become pairs including a topic and a session. We extend the syntax of behaviors modifying the signal emission primitive and adding the capability to generate new topics (see Table 3.4a) The *signal emission* $\mathtt{out}\langle\tau_\circledcirc\tau'\rangle$ raises an event. Now the event is characterized by the topic τ and the session identifier τ'. Notice that both topics and sessions are names and are freely interchangeable. Moreover, topics and sessions can be dynamically created using the *generation* primitive $(\nu\tau)B$.

The reactive parts of the component interface (reactions) are extended to handle this new notion of events (see Table 3.4b) A *lambda reaction* $\tau\,\lambda\,\tau' \gg B$ handles all signals with topic τ, regardless of their session. In the behavior B, τ' is bound by the lambda reaction. A *check reaction* $\tau_\circledcirc\tau' \gg B$ can handle only signals having the topic τ issued for the session τ' and does not declare bound names. For simplicity, the syntax of flows has not modified (Table 3.4c), components declares their connections filtering on the topics of events. This implies that the topology of connections cannot be specialized for a specific session. Nevertheless, extending the calculus with flows specialized on sessions is straightforward.

Table 3.4d describes the syntax of networks. The *envelope* $\langle\tau_\circledcirc\tau'\rangle@a$ now

$$B ::= \quad 0 \mid \varepsilon;B \mid B \,|\, B \mid \mathtt{rupd}(R);B \mid \mathtt{fupd}(F)\,B;B \mid \mathtt{out}\langle \tau_{@}\tau' \rangle;B \mid (\nu\tau)B$$

(a) Behaviors

$$R ::= 0 \mid R|R \mid \tau\ \lambda\ \tau' \gg B \mid \tau_{@}\tau' \gg B \qquad F ::= 0 \mid \tau \rightsquigarrow \vec{a} \mid F|F$$

(b) Reactions (c) Flows

$$N ::= \quad \emptyset \mid a[B]^R_F \mid N \parallel N \mid \langle \tau_{@}\tau' \rangle @ a \mid (\nu n)N \qquad \textit{where } n \in \mathcal{A} \cup \mathcal{T}$$

(d) Networks

Table 3.4: *SC* syntax to handle sessions

$fn((\nu\tau)B) = fn(B) \setminus \{\tau\}$	$bn((\nu\tau)B) = bn(B) \cup \{\tau\}$
$fn(\tau\ \lambda\ \tau' \gg B) = fn(B) \setminus \{\tau'\} \cup \{\tau\}$	$bn(\tau\ \lambda\ \tau' \gg B) = bn(B) \cup \{\tau'\}$
$fn((\nu n)N) = fn(N) \setminus \{n\}$	$bn((\nu n)N) = bn(N) \cup \{n\}$

Table 3.5: *SC* free and bound names

carries both the topic τ and the session τ'. We also introduce a primitive to generate names over a network. Indeed, the restriction $(\nu n)N$ provides a scoping mechanism for names of components, of sessions and of topics. Notice that, since component names cannot be communicated, the restriction of a component allows to hide behavior of part of a network.

Free (fn) and bound (bn) names are defined in the standard way. We require that for each term the intersection between free and bound names is empty. The rules for terms containing binders are given in Table 3.5.

The structural congruence over reactions, flows, behaviors and networks is the smallest congruence relation that satisfies the laws given in Table 3.6. Notice that the first statements guarantee that $(R, |, 0)$, $(F, |, 0)$, $(B, \mid, 0)$ and $(N, \|, \emptyset)$ are commutative monoids.

3.2.1 Reduction Semantics

The reduction relation $\rightarrow$ is depicted in Table 3.7. The intuitive interpretation of the reduction rules is straightforward. Rules *CHECK* and *LAMBDA* describe the

$$
\begin{array}{ccc}
0|R \equiv R & R_1|R_2 \equiv R_2|R_1 & R_1|(R_2|R_3) \equiv (R_1|R_2)|R_3 \\
0|F \equiv F & F_1|F_2 \equiv F_2|F_1 & F_1|(F_2|F_3) \equiv (F_1|F_2)|F_3 \\
0 \mid B \equiv B & B_1 \mid B_2 \equiv B_2 \mid B_1 & B_1 \mid (B_2 \mid B_3) \equiv (B_1 \mid B_2) \mid B_3 \\
\emptyset \parallel N \equiv N & N_1 \parallel N_2 \equiv N_2 \parallel N_1 & N_1 \parallel (N_2 \parallel N_3) \equiv (N_1 \parallel N_2) \parallel N_3
\end{array}
$$

$$
\tau \rightsquigarrow \vec{a} | \tau \rightsquigarrow \vec{b} \equiv \tau \rightsquigarrow \vec{a} \cup \vec{b}
$$

$$
\begin{array}{cccc}
(\nu\tau)0 \equiv 0 & (\nu\tau)(\nu\tau')B \equiv (\nu\tau')(\nu\tau)B & ((\nu\tau)B) \mid B' \equiv (\nu\tau)(B \mid B') & \text{if } \tau \notin fn(B') \\
(\nu n)\emptyset \equiv \emptyset & (\nu n)(\nu n')N \equiv (\nu n')(\nu n)N & ((\nu n)N) \parallel N' \equiv (\nu n)(N \mid N') & \text{if } n \notin fn(N') \\
\multicolumn{4}{c}{(\nu\tau)a\,[B]_F^R \equiv a\,[(\nu\tau)B]_F^R \qquad \text{if } \tau \notin \{a\} \cup fn(F) \cup fn(R)}
\end{array}
$$

Table 3.6: *SC* structural congruence laws

activation of check reactions, that require the exact match of the session identifier, and of lambda reactions, receiving the session identifier as argument. Notice that the reduction relation permits to consume check reactions and to maintain lambda reactions installed. Informally, a lambda reaction can be used by a service to publish a public interface that establishes a session with the consumer. Instead, a check reaction permits a service to handle only signals that belong to a given session. For example, if the session is used to identify an instance of a work-flow, this mechanism allows the service to specialize its behavior for each instance and to track the progress of the control flow.

3.2.2 Examples

Private events

We have already pointed out in Sections 3.1.2 and 3.1.2 how the non-deterministic choice and replication can be naively encoded in *SC*. Figure 3.5 illustrates a more general approach.

A component named a can exploit the $+_a$ primitive to represent a non-deterministic choice. The restriction over the name τ_+ ensures that the topic used by the encoding is fresh, meaning that no other component can initially know this name. The constraint $\tau_+ \notin fn(B_1) \cup fn(B_2)$ guarantees that the encoding will never be used or communicated later.

$$\frac{}{a\,[\varepsilon;B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R}\ (SKIP)$$

$$\frac{}{a\,[\mathtt{rupd}\,(R_1);B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^{R|R_1}}\ (RUPD)$$

$$\frac{}{a\,[\mathtt{fupd}\,(F_1);B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_{F|F_1}^R}\ (FUPD)$$

$$\frac{F \downarrow_\tau = \vec{b}}{a\left[\mathtt{out}\langle \tau \circledcirc \tau' \rangle;B_1 \mid B_2\right]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R \parallel \prod_{b_i \in \vec{b}} \langle \tau \circledcirc \tau' \rangle @ b_i}\ (OUT)$$

$$\frac{N \rightarrow N_1}{N \parallel N_2 \rightarrow N_1 \parallel N_2}\ (PAR) \qquad \frac{N \equiv N_1 \rightarrow N_2 \equiv N_3}{N \rightarrow N_3}\ (STRUCT)$$

$$\frac{}{\langle \tau \circledcirc \tau' \rangle @ a \parallel a\,[B_1]_F^{R|\tau \circledcirc \tau' \gg B_2} \rightarrow a\,[B_1 \mid B_2]_F^R}\ (CHECK)$$

$$\frac{}{\langle \tau \circledcirc \tau' \rangle @ a \parallel a\,[B_1]_F^{R|\tau \circledcirc \tau'' \gg B_2} \rightarrow a\left[B_1 \mid \{\tau'/\tau''\}B_2\right]_F^{R|\tau \circledcirc \tau'' \gg B_2}}\ (LAMBDA)$$

$$\frac{N \rightarrow N'}{(\nu n)N \rightarrow (\nu n)N'}\ (NEW)$$

Table 3.7: *SC* semantic rules

$$B_1 +_a B_2 \triangleq \quad (\nu\tau_+)\left(\begin{array}{l}\texttt{rupd}(\tau_+ \circledcirc \tau_+ \gg B_1 | \tau_+ \circledcirc \tau_+ \gg B_2);\\ \texttt{fupd}(\tau_+ \rightsquigarrow a);\\ \texttt{out}\langle \tau_+ \circledcirc \tau_+ \rangle\end{array}\right)$$

$$!_a B \triangleq \qquad (\nu\tau_!, \tau)\left(\begin{array}{l}\texttt{rupd}(\tau_! \ \lambda\, \tau \gg \texttt{out}\langle \tau_! \circledcirc \tau \rangle \,|\, B);\\ \texttt{fupd}(\tau_! \rightsquigarrow a);\\ \texttt{out}\langle \tau_! \circledcirc \tau \rangle\end{array}\right)$$

where $\tau_+ \notin fn(B_1) \cup fn(B_2)$ and $\tau_!, \tau \notin fn(B)$.

Figure 3.5: *SC* encoding of the primitives $+_a$ and $!_a$

We now discuss the behavior of this encoding. Let us consider the following network:

$$a\left[(B_1 +_a B_2) \mid B'\right]_F^R$$

Since the name τ_+ is fresh, it can be extruded outside the component through $\alpha-$renaming. Then, the component is rewritten as follows:

$$(\nu\tau_+)a\left[\left(\begin{array}{l}\texttt{rupd}(\tau_+ \circledcirc \tau_+ \gg B_1 | \tau_+ \circledcirc \tau_+ \gg B_2);\\ \texttt{fupd}(\tau_+ \rightsquigarrow a);\\ \texttt{out}\langle \tau_+ \circledcirc \tau_+ \rangle\end{array}\right) \mid B'\right]_F^R$$

The reaction and flow update the component interface as follow:

$$(\nu\tau_+)a\left[\left(\ \texttt{out}\langle \tau_+ \circledcirc \tau_+ \rangle\ \right) \mid B'\right]_{F|\tau_+ \rightsquigarrow a}^{R|\tau_+ \copyright \tau_+ \gg B_1 | \tau_+ \copyright \tau_+ \gg B_2}$$

The component raises an event of kind τ_+. The spawned envelope depends by the flow of the component for the corresponding topic. Since the topic is restricted, the starting flow F cannot contain the name τ_+, then $(F|\tau_+ \rightsquigarrow a)\downarrow_{\tau_+} = \{a\}$:

$$(\nu\tau_+)a\left[B'\right]_{F|\tau_+ \rightsquigarrow a}^{R|\tau_+ \copyright \tau_+ \gg B_1 | \tau_+ \copyright \tau_+ \gg B_2} \parallel \langle \tau_+ \circledcirc \tau_+ \rangle @ a$$

Both reactions can consume the envelope. One of them will be activated non-deterministically, executing the behavior B_1 or B_2:

$$(\nu\tau_+)a\left[B' \mid B_1\right]_{F|\tau_+ \rightsquigarrow a}^{R|\tau_+ \copyright \tau_+ \gg B_2}$$

The other reaction is not activated and consumed. However, it will not be activated later, since the condition $\tau_+ \notin fn(B_1) \cup fn(B_2)$ ensures that the topic τ_+ will not be used anymore.

Similarly, the primitive $!_a B$ can be used by a component a to implement the unbound replication of the behavior B. The two generated names $\tau_!$ and τ are used to represent the request of execution and a dummy session for the replication. The two names can be extruded outside the component:

$$a\left[!_a B \mid B'\right]_F^R \equiv (\nu\tau_!,\tau)a\left[\left(\begin{array}{l}\texttt{rupd}(\tau_! \;\lambda\, \tau \gg \texttt{out}\langle\tau_!\circledcirc\tau\rangle \mid B);\\ \texttt{fupd}(\tau_! \rightsquigarrow a);\\ \texttt{out}\langle\tau_!\circledcirc\tau\rangle\end{array}\right) \mid B'\right]_F^R$$

The interface of the component is updated according with the flow and reaction update constructs:

$$(\nu\tau_!,\tau)a\left[\texttt{out}\langle\tau_!\circledcirc\tau\rangle \mid B'\right]_{F|\tau_!\rightsquigarrow a}^{R|\tau_!\;\lambda\;\tau\gg\texttt{out}\langle\tau_!\copyright\tau\rangle \mid B}$$

The component notifies itself the private event $\tau_!\circledcirc\tau$, representing the request of execution and the session of the recursion:

$$(\nu\tau_!,\tau)a\left[B'\right]_{F|\tau_!\rightsquigarrow a}^{R|\tau_!\;\lambda\;\tau\gg\texttt{out}\langle\tau_!\copyright\tau\rangle \mid B} \parallel \langle\tau_i\circledcirc\tau\rangle@a$$

The component activates the recursion handler, executing the behavior B and a new notification to continue the unbound replication. Since the recursion handler is a lambda reaction, it is not consumed and is able to react to the further requests.

$$(\nu\tau_!,\tau)a\left[B' \mid \texttt{out}\langle\tau_!\circledcirc\tau\rangle \mid B\right]_{F|\tau_!\rightsquigarrow a}^{R|\tau_!\;\lambda\;\tau\gg\texttt{out}\langle\tau_!\copyright\tau\rangle \mid B}$$

Notice that both $+_a$ and $!_a$ require to know the component name (a). This name is used by the encoding in the flow update primitive. However, extending the language with a meta-component name *self* is immediate.

Joining events

Since *SC* components are autonomous entities communicating through asynchronous primitives, it could be useful to introduce a lightweight synchronization mechanism. We now present an *SC* process that can be used to encode a form of join synchronization among concurrent tasks. It allows us to express that a task can be executed whenever other concurrent tasks have been completed. Figure 3.7 depicts an emitter e, two intermediate components c_1 and c_2, and the join service j. The emitter e starts the communications raising two events of different topics towards c_1 and c_2. Components c_1 and c_2 perform their internal computations and then notify their termination by issuing an event to the join service. The component j waits for both the intermediate services have completed their

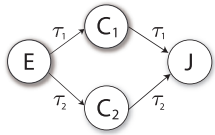


Figure 3.6: Graphical representation

$$
\begin{array}{rcll}
N_e & \triangleq & e\,[(\nu\tau)\mathtt{out}\langle\tau_1 \circledcirc \tau\rangle;\mathtt{out}\langle\tau_2 \circledcirc \tau\rangle]^0_{\tau_1 \rightsquigarrow c_1 | \tau_2 \rightsquigarrow c_2} & \\
N_{c_i} & \triangleq & c_i\,[0]^{\tau_i\ \lambda\ \tau \gg \varepsilon;\mathtt{out}\langle\tau_i \copyright \tau\rangle}_{\tau_i \rightsquigarrow j} & i = 1,2 \\
N_j & \triangleq & j\,[0]^{\tau_1\ \lambda\ \tau \gg \mathtt{rupd}(\tau_2 \copyright \tau \gg B)}_{0} & \\
N_{join} & \triangleq & N_e \parallel N_{c_1} \parallel N_{c_2} \parallel N_j &
\end{array}
$$

Figure 3.7: *SC* model

tasks and then executes its internal behavior B. The signals sent to c_1 and c_2 are both related to the same session τ that is used later by j to apply the synchronization on the same work-flow. Clearly, the two intermediate services c_1 and c_2 can concurrently perform their tasks, while the execution of the service j can be triggered only after the completion of their execution. This example can be modeled by the *SC* network N_{join} depicted in Figure 3.7.

As usual we focus on the management of the control flow. For this reason we do not deal on the behavior currently executed inside the components. Moreover, we assume that the action performed by the two intermediate services c_1 and c_2 have not side effects, allowing us to represent them as internal actions ε.

The join component has only one active reaction installed for signals having topic τ_1. The reception of the τ_1 envelope triggers the activation of the join lambda reaction. This *reads* the session of the signal τ_1 and creates a new specialized reaction for the signal topic τ_2. This reaction can be triggered only by signals that refer to the session received by the τ_1 signal. When such kind of signal is received, the proper behavior B is executed. This synchronization mechanism exploits the asynchronous nature of *SC* event notification. If c_2 is faster than c_1 to notify its termination, the spawned envelope containing the τ_2 event remains pendent over the network. It will be handled as soon as the component j installs

$$
\begin{array}{ll}
N_{par} = & (\nu d)\left(\begin{array}{c} N_i \parallel N_d \parallel \\ Task(a, f \rightsquigarrow \{d\}) \parallel Task(b, f \rightsquigarrow \{d\}) \parallel Task(c, 0) \end{array}\right) \\
where\ N_i = & i\,[0]^{f\ \lambda\ \tau \gg \mathtt{out}\langle f \copyright \tau\rangle \mid \mathtt{out}\langle n \copyright \tau\rangle}_{f \rightsquigarrow \{a,b\} \mid n \rightsquigarrow \{d\}} \\
and\ N_d = & d\,[0]^{n\ \lambda\ \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{out}\langle f \copyright \tau\rangle))}_{f \rightsquigarrow \{c\}} \\
and\ Task(i, F) = & i\,[0]^{f\ \lambda\ \tau \gg \varepsilon; \mathtt{out}\langle f \copyright \tau\rangle}_{F}
\end{array}
$$

Figure 3.8: *SC* models of the BPMN design2.2c

the specialized reaction for the topic τ_2. Notice that the reactive part of the join component is not symmetric. However, the order used by the component j to handle notifications of c_1 and c_2 is not relevant. Modifying the reactions of j to $\tau_2\ \lambda\ \tau \gg \mathtt{rupd}\,(\tau_1 \circledcirc \tau \gg B)$ does not affect the synchronization mechanism.

BPMN Parallel composition

Figure 3.8 presents the *SC* network N_{par}, which models the BPMN business process depicted in Figure 2.2c. The example uses the *SC* session handling mechanism to distinguish concurrent executions of the process. We assume that any execution of the business process is identified by an unique session. Namely, all event raises during the forward-flow will be annotated with the corresponding topic. The notification to a component of an *forward* event (having topic f) represents that all previous stages of the forward-flow have completed their execution.

We implement each BPMN task A with an *SC* component named a defined as $Task(a, F)$, where F are its flows. The Task component has a lambda reaction to handle forward signals. When the reaction is activated, the component binds the session, performs its internal action and then raises a new event of topic f to inform all dependent tasks about its termination. Since the reaction is a lambda, it is not consumed when it is activated by a signal, thus permitting a component to satisfy concurrent demands of the business process. Notice that Task components are implemented independently from the BPMN design, meaning that their reactions and behavior are always the same. These components represent the main building blocks of the system. The only difference among these com-

ponents is represented by their flows. They describe the component view of the coordination.

This example highlights how coordination are implemented using *SC*

1. the logic of a component is implemented via reaction and behavior primitives,

2. the component view of the coordination is obtained by connecting the its flows.

The BPMN tasks A, B and C are implemented via the components a, b and c. To guarantee that the dependencies among tasks satisfy the BPMN design, we exploit the flows of the building blocks and two "special" components named i and d. The former represents the entry point of the business process, while the latter is in charge of synchronizing the executions of the components a and b, before activating the component c. Intuitively, the components i and d have the same roles of the components e and j of the example discussed in Section 3.2.2. However, they employ a slightly different synchronization mechanism. In the join example, the component j synchronizes two notifications having different topics (τ_1 and τ_2). Instead, the component d has to synchronize notifications having always the same topic (f). For this reason, the component d cannot comply its role by installing a check reaction for signal having topic f after the reception, by a lambda reaction, of a first signal with the same topic. In fact, it is not ensured that the further notification will trigger the check reaction, since the lambda reaction is persistent. Both f signals must be consumed by check reactions, that are installed after a notification of the work-flow session by the component i.

Notice that the name of the component d is restricted. This represents that it is not visible from components living outside the work-flow implementation. In fact, the role of the component is to synchronize the internal notifications raised by the components a and b.

In order to have a snapshot of the execution, we send a request to the entry point i of the business process. We spawn into the network an envelope $\langle f_{\circledcirc} s \rangle @ i$, where s represent the session of this execution:

$$\langle f_{\circledcirc} s \rangle @ i \parallel N_{par} \downarrow$$

The entry point consumes the envelope, binds the received session and activates the corresponding behavior. The reaction is not consumed, thus permitting to the

entry point to serve other concurrent requests:

$$\downarrow$$
$$(\nu d)\left(\begin{array}{c} i\left[\mathtt{out}\langle f \circledcirc s\rangle \mid \mathtt{out}\langle n \circledcirc s\rangle\right]^{f\ \lambda\ \tau \gg \mathtt{out}\langle f \copyright \tau\rangle \mid \mathtt{out}\langle f \copyright \tau\rangle}_{f \rightsquigarrow \{a,b\} \mid n \rightsquigarrow \{d\}} \parallel \\ N_d \parallel \mathit{Task}(a, f \rightsquigarrow \{d\}) \parallel \mathit{Task}(b, f \rightsquigarrow \{d\}) \parallel \mathit{Task}(c, 0) \end{array}\right)$$
$$\downarrow^*$$

The entry point notifies the events f and n. The former forwards the request to the implementations of the tasks A and B, while the latter delivers the session to the join component. Since i has two components connected by the flow f, the envelopes spawned by the emission of event f are two:

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} N_i \parallel N_d \parallel \mathit{Task}(a, f \rightsquigarrow \{d\}) \parallel \mathit{Task}(b, f \rightsquigarrow \{d\}) \parallel \mathit{Task}(c, 0) \\ \langle f \circledcirc s\rangle @ a \parallel \langle f \circledcirc s\rangle @ b \parallel \langle n \circledcirc s\rangle @ d \end{array}\right)$$
$$\downarrow^*$$

The components a, b and d can consume concurrently their envelopes, activating the behaviors of their lambda reactions:

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} N_i \parallel \\ d\left[\mathtt{rupd}\left(f \circledcirc s \gg \mathtt{rupd}\left(f \circledcirc s \gg \mathtt{out}\langle f \circledcirc s\rangle\right)\right)\right]^{n\ \lambda\ \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{out}\langle f \copyright \tau\rangle))}_{f \rightsquigarrow \{c\}} \parallel \\ a\left[\varepsilon; \mathtt{out}\langle f \circledcirc s\rangle\right]^{f\ \lambda\ \tau \gg \varepsilon; \mathtt{out}\langle f \copyright \tau\rangle}_{f \rightsquigarrow \{d\}} \parallel \\ b\left[\varepsilon; \mathtt{out}\langle f \circledcirc s\rangle\right]^{f\ \lambda\ \tau \gg \varepsilon; \mathtt{out}\langle f \copyright \tau\rangle}_{f \rightsquigarrow \{d\}} \parallel \\ \mathit{Task}(c, 0) \end{array}\right)$$
$$\downarrow^*$$

The component d installs a check reaction to implement the synchronization of tasks A and B. This reaction can be activated only by events raised exactly for this instance of the business process, in other words having session s. Concurrently, both component a and b can perform their internal behavior, which represents the execution of the corresponding BPMN task.

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} I \parallel \\ d\left[0\right]^{n\ \lambda\ \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{rupd}(f \copyright \tau \gg \mathtt{out}\langle f \copyright \tau\rangle)) \mid f \copyright s \gg \mathtt{rupd}(f \copyright s \gg \mathtt{out}\langle f \copyright s\rangle)}_{f \rightsquigarrow \{c\}} \parallel \\ a\left[\mathtt{out}\langle f \circledcirc s\rangle\right]^{f\ \lambda\ \tau \gg \varepsilon; \mathtt{out}\langle f \copyright \tau\rangle}_{f \rightsquigarrow \{d\}} \parallel \\ b\left[\mathtt{out}\langle f \circledcirc s\rangle\right]^{f\ \lambda\ \tau \gg \varepsilon; \mathtt{out}\langle f \copyright \tau\rangle}_{f \rightsquigarrow \{d\}} \parallel \\ \mathit{Task}(c, 0) \end{array}\right)$$
$$\downarrow^*$$

The components a and b inform the dispatcher about their termination, rising an event with topic f. The two spawned envelopes are identical, so the component d cannot distinguish who raised the signal it consumes:

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} N_i \parallel \\ d\,[0]^{n\ \lambda\ \tau \gg \texttt{rupd}(f \copyright \tau \gg \texttt{rupd}(f \copyright \tau \gg \texttt{out}\langle f \copyright \tau\rangle))\mid f \copyright s \gg \texttt{rupd}(f \copyright s \gg \texttt{out}\langle f \copyright s\rangle)}_{f \rightsquigarrow \{c\}} \parallel \\ \parallel Task(a, f \rightsquigarrow \{d\}) \parallel Task(b, f \rightsquigarrow \{d\}) \parallel Task(c, 0) \parallel \\ \langle f \copyright s\rangle @ d \parallel \langle f \copyright s\rangle @ d \end{array}\right)$$
$$\downarrow^*$$

The component d activate the check reaction consuming non-deterministically one of the two envelopes. The behavior installs a new check reaction for the same kind of events, to implement the synchronization of the two branches of the forward-flow.

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} N_i \parallel \\ d\,[0]^{n\ \lambda\ \tau \gg \texttt{rupd}(f \copyright \tau \gg \texttt{rupd}(f \copyright \tau \gg \texttt{out}\langle f \copyright \tau\rangle))\mid f \copyright s \gg \texttt{out}\langle f \copyright s\rangle}_{f \rightsquigarrow \{c\}} \parallel \\ \parallel Task(a, f \rightsquigarrow \{d\}) \parallel Task(b, f \rightsquigarrow \{d\}) \parallel Task(c, 0) \parallel \\ \langle f \copyright s\rangle @ d \end{array}\right)$$
$$\downarrow^*$$

Notice that, at run-time, the component d can have several check reactions installed. Each of them represents a pending synchronization of the forward-flow for different instances of the business process. Now, d can terminate its synchronization, consuming the pending envelope and notifying c that all previous stages have terminated their execution:

$$\downarrow$$
$$(\nu d)\left(\begin{array}{l} N_i \parallel N_d \parallel Task(a, f \rightsquigarrow \{d\}) \parallel Task(b, f \rightsquigarrow \{d\}) \parallel Task(c, 0) \parallel \\ \langle f \copyright s\rangle @ c \end{array}\right)$$

Finally, the component c consumes its envelope and performs the BPMN corresponding task C.

3.3 Dynamic types: *xSC*

The *SC* dialects illustrated in the previous sections can be classified as a topic-based event-notification [62]. In topic-based systems, events are categorized into

topics which subscribers register to. When an event belonging to a topic τ is emitted, all the components subscribed for τ will eventually react to the event. Notice that publishers and subscribers have to know the topics at hand.

In the event notification paradigm, component decoupling can be enforced by allowing subscribers to register for events satisfying a given *properties*. When an event is emitted, it is dispatched to *all* the subscribers whose property holds on that event. In literature, this approach is called *type-based* event-notification [62] where topics are replaced by types (in a suitable type language). Usually, typed events are used to let programmers to specify properties through the public interface of events described by their type.

This section recasts *SC* into a type-based framework, called the eXtended Signal Calculus (*xSC*). The *xSC* is a "typed version" of *SC* where events are emitted with types that coordinate publishers/subscribers interactions. For instance, an *xSC* publisher can emit an event with type $\tau \times \tau'$ that should be received by subscribers that can react to events of type τ *and* τ'. *xSC* types have a twofold role. First, typing allows subscribers to filter their events of interest (as usual in type-based event-notification). Second, publishers exploit type information to specify which (kind of) subscribers should react to events. For instance, in the previous example, a subscriber that is able to react only to events of type τ will not be capable of reacting to an event $\tau \times \tau'$. The way types are used is indeed the main original contribution of *xSC* with respect to standard type-based EN systems.

In this section we develop this idea by introducing some operators on topics that induce an algebraic structure on events. We then show how the algebraic structure on events can be used to have a finer control over the coordination activities of components.

The syntax of signal topics t is defined in Table 3.8a. The constant topics $\bullet$ and $\star$ are used to define the *empty* and the *global* event kinds, respectively. Intuitively, a signal having an empty topic can be consumed by a reaction having an empty behavior. A signal having a global topic can be handled by any component, activating any reaction. Signal topics can be composed using the constructors $\times$ and $+$. A signal having topic $t \times t'$ can be consumed only by components that can handle both event kinds t and t'. Moreover a signal having topic $t + t'$ can be consumed by any component that can handle event kinds t or t'. The constructors $+$ and $\times$ can be informally interpreted as logical *disjunction* and *conjunction*.

The formal definition of the meaning of structured topics is given algebraically by introducing a structural congruence over them (see Table 3.8b). Notice that the $\times$ and $+$ are associative, commutative and idempotent. Also, $\times$ distributes over $+$, moreover, $\star$ and $\bullet$ are their respective neutral elements. For instance, $t \times \star \equiv t$ and $t + \bullet \equiv t$ states that a signal of topic $t \times \star$ or $t + \bullet$ activates the

$$t ::= \bullet \mid \star \mid \tau \mid t \times t \mid t + t$$

(a) *SC* syntax of topics

$$t' \times t'' \equiv t'' \times t' \qquad t' + t'' \equiv t'' + t'$$
$$t \times t \equiv t \qquad t + t \equiv t$$
$$t' \times (t'' \times t''') \equiv (t' \times t'') \times t''' \qquad t' + (t'' + t''') \equiv (t' + t'') + t'''$$
$$t \times \star \equiv t \qquad t + \bullet \equiv t$$
$$t \times \bullet \equiv \bullet \qquad t + \star \equiv \star$$
$$t \times (t' + t'') \equiv (t \times t') + (t \times t'')$$

(b) *SC* structural congruence over topics

$$t \sqsubseteq \bullet, \qquad \star \sqsubseteq t, \qquad t \sqsubseteq t, \qquad t \sqsubseteq t \times t', \qquad t + t' \sqsubseteq t$$

(c) *SC* subtype relation over topics

Table 3.8: *SC* topics

same reactions activated by signals having topic t; similarly $t \times \bullet \equiv \bullet$ states that a signal of topic $t \times \bullet$ cannot activate any reaction, while $t + \star \equiv \star$ states that a signal of topic $t + \star$ activates any reaction. Formally, the algebraic structure over topic takes the form of a C-Semiring [63]. This interpretation induces a natural preorder relation over topics.

Definition 3 *The binary relation $\sqsubseteq$ over topics is the least preorder satisfying the axioms in Table 3.8c*

Intuitively the preorder $t_1 \sqsubseteq t_2$ formalizes the idea that the topic t_1 is less restrictive than the topic t_2. For example, a signal having topic $\tau_1 + \tau_2$ triggers either a reaction for τ_1 or one for τ_2. Hence, the coordination policy expressed by $\tau_1 + \tau_2$ is less restrictive than the one expressed by τ_1.

The algebraic structure over topics allows us to define policies to aggregate events. The *xSC* syntax of behaviors can be extended to deal with the structure of topics by simply refining the signal emission primitive as $\texttt{out}\langle t \triangleleft \tau \rangle$, where t represents the signal topic. We have now to specify the way a component may react upon the reception of a signal of a certain topic. In other words, a main question, here, is to understand which reactions a component may dynamically activate to match the policy specified by the topics of events. We will answer this question by introducing a suitable type system over component reactions. The type system allows us to precisely identify the set of reactions matching a given event topic.

$$T ::= \quad t_{\circledcirc}\tau \mid \quad \textit{(Session conversation type)}$$
$$\qquad\quad t_{\circledcirc}\star \quad \textit{(Generic conversation type)}$$

(a) *SC* syntax of conversation types

$$\frac{t \equiv t'}{t_{\circledcirc}\tau \equiv t'_{\circledcirc}\tau} \qquad \frac{t \equiv t'}{t_{\circledcirc}\star \equiv t'_{\circledcirc}\star}$$

(b) *SC* structural congruence over conversation types

$$\frac{t \sqsubseteq t'}{t_{\circledcirc}\tau \sqsubseteq t'_{\circledcirc}\tau}\ (1) \qquad \frac{t \sqsubseteq t'}{t_{\circledcirc}\tau \sqsubseteq t'_{\circledcirc}\star}\ (2) \qquad \frac{t \sqsubseteq t'}{t_{\circledcirc}\star \sqsubseteq t'_{\circledcirc}\star}\ (3)$$

(c) *SC* subtype relation over conversation types

Table 3.9: *SC* conversation types

First, we introduce the notion of *Conversation Types* (ranged over by T). Conversation Types classify signals by their topic structures (policies) and sessions. The syntax is defined in Table 3.9a. A *session conversation type* $t_{\circledcirc}\tau$ characterizes signals (of a topic t) *within* a session τ. A *generic conversation type* $t_{\circledcirc}\star$ captures the notion of signals (of a topic t) not belonging to a specific session. Two conversation types are equivalent if the structures of their topics and their sessions are equivalent. Formally, equations in Table 3.8b are extended with rules in Table 3.9b.

Conversation types can be equipped with a subtype relation which will be used to formalize how signals are consumed by reactions. The intuition is that if $T \sqsubseteq T'$ then reactions capable to consume signals with conversation type T' can consume signals with conversation type T as well.

Definition 4 *The subtype relation $T \sqsubseteq T'$ over conversation types is defined as the smallest preorder relation that satisfies the inference rules in Table 3.9c*

Rules (1) and (3) have a clear interpretation in terms of the preorder over topics. Rule (2) is controvariant with respect to the session part of the conversation type and formalizes the idea that a lambda reaction can be activated by signals independently by their session.

A *reaction type* is a (possibly empty) set of conversation types and describes the set of signals that can be consumed by a reaction. We use $\vdash R : \mathbb{T}$ to say that the reaction R has reaction type $\mathbb{T}$. The type of a reaction is inferred from the

$$\mathbb{T} \in \mathbb{P}(T)$$

(a) *SC* syntax of reaction types

$$\frac{\mathbb{T}_1 \subseteq \mathbb{T}}{\mathbb{T}_1 \sqsubseteq \mathbb{T}}$$

(b) *SC* subtype relation over reaction types

$$\frac{}{\vdash 0 : \emptyset}\ (1) \qquad \frac{}{\vdash \tau \circledcirc \tau' \gg B : \{\tau \circledcirc \tau'\}}\ (2)$$

$$\frac{}{\vdash \tau\ \lambda\ \tau' \gg B : \{\tau \circledcirc \star\}}\ (3) \qquad \frac{\vdash R_1 : \mathbb{T}_1 \qquad \vdash R_2 : \mathbb{T}_2}{\vdash R_1 | R_2 : \mathbb{T}_1 \cup \mathbb{T}_2}\ (4)$$

(c) *SC* typing tules for reactions

$$\frac{\mathbb{T} = \{\tau_1 \circledcirc r_1, \ldots, \tau_n \circledcirc r_n : r_i \in \Lambda \cup \{\star\} \text{ for } i = 1, \ldots, n\}}{\begin{array}{ll} {}^{\times}\mathbb{T} = \tau_1 \times \ldots \times \tau_n & \mathbb{T}^{\times} = r_1 \times \ldots \times r_n \\ {}^{+}\mathbb{T} = \tau_1 + \ldots + \tau_n & \mathbb{T}^{+} = r_1 + \ldots + r_n \end{array}}$$

$$\frac{\mathbb{T} = \emptyset}{\begin{array}{l} {}^{\times}\mathbb{T} = \star = \mathbb{T}^{\times} \\ {}^{+}\mathbb{T} = \bullet = \mathbb{T}^{+} \end{array}}$$

(d) Operators over reaction types

Table 3.10: *SC* reaction types

rules in Table 3.10c Rules $(1 \div 4)$ are quite natural. For instance, rule (3) states that the type of a lambda reaction $\tau\ \lambda\ \tau' \gg B$ is the singleton $\{\tau \circledcirc \star\}$. Reaction types have a natural subtype relation given by the subset inclusion. We use $\mathbb{T} \sqsubseteq \mathbb{T}'$ to denote that the subtype yields.

In Table 3.10d we introduce some auxiliary operators over reaction types. These operators simply merge the topic and session parts of a reaction type. The following properties trivially hold.

$$\begin{array}{ll} {}^{\times}\mathbb{T} = \star \Leftrightarrow \mathbb{T} = \emptyset & \mathbb{T}^{\times} = \tau \Rightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \forall r_i . r_i \in \{\tau, \star\}) \\ \mathbb{T}^{+} = \bullet \Leftrightarrow \mathbb{T} = \emptyset & \mathbb{T}^{\times} = \star \Leftrightarrow (\mathbb{T} = \emptyset \ \vee\ \forall r_i . r_i = \star) \\ \mathbb{T}^{+} = \star \Leftrightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \exists r_i . r_i = \star) & \mathbb{T}^{+} = \tau \Leftrightarrow (\mathbb{T} \neq \emptyset \ \wedge\ \forall r_i . r_i = \tau) \end{array}$$

After having defined the preorder on topics and the subtype relation for conversation types, we define a formal mechanism that establishes when a reaction is *enabled* to handle a signal reception. This definition is the basic tool that will

Conversation Type $t_{\circledcirc}\tau$	Reaction Type $\mathbb{T}$	$t \sqsubseteq^{\times} \mathbb{T}$	$\mathbb{T}^{\times} \sqsubseteq \tau$	Cond. 2
$\tau_1 + \tau_{2\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau\}$	$\surd$	$\surd$	$\surd$
$\tau_1 \times \tau_{2\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau, \tau_{2\circledcirc}\star\}$	$\surd$	$\surd$	$\surd$
$\tau_1 \times \tau_{2\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau\}$	$\times$	$\surd$	$\surd$
$\tau_{1\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau'\}$	$\surd$	$\times$	$\surd$
$\tau_1 + \tau_{2\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau, \tau_{2\circledcirc}\star\}$	$\surd$	$\surd$	$\times$
$\tau_1 \times \tau_{2\circledcirc}\tau$	$\{\tau_{1\circledcirc}\tau, \tau_{2\circledcirc}\star, \tau_{3\circledcirc}\star\}$	$\surd$	$\surd$	$\times$

Figure 3.9: Reaction enabling examples

be exploited at run-time to activate the reaction matching an event notification.

Definition 5 *Let $T \equiv t_{\circledcirc}\tau$ be a conversation type and $\mathbb{T}$ a non empty reaction type. We say that* reactions with type $\mathbb{T}$ can be activated by signals with conversation type T*, and we write $T \Subset \mathbb{T}$, if the following conditions hold:*

1. $t \sqsubseteq^{\times} \mathbb{T}$ *and* $\mathbb{T}^{\times} \sqsubseteq \tau$
2. $\nexists \mathbb{T}' \subset \mathbb{T} \mid \mathbb{T}' \not\equiv \emptyset \wedge (t \sqsubseteq^{\times} \mathbb{T}'$ *and* $\mathbb{T}'^{\times} \sqsubseteq \tau)$

Figure 3.9 gives examples where conditions 1 and 2 hold or not.

Condition 1 expresses that the topic of the signals is less restrictive than the conjunction of the topics of the reactions ($t \sqsubseteq^{\times} \mathbb{T}$). Notice that, since $\mathbb{T}$ is not empty then it is of the form

$$\{\tau_{1\circledcirc}r_1, \ldots, \tau_{n\circledcirc}r_n : r_i \in \Lambda \cup \{\star\}\, for\, i = 1, \ldots, n\}$$

Also, reactions waiting for a session topic different from τ cannot be activated. In fact, to ensure $\mathbb{T}^{\times} \sqsubseteq \tau$, must hold that $\forall i.r_i \equiv \tau \vee r_i \equiv \star$. Intuitively, any subreaction must be a lambda reaction ($r_i \equiv \star$) or check reaction waiting for the signal session ($r_i \equiv \tau$).

Condition 2 ensures that enabled reactions are *minimal*, namely, that each subreaction ($\forall \mathbb{T}' \subset \mathbb{T}$) cannot be activated by signals having signal type T.

Definition 6 *Given a reaction R, the* set of enabled subreactions by a conversation type $t_{\circledcirc}\tau$*, written $R_{t\circledcirc\tau}$, is defined as:*

$$R_{t\circledcirc\tau} = \{R'.R \equiv R' | R'' \wedge \vdash R' : \mathbb{T} \wedge t_{\circledcirc}\tau \Subset \mathbb{T}\}$$

Conversation Type $t_{\circledcirc}\tau$	Reaction R	$R_{t\circledcirc\tau}$
$\tau_1 + \tau_{2\circledcirc}\tau$	R_1	$\{R_1\}$
$\tau_1 \times \tau_{2\circledcirc}\tau$	R_1	$\emptyset$
$\tau_1 + \tau_{2\circledcirc}\tau$	$R_1 \vert R_2$	$\{R_1, R_2\}$
$\tau_1 \times \tau_{2\circledcirc}\tau$	$R_1 \vert R_2$	$\{R_1 \vert R_2\}$

Figure 3.10: Enabled reaction set example

We illustrate the notion of the *set of enabled subreactions by a conversation type* by a simple example in Table 3.10, where R_1 is $\tau_{1\circledcirc}\tau \gg B_1$ and R_2 is $\tau_2\ \lambda\ \tau' \gg B_2$. Notice that in the last row of the table only one reaction $(R_1|R_2)$ is enabled. Upon the reception of a signal having conversation type $\tau_1 \times \tau_{2\circledcirc}\tau$, both subreactions R_1 and R_2 will be concurrently activated. Also, in the third row of the table two different reactions (R_1 and R_2) are enabled. Upon the reception of a signal having conversation type $\tau_1 + \tau_{2\circledcirc}\tau$, only one of them will be activated non-deterministically.

Definition 7 *Let R be a reaction and $t_{\circledcirc}\tau$ be a session conversation type. The set of* preferred reactions in R wrt $t_{\circledcirc}\tau$ *is defined as:*

$$R_{t\circledcirc\tau\downarrow} = \left\{ R_1 \in R_{t\circledcirc\tau}.\ \vdash R_1 : \mathbb{T}_1 \Rightarrow \forall R_2 \in R_{t\circledcirc\tau}.\vdash R_2 : \mathbb{T}_2 \Rightarrow \left(\begin{array}{c} \mathbb{T}_1^+ \equiv \tau \\ \vee \\ \mathbb{T}_2^\times \sqsubseteq \mathbb{T}_1^\times \end{array} \right) \right\}$$

Basically, each reaction $R_1 \in R_{t\circledcirc\tau\downarrow}$ satisfies the one following properties:

- it consists only of check sub-reactions for the τ session ($\mathbb{T}_1^+ \equiv \tau$)
- it consists of at least a lambda sub-reaction, but any other candidate ($\forall R_2 \in R_{t\circledcirc\tau}.\vdash R_2 : \mathbb{T}_2$) is less specialized ($\mathbb{T}_2^\times \sqsubseteq \mathbb{T}_1^\times$)

The topic structures can be adopted to model the join example described in Section 3.2.2, refining the emitter component as

$$N'_e \triangleq e\,[(\nu\tau)\mathtt{out}\langle\tau_1 + \tau_{2\circledcirc}\tau\rangle]^0_{\tau_1 \rightsquigarrow c_1 | \tau_2 \rightsquigarrow c_2}$$

The operational semantics of *xSC* is given in the classical reduction style and exploits the structural congruences defined in Section 3.2. Some auxiliary functions on flows and reactions are introduced for simplifying the definition of the reduction relation on networks.

$\tau \rightsquigarrow \vec{a} \downarrow_{\tau} = \vec{a}$	$\tau \rightsquigarrow \vec{a} \downarrow_{\tau'} = \tau \rightsquigarrow a \downarrow_{\bullet} = 0 \downarrow_{t} = \emptyset$
$\tau \rightsquigarrow \vec{a} \downarrow_{\star} = \vec{a}$	$F_1 \mid F_2 \downarrow_{t} = F_1 \downarrow_{t} \cup F_2 \downarrow_{t}$
$F \downarrow_{t_1 + t_2} = F \downarrow_{t_1} \cup F \downarrow_{t_2}$	$F \downarrow_{t_1 \times t_2} = F \downarrow_{t_1} \cap F \downarrow_{t_2}$

(a) Flow projection function

$(0)\downarrow_{\star} = (0,0)$
$(\tau' \circledcirc \tau'' \gg B)\downarrow_{t \circledcirc \tau} = (B,0)$
$(\tau' \;\lambda\; \tau'' \gg B)\downarrow_{t \circledcirc \tau} = (\{\tau/\tau''\}B, \tau' \;\lambda\; \tau'' \gg B)$
$(R_1 \mid R_2)\downarrow_{t \circledcirc \tau} = (B' \mid B'', R' \mid R''),$ if $(R_1)\downarrow_{t \circledcirc \tau} = (B', R')$ and $(R_2)\downarrow_{t \circledcirc \tau} = (B'', R'')$

(b) Reaction projection function

Table 3.11: x*SC* projection functions

In Table 3.11 we extend the *flow projection* function and we introduce the *reaction projection* function. The former, $F \downarrow_t$, takes a flow and a topic and yields the set of target component names for the topic t. The latter, $(R)\downarrow_{s:T}$, takes a reaction R and a signal s typed by T and returns a pair (B, R') such that B is the behavior of R instantiated with s and R' is the reaction to be installed. Notice that reaction projection permits to consume check reactions and to maintain lambda reactions installed. Also, reaction projection is applied, by construction, to reactions that can consume the signal s. This assumption is guaranteed by the reduction rules using the type system.

The reduction relation $\rightarrow$ over networks is defined in Table 3.12. We comment on the two reduction rules specific for this dialect of *SC*. The rule (EMIT) defines dispatching of notifications. At emission time, component a spawns into the network a signal targeted to all the components ($c_i \in \vec{b}$) subscribed for the signal type (according to the $F \downarrow_t$ projection). On the other hand, rule (REACT) substitute the *SC* reduction rules (LAMBDA) and (CHECK). Once an envelop has been spawn into the network the rule (REACT) can be applied to the target component; the application of this rule activates, non deterministically, a reaction among the ones in the reaction projection $R' \in R_{t \circledcirc \tau\downarrow}$. Then, the activated reaction is replaced in the interface of a by R'' reaction obtained by applying the reaction projection.

$$\frac{}{a\,[\varepsilon;B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R}\ (SKIP)$$

$$\frac{}{a\,[\mathtt{rupd}\,(R_1);B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_F^{R|R_1}}\ (RUPD)$$

$$\frac{}{a\,[\mathtt{fupd}\,(F_1);B_1 \mid B_2]_F^R \rightarrow a\,[B_1 \mid B_2]_{F|F_1}^R}\ (FUPD)$$

$$\frac{t \downarrow_\tau = \vec{b}}{a\left[\mathtt{out}\langle t \circledcirc \tau' \rangle;B_1 \mid B_2\right]_F^R \rightarrow a\,[B_1 \mid B_2]_F^R \parallel \prod_{b_i \in \vec{b}} \langle t \circledcirc \tau' \rangle @ b_i}\ (OUT)$$

$$\frac{N \rightarrow N_1}{N \parallel N_2 \rightarrow N_1 \parallel N_2}\ (PAR) \qquad \frac{N \equiv N_1 \rightarrow N_2 \equiv N_3}{N \rightarrow N_3}\ (STRUCT)$$

$$\frac{R \equiv R'|R_0 \quad R' \in R_{t \circledcirc \tau\downarrow} \quad (R')\downarrow_{t \circledcirc \tau} = (B_2, R'')}{\langle t \circledcirc \tau \rangle @ a \parallel a\,[B_1]_F^R \rightarrow a\,[B_1 \mid B_2]_F^{R_0|R''}}\ (REACT)$$

$$\frac{N \rightarrow N'}{(\nu n)N \rightarrow (\nu n)N'}\ (NEW)$$

Table 3.12: Operational semantics

3.4 Related works

In this Chapter we have introduced three dialects of the *SC* calculus. These dialects have been designed around the adoption of the event notification paradigm to coordinate distributed components.

The *SC* dialect presented in Section 3.1 illustrates the main features of the *SC* programming model. The distinguished feature of *SC* is multicast notification. Even though multicast notification can be encoded using the existing process calculi, we argue that its integration into the programming model simplifies the design of complex systems (see Section 3.1.2). The adoption of a notification style different from the unicast message passing was already investigated by SCCS [64] and CBS [65]. However, these calculi feature broadcast communi-

cation patterns that are not suitable for distributed and loosely coupled systems. Broadcast requires the definition of broadcasting domains to confine its scope on wide networks. Furthermore, multicast communication mechanism of *SC* is asynchronous and that the envelopes generated by an event rising are delivered independently.

The notion of *SC* component is strictly related to ambients of the ambient calculus [66] and agent of the Nomadic PICT [67] to cite a few. However, we do not allow locations to be nested and to migrate. Nested locations are usually used to model hierarchical networks (i.e. VPN) and to restrict the access privileges to services. These notions are out of the scope of our programming model and should be delegated to a different level of abstraction. We also do not allow component migration because we want provide a coordination framework free from platform constraints. In fact, component migration requires to establish a reference platform that all parties must comply.

The idea of exploiting a peer-to-peer like structure (*SC* flows) to coordinate agents has been already investigated by Reo [68]. Reo is a coordination model based on connections among components that allows dynamic reconfiguration. Our work mainly differs from Reo on the communication model adopted for composition. Reo is a channel based framework, while *SC* is an event based one. Hence, Reo handles component migration and channel management as basic notions, while *SC* focuses on the activities performed and on the coordination over dynamic network topologies.

The *SC* calculus can be classified as a non-brokered approach to the event notification paradigm. In fact, each component is responsible to manage its subscribers and to deliver the raised events to them. A different approach is the so called brokered event notification. The Linda coordination language [59] has been one of the first formalism able to model this kind of systems. Basically, Linda permits to model process coordination by the production and consumption of tuples from a global memory, called tuple space. The tuple space can be used to represent the subscriptions among components by a global point of view. However, the implementation of this solution usually requires a centralization point that can represent a key issue in SOA [69]. Klaim [70] is a distributed version of Linda that has been developed to model and reason about code mobility. Klaim locations resembles *SC* components.

The dialect of *SC* presented in Section 3.2 extends our programming model with primitives to manage sessions. Session handling is a key issue of SOA. In fact, services involved into a business process must be able to correlate messages (*SC* events) that belong to the same instance of the process. The *SC* session handling is quite simple: we annotate events with a session identifier and we extend reactions with a simple form of pattern matching. They can filter mes-

sages by their sessions (i.e. check reactions) and can acquire the session of a received event (i.e. lambda reactions). This session handling mechanism take inspiration from the *Correlation Sets*, which have been used by WS-BPEL [9] and COWS [71]) to represent properties of message structures and exploit these properties to correlate distinguished messages.

Other approaches (e.g. SCC [14] and Muse [18] and the π-calculus extension given in [72]) exploit the notion of session to identify the scope of interactions. Sessions are first order entities and calculi can provide primitives to create, communicate and fuse sessions.

Chapter 4

Reasoning with *SC*

SOA applications require methodologies to clearly define the coordination of involved services. Unambiguous specifications of the service interactions permit to verify correctness of systems and avoid unattended behavior. In this Chapter we introduce our reasoning techniques for *SC*.

In Section 4.1 we introduce the syntax and the observational of a process calculus, called Network Coordination Policies (*NCP*), which extends and equips our framework with a choreography model. The *SC* and *NCP* lay at two different levels of abstraction. The former is tailored to support the (formal) design of services, the latter is the specification language to declare the coordination policies. Policies take the form of processes that represent the behavior as observed from a *global* point of view, namely by observing all the public interactions taking place on the network infrastructure. Hence, an *NCP* process describes the interactions that are expected to happen and how these are interleaved. Indeed, certain features can be described at both levels: the *NCP* specification declares *what* is expected from the service network infrastructure, the *SC* design specifies *how* to implement it.

NCP and *SC* share the same computational paradigm and the two semantics are related by a correctness result: for each *SC* network, there is an *NCP* policy that reflects all the properties of the network. In Section 4.2 we establish this result by the introduction of a semantics-based transformation, mapping a *SC* design into *NCP* network. We show that the transformation is fully abstract with respect to an abstract semantics notion. The converse is not true: not every *NCP* coordination policy that one can specify is implementable in *SC*. We also formalize when an *SC* design respect an *NCP* policy. These results allow us to exploit the choreography model to check consistency of *SC* designs and suggest

a model driven development approach. The designer can define successive *SC* models that implement a system, each of them is obtained refining the previous one to add more details. Moreover, the consistency of each model with respect to a *NCP* specification can be formally verified.

In Section 4.3, we use *SC* to implement LRT designed in saga. The encoding enables designers to specify LRTs using a flow language and to mechanically obtain a reference *SC* design that respects the transactional requirements. Then, the designer can enrich the *SC* model to care about relevant aspects that cannot be described using saga, which is a more abstract than *SC*. The correctness of the refined *SC* models can be checked respect to the reference implementation, by using the results presented in Sections 4.1 and 4.2.

Our theoretical results inspired a more practical methodology to refine the *SC* models, without break their correctness. Section 4.4 illustrates some refactoring rules that address some crucial issues of the deployment phase, that is the possible alterations that one would like (or has) to apply at the *SC* level where they can be more suitably tackled. Arguably, refactoring does not have to alter the intention of the designer, namely, refactoring rules must preserve the intended semantics. Our refactoring rules are proved sound by showing that they preserve (weak) bisimulation. The proof relies on a bisimulation preserving mapping from *SC* to its choreographic view expressed in *NCP*.

4.1 Network Coordination Policies

The Network Coordination Policies (*NCP*) calculus has been specifically designed to provide the choreography model for *SC*. We developed *NCP* to specify systems that can be implemented by *SC* components and to provide a verification methodology to reason about *SC* designs. For these reasons, the primitives supplied by *NCP* are inspired by the *SC* programming model; the language specifies the computation of components, which communicate via the event notification paradigm. We start introducing the syntax of *NCP*.

We assume that components are uniquely identified by names $a, b, \ldots \in A$, which can be infinite. *NCP* classifies events analogously to the *SC* dialect presented in Section 3.2; namely, an event is couples of names, which represent its topic and its session identifier. We assume that topic names are $\tau_1, \tau_2, \ldots \in \mathcal{T}$.

The *NCP* calculus is equipped with the same multicast notification mechanism of *SC*. However, the different goals of *NCP* and *SC* have induced two separate approaches to model the subscription relation among components. *SC* exploits the flows of components, while *NCP* model this information by a global point of view, introducing the notion of *network topologies*. Informally, a net-

work topology represents the flows of all components involved by the coordination.

An *NCP* specification is composed by two entities: a policy and a network topology. The former describes the actions that should be performed by components, the latter describes the component inter-connections.

A *network topology* is a structure $G = (V, E)$, where $V \subseteq A$ consists of the restricted component names of the network and $E \subseteq A \times \mathcal{T} \times A$ are the flow connections among components; $(a, \tau, b) \in E$, representing that component a has a flow towards b for signals of topic τ. Notice that, G induces a directed labeled graph whose vertexes are the names of the network components (the restricted ones of which are highlighted in V) and whose edges are the elements in E. Abusing of notation, hereafter we will confuse G with its associated graph. We also introduce the notion of *topicgraph*, ranged by T, which is an unlabeled directed graph having component names as nodes. Figure 4.1 depicts some network topologies that will be used in the following as illustrative examples.

It is useful to define the auxiliary notations in Table 4.1, where $|\mathcal{G}|$ denotes the set of vertexes of graph $\mathcal{G}$, $G = (V, E)$ is a network topology, and $a \in A$ and $\tau \in \mathcal{T}$. We report some examples of the *NCP* auxiliary notations, exploiting the network topologies depicted in Figure 4.1.

$$G_1(a) = G_2(a) = \{(\tau, b)\} \qquad G_5(a) = \emptyset$$
$$G_3(a) = \{(\tau, b), (\tau, c)\} \qquad G_4(a) = \{(\tau, b), (\tau, c), (\tau', b)\}$$
$$G_1 \cap a = G_2 \cap a = G_1$$
$$G_4(\tau') = G_1(\tau) = G_6(\tau') = G_7(\tau') = \{(a, b)\}$$
$$G_4(\tau) = G_3(\tau) = \{(a, b), (a, c), (b, c)\}$$
$$G_6(a, \tau) = G_7(a, \tau) = G_3(a, \tau) = \{b, c\}$$
$$a \boxtimes (\tau, b) = G_1 \qquad G_4 \uplus G_5 = G_7$$

The following statements over *NCP* network topologies holds trivially:

$$G \uplus G = G$$

$$G \uplus \emptyset = G \qquad G \uplus G' = G' \uplus G$$

$$\frac{T = G(\tau)}{G \setminus \tau \boxdot T \uplus \tau \boxdot T = G} \qquad \frac{T = G(\tau)}{\tau \notin fn(G \setminus \tau \boxdot T)}$$

$$\frac{T = T_1 \cup T_2}{T(\tau) = T_1(\tau) \cup T_2(\tau)} \qquad \frac{G = G_1 \uplus G_2}{G(a) = G_1(a) \cup G_2(a)}$$

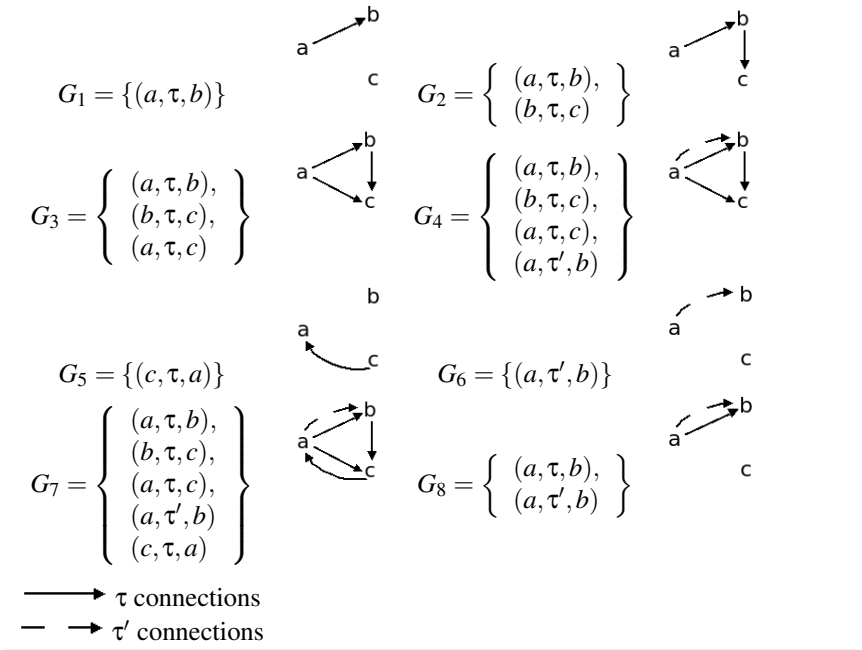


Figure 4.1: Examples of *NCP* network topologies

An *NCP* process is called a *coordination policy*. We use the word policy to emphasize the fact that the calculus has been introduced to specify and constrain the behavior of *SC* networks. The syntax of coordination policies is defined in Table 4.2. Non-deterministic (guarded) choice is denoted as $\sum$; a policy $p@a.P$ represents an action p executed by the component a with continuation P; prefix $\tau(\tau')$ allows to receive on $\tau \in \mathcal{T}$ and is called *lambda input* since it corresponds to *SC* lambda reactions; $\tau\ \tau'$ allows to receive signals having topic τ and session τ' and is therefore called *check input*. Since a lambda input can handle events regardless their sessions. the name τ' represents a binder for the received session identifier. The policy $\overline{\tau}\ \tau'$ raises an event on session τ' with topic τ. The component delivers the corresponding notifications to all services that are subscribed on the topic τ. The *envelope* $\langle \tau \circledcirc \tau' \rangle @a$ represents a pending message/notification on the network targeted to a. Notice that only the target of the envelope is declared. The communication model of *NCP* is strictly related to *SC*. In fact, the emission

- $G(a)$ are the *flows emanating from a in G*, namely $G(a) = \{(\tau, b) \mid (a, \tau, b) \in E\}$;
- $G \cap a$ is the *sub-topology of G involving a*, namely $G \cap a = \{(a, \tau, b) \in E\} \cup \{(b, \tau, a) \in E\}$
- $G(\tau)$ is the *topic graph of* τ *in G*, namely the unlabeled directed graph such that $|G(\tau)| = |G|$ and the edges are $\{(a,b) \in A \times A \mid (a, \tau, b) \in E\}$ (hereafter, we let T range on such graphs for which $\tau \boxdot T = \{(a, \tau, b) \mid (a,b) \in T\}$);
- $G(a, \tau) = \{b \mid (\tau, b) \in G(a))\}$ is the *flow projection of* τ *for a in G*.
- $a \boxtimes F = \{(a, \tau, b) \mid (\tau, b) \in F\})$, for $F \subseteq \mathcal{T} \times A$;
- if $G' = (V', E')$ is a network topology, $G \uplus G' = (V \cup V', E \cup E')$ and $G \setminus G' = (V, E \setminus E')$.
- the free names of G ($fn(G)$) and its bound names ($bn(G)$) are defined as:
  - $bn(G) = V$
  - $fn(G) = \{a \mid (a, \tau, b) \in E\} \cup \{\tau \mid (a, \tau, b) \in E\} \cup \{b \mid (a, \tau, b) \in E\} \setminus bn(G)$

Table 4.1: *NCP* auxiliary notations

of an event and its reception are performed by two phases. Initially, the emitter spawns into the network the proper envelopes, according with the actual network topology. Subsequently, a subscriber can react to the envelope targeted to him. The policy $\mathtt{fupd}(F)$ adds F to the flows departing from a. Prefix $\iota.P$ represents the execution of an internal activity before the execution of P. The name restrictions, namely $(\nu\ \tau : T)P$ and $(\nu\ a : G)P$ restrict τ and a in P; noteworthy, graph T permits to extend the topology with the connections among components for the fresh topic τ, while the network topology G yields the flows from/to a. Finally, coordination policies can be composed in parallel. (Free names $fn(P)$ and bound names $bn(P)$ are defined as expected.) Notice that, differently from *SC*, the *NCP* policies involves components, but are not boxed inside them, to permit to express coordination from a global point of view. For example $\overline{\tau}\ \tau'@a.\overline{\tau}\ \tau'@a$ is a valid *NCP* policy.

$$
\begin{array}{rcl}
P & ::= & \sum_{i\in I} p_i@a_i.P_i \;\mid\; \bar{\tau}\,\tau'@a.P \;\mid\; \langle \tau_{\circledcirc}\tau' \rangle @a \\
 & \mid & \mathtt{fupd}\,(F)\,@a.P \;\mid\; \imath.P \;\mid\; P \parallel P \\
 & \mid & (\nu\,\tau : T)\,P \;\mid\; (\nu\,a : G)\,P \\
\text{where} p & ::= & \tau(\tau') \;\mid\; \tau\,\tau'
\end{array}
$$

where $\tau, \tau' \in \mathcal{T}$, $a \in A$, T is a topic graphs and I is a finite index set and $\sum_{i\in I} p_i@a_i.P_i = \mathbf{0}$ when $I = \emptyset$

Table 4.2: *NCP* policies syntax

Let G be an *NCP* topology and P an *NCP* policy, then the pair $\langle G\,;\,P\rangle$ is called *NCP* state. Free and bound names of *NCP* states are defined trivially:

$$
\begin{array}{c}
fn(\langle G\,;\,P\rangle) = fn(P) \cup fn(G) \setminus bn(G) \\
bn(\langle G\,;\,P\rangle) = bn(P) \setminus fn(G) \cup bn(G)
\end{array}
$$

NCP states can represent the specification of a system. Several states can be defined starting from a policy, according with the topology in which the policy is embedded.

A key feature of *NCP* is that network topologies are first order entities. Many other process calculi have been designed to deal with process distribution. The novel feature of *NCP* is given by the capability to naturally restrict a part of the network topology. The following examples clarify the intuition of *NCP* name restriction:

- $\langle G_1\,;\,(\nu\,\tau : \{(a,b)\})\,(P) \parallel P'\rangle$, where G_1 is the network topology of Figure 4.1, represents a specification composed by two concurrent policies. The policy P' know only the linkage described by G_1 and have no notion regarding τ', because it is restricted. Instead, the policy P know both the topic τ' and the corresponding topic graph. Informally, the specification will evaluate P under the extended network topology $G_8 \uplus G_1$ of Figure 4.1.
- $\left\langle G_8\,;\,\left(\nu\,c : \left\{ \begin{array}{c} (a,\tau,c) \\ (b,\tau,c) \\ (c,\tau,a) \end{array} \right\}\right)(P) \parallel P' \right\rangle$, where G_8 is the network topology of Figure 4.1, represents a specification with two concurrent policies. P' knows the linkage G_8 and has no knowledge about the component c. Instead, P knows both the component c and all connection departing from or targeted to it. Informally, P will be evaluated under the topology G_7 of Figure 4.1.

$$Sbj((\vec{b},E)) = \{a \mid \exists c,\tau.(a,\tau,c) \in E\}$$

$$\begin{array}{llll}
Sbj(\mathbf{0}) & = \emptyset & Sbj(\iota.P) & = Sbj(P) \\
Sbj(P_1 \parallel P_2) & = Sbj(P_1) \cup Sbj(P_2) & Sbj(\mathtt{fupd}\,(F)\,@a.P) & = Sbj(P) \cup \{a\} \\
Sbj(\langle \tau_{\circledcirc} \tau' \rangle @a) & = \emptyset & Sbj(\overline{\tau}\ \tau'@a.P) & = Sbj(P) \cup \{a\} \\
Sbj(\tau\ \tau'@a.P) & = Sbj(P) \cup \{a\} & Sbj(\tau(\tau')\ @a.P) & = Sbj(P) \cup \{a\}
\end{array}$$

$$Sbj((\nu\ \tau : T)\,P) = Sbj(P) \cup \{b \mid \exists c.(b,c) \in T\}$$

$$Sbj((\nu\ a : G)\,P) = Sbj(G) \cup Sbj(P) \setminus \{a\}$$

$$Sbj(\langle G\ ;\ P \rangle) = Sbj(G) \cup Sbj(P)$$

Table 4.3: *NCP* subjects

The network topology G in the component name restriction should contain only linkage departing from or targeted to the restricted component. We introduce a well formed constraint.

Definition 8 *An* NCP *policy is well formed is each contained component name restriction* $(\nu\ a : G)\,P$ *satisfies* $G \cap a = G$.

For example, $(\nu\ a : \{(b,\tau,c)\})\,P$ is not a well formed policy.

The *NCP* states, policies and network topologies can involve component names in two different ways: either by describing the behavior of the component (e.g. the policy $\overline{\tau}\ \tau'@a$) or by describing which event are notified to it (e.g. the policy $\langle \tau_{\circledcirc} \tau' \rangle @a$). We refer to the components whose behavior is described as *subject*.

Definition 9 *The subjects* $(Sbj(*))$ *of a network topology* $G = (\vec{b},E)$*, of a policy* P *and of a* NCP *state* $\langle G\ ;\ P \rangle$ *is the set component names obtained by the rules in Table 4.3.*

Definition 10 *Let* X *and* Y *range over* NCP *terms (states, policies and network topologies). We say that they predicate on distinct subjects (and we write* $X \perp Y$*) if* $Sbj(Y) \cap Sbj(Y) = \emptyset$.

4.1.1 Semantics of *NCP*

Though *NCP* is reminiscent of the asynchronous π-calculus, its semantics is centered on network topologies, that is the environment of the computation (while in the π-calculusit is implicit in the knowledge about channels). This enables us to model in a natural way multi-cast communication: for example, in order to receive $\overline{\tau}\,\tau'$, it is not sufficient to listen on τ, but it is necessary that the network topology has a "τ-connection" between listener and emitter.

The semantics of *NCP* is specified by a labeled transition system (LTS) inspired by the HT-LTS presented in Section 2.5. The labels α are defined by the following grammar:

$$\alpha \quad ::= \quad \varepsilon \;\mid\; \tau\,\tau'@a \;\mid\; (\tau\,\tau'@a) \;\mid\; \langle \tau_{\circledcirc}\tau' \rangle @a \;\mid\; \langle \tau_{\circledcirc}(\tau' : T) \rangle @a$$

where $\tau, \tau' \in \mathcal{T}$, $a \in A$ and T is a topic graph

Table 4.4: *NCP* actions

The action ε is the silent action, representing unobservable activities like internal communications. The action $\tau\,\tau'@a$ is a *free reaction activation* . The action $(\tau\,\tau'@a)$ represents the reception of a message that will be spawned in parallel with the current process (this action is observable in any system, including the empty policy). The action $\langle \tau_{\circledcirc}\tau' \rangle @a$ is the *free* (asynchronous) event notification of kind τ, session τ' and destination a. A key feature of *NCP* is the interplay between restriction of topics and multi-cast communications. Indeed, in *NCP* the extrusion of a topic τ' enriches the receiver policy with topologies that were absent before. Hence, further emissions of signals on τ' from the policy itself will generate envelopes according with the inherited linkage. The action $\langle \tau_{\circledcirc}(\tau' : T) \rangle @a$ is a *bound* event notification on τ of a topic τ' with network graph T. The linkage T is exploited to inform the receiver about the actual state of the network topology for the extruded topic τ'. Hereafter, $n(\alpha)$ will denote the names of α.

The observational semantics of *NCP* is given by the transitive closure of $\equiv \xrightarrow{\alpha} \equiv$ where $\xrightarrow{\alpha}$ is the smallest relation closed under the rules in Tables 4.6 and 4.7 (where $\vec{a}, \vec{b}, \ldots$ range over subsets of A), which rely on the congruence rule in Table 4.5.

The structural congruence rule permits scope extrusion of component name. Notice that the side condition $G' \perp P'$ ensures that the network topology (G') involving the component name restricted (b) must not contain any component as

$$\frac{b \notin fn(G) \cup fn(P') \quad G' \perp P'}{\langle (\vec{a} \cup \{b\}, G \cup G') \; ; \; P \parallel P' \rangle \equiv \langle (\vec{a}, G) \; ; \; (\nu \, b : G') \, P \parallel P' \rangle}$$

Table 4.5: *NCP* congruence rule

starting edge whose behavior is described in P'.

The observational semantics of an *NCP* policy depends on and can affect the network topology. We use $\langle G \; ; \; P \rangle \xrightarrow{\alpha} \langle G' \; ; \; P' \rangle$ to represent that the coordination policy P, plugged into the topology G, by performing the action α evolves to the policy P' and changes the topology to G'.

Rule `skip` trivially fires the silent action. Rule `fupd` changes the network topology, by appending the sub-network $a \boxtimes F$ to the environment G; notice that newly added flows departs only from a. Rule `emit` allows multi-casting communications: it spawns in the network an envelope for each subscriber in $G(\tau)(a)$; notice that the continuation policy P is executed regardless the reception of envelopes as typical in asynchronous communications. Notification of envelopes is ruled by `notify` as much like as the output in the asynchronous π-calculus. Rules `lambda` and `check` model input actions. In the former, the selected input p_j reads any signal with topic τ and binds τ_1 to τ_1' in an early-style semantics. When a check input is selected, only envelopes of topic τ in session τ_1 can be consumed. Notice that the reception by a check reaction of a topic cannot change the network topology, because the two topics involved by the communication are already known. The reception of a fresh name (τ_1') by a lambda reaction, instead, can extend the environment knowledge of the component; namely the receiver can discover all the existing linkage involving the received name τ_1'. In the spirit of early-style semantics, we allow the rule to extend the topology with any possible graph (T). Differently from *SC*, this two rules permit to express external non-deterministic choice and can involve several components. Notice that, after the communication has occurred, all competitor inputs are garbaged and that a lambda inputs is a *singleton*; it is not removed after the reception of an envelope.

Rule `async` permits to any *NCP* state to perform an input, simply storing the received message for subsequent usages, allowing to arbitrarily delay the communication. This rule is inspired by the rule in_0 presented in Section 2.5.

Rules `open` and `close` govern scope extrusion of a topics. We recall that the input transitions could been generated by the rule `lambda` om an early-style semantics. Informally, the session τ' and the topology carried by the envelope

$$(\texttt{skip})\ \ \langle G\ ;\ \iota.P\rangle \xrightarrow{\varepsilon} \langle G\ ;\ P\rangle$$

$$(\texttt{fupd})\ \ \langle G\ ;\ \texttt{fupd}\,(F)\,@a.P\rangle \xrightarrow{\varepsilon} \langle G \uplus (a \boxtimes F)\ ;\ P\rangle$$

$$(\texttt{emit})\ \ \langle G\ ;\ \bar{\tau}\,\tau'@a.P\rangle \xrightarrow{\varepsilon} \left\langle G\ ;\ P \parallel \prod_{b\in G(\tau,a)} \langle \tau \circledcirc \tau'\rangle @b \right\rangle$$

$$(\texttt{notify})\ \ \langle G\ ;\ \langle \tau \circledcirc \tau'\rangle @a\rangle \xrightarrow{\langle \tau \copyright \tau'\rangle @a} \langle G\ ;\ \mathbf{0}\rangle$$

$$\frac{j \in I \qquad p_j = \tau(\tau_1)}{\left\langle G\ ;\ \sum_{i\in I} p_i @ a_i.P_i \right\rangle \xrightarrow{\tau\,\tau_1'@a_j} \left\langle G \uplus \tau_1' \boxdot T\ ;\ \{\tau_1'/\tau_1\}P_j \parallel p_j @ a_j.P_j \right\rangle}\ (\texttt{lambda})$$

$$\frac{j \in I \qquad p_j = \tau\,\tau'}{\left\langle G\ ;\ \sum_{i\in I} p_i @ a_i.P_i \right\rangle \xrightarrow{p_j @ a_j} \langle G\ ;\ P_j\rangle}\ (\texttt{check})$$

$$\frac{}{\langle G\ ;\ P\rangle \xrightarrow{(\tau\,\tau'@a)} \left\langle G\ ;\ P \parallel \langle \tau \circledcirc \tau'\rangle @a \right\rangle}\ (\texttt{async})$$

Table 4.6: *NCP* labelled transition rules

allows the rules `open` and `close` to chose between all input transitions.

Rule `new` permits to extend the topology with a freshly generated topic provided that it is not extruded ($\tau \notin n(\alpha)$) and hides the changes to the environment that involve the name outside the scope $G' \setminus (\tau \boxtimes T')$. Rule `com` allows the communication of a free session name τ'. Notice that the rule can choose only the input transitions that does not affect the topology, because the communicated session name is free. Finally, Rule `par` has the standard meaning.

Theorem 1 *Let* $\langle G\ ;\ P\rangle$ *be a* NCP *state and* G_1 *a network topology, if* $P \perp G_1$ *then*

$$\langle G\ ;\ P\rangle \xrightarrow{\alpha} \langle G'\ ;\ P'\rangle \textit{ if and only if } \langle G \uplus G_1\ ;\ P\rangle \xrightarrow{\alpha} \langle G' \uplus G_1\ ;\ P'\rangle$$

The subjects of a network topology ($Sbj(G_1)$) represents the set of components

$$\frac{\tau' \notin fn(G) \quad \langle G \uplus (\tau' \boxtimes T)\ ;\ P\rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle G \uplus (\tau' \boxtimes T\ ;\ P'\rangle}{\langle G\ ;\ (\nu\ s : T)\, P\rangle \xrightarrow{\langle \tau \copyright (\tau' : T) \rangle @a} \langle G \uplus (\tau' \boxtimes T\ ;\ P'\rangle} \ (\texttt{open})$$

$$\frac{\begin{array}{c}\tau' \notin fn(\langle G\ ;\ P_1\rangle) \\ \langle G\ ;\ P_1\rangle \xrightarrow{\tau\ \tau' @a} \langle G \uplus \tau' \boxdot T\ ;\ P_1'\rangle \\ \langle G\ ;\ P_2\rangle \xrightarrow{\langle \tau \copyright (\tau' : T) \rangle @a} \langle G \uplus (\tau' \boxtimes T)\ ;\ P_2'\rangle\end{array}}{\langle G\ ;\ P_1 \parallel P_2\rangle \xrightarrow{\varepsilon} \langle G\ ;\ (\nu\ \tau' : T)\,(P_1' \parallel P_2')\rangle} \ (\texttt{close})$$

$$\frac{\langle G \uplus (\tau \boxtimes T)\ ;\ P\rangle \xrightarrow{\alpha} \langle G'\ ;\ P'\rangle \qquad \tau \notin n(\alpha) \cup fn(G) \qquad T' = G'(\tau)}{\langle G\ ;\ (\nu\ \tau : T)\, P\rangle \xrightarrow{\alpha} \langle G' \setminus (\tau \boxtimes T')\ ;\ (\nu\ \tau : T')\, P'\rangle} \ (\texttt{new})$$

$$\frac{\langle G\ ;\ P_1\rangle \xrightarrow{\tau\ \tau' @a} \langle G\ ;\ P_1'\rangle \quad \langle G\ ;\ P_2\rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle G\ ;\ P_2'\rangle}{\langle G\ ;\ P_1 \parallel P_2\rangle \xrightarrow{\varepsilon} \langle G\ ;\ P_1' \parallel P_2'\rangle} \ (\texttt{com})$$

$$\frac{\langle G\ ;\ P\rangle \xrightarrow{\alpha} \langle G'\ ;\ P'\rangle}{\langle G\ ;\ P \parallel P_1\rangle \xrightarrow{\alpha} \langle G'\ ;\ P' \parallel P_1\rangle} \ (\texttt{par})$$

Table 4.7: *NCP* labelled transition rules

whose the topology specifies the flows. The statement $P \perp G_1$ means that the topology added to the specification ($\langle G \uplus G_1\ ;\ P\rangle$) does not add any flow to the components whose behavior has been described. This theorem provides a sufficient condition on the semantic context (aka the network topology added), which guarantees that the behavior of the system is not affected. The theorem is proved in Appendix A.1 using induction over the *NCP* transition rules.

Theorem 2 *Let* $\langle G\ ;\ P\rangle$ *be an* NCP *state, such that* $\langle G\ ;\ P\rangle \xrightarrow{\alpha} \langle G'\ ;\ P'\rangle$. *If* $G' = G$ *or* $\alpha \neq \tau\ \tau' @a$, *then* $Sbj(\langle G'\ ;\ P\rangle) \subseteq Sbj(\langle G\ ;\ P\rangle)$.

The theorem provide a linearity statement for the subjects of policies. Since component names cannot be communicated, a new component can be discovered only

$$S_1 = \langle G \,;\, \tau(\tau')\,@a.\imath.\overline{\tau_1}\,\tau'@b \rangle$$

Table 4.8: *NCP* hidden communications

by the extrusion of an hidden network topology by the reception of a fresh topic. If the input does not affect the network topology ($G' = G$) it is straightforward to verify that the subjects of the specification can be bounded statically.

4.1.2 Examples

In this Section we highlight the main features of *NCP* semantics with some illustrative examples.

Hidden communications

Let G be a topic topology. The *NCP* state in Figure 4.8 specifies a system having two components, named a and b. Intuitively, the component a can receive events having topic τ. After some internal activities, the component b must raise an event having the same session (τ') of the one received by a. Notice that a system implementing this specification must involve a communication of the name of session τ' between a and b, however, this communication is not explicitly represented.

The operational rules `lambda`, `skip` and `emit` detail the required behavior. The `lambda` rule handles the early instantiation of the input, allowing the transition for any name τ''. Notice that the lambda reaction remains active and that the envelopes spawned by the component b have the same session of the received one. The derivation is given below:

$$\langle G \,;\, \tau(\tau')\,@a.\imath.\overline{\tau_1}\,\tau'@b \rangle$$
$$\xrightarrow{(\tau\ \tau''@a)}\xrightarrow{\varepsilon}\xrightarrow{\varepsilon}$$
$$\langle G \,;\, \tau(\tau'')\,@a.\imath.\overline{\tau_1}\,\tau'@b \parallel \textstyle\prod_{c \in G(b,\tau_1)} \langle \tau_1 \circledcirc \tau'' \rangle @c \rangle$$

Scope of topology

Let $G = \{(a, \tau, b)\}$ the topology describing a single connection from the component a to the component b for the topic τ. Let us consider the *NCP* state in Figure 4.9 Initially, the topology for the topic τ' is hidden outside the right part

$$\left\langle G\ ;\ \tau(\tau_1)\,@b.\overline{\tau_1}\ \tau_s@a \parallel ((\nu\ \tau' : \emptyset)\,\mathtt{fupd}\left(\{(\tau', b)\}\right)\,@a.\langle \tau \circledcirc \tau' \rangle @b)\right\rangle$$

Table 4.9: *NCP* scope of topology

of the parallel policy, because τ' is restricted. The system starts updating the connections of the component a for the topic τ'. Since this update of the network topology cannot be visible to the left policy, the new linkages are not stored into G and then are confined into the restriction:

$$\dfrac{\dfrac{\dfrac{\langle G\ ;\ \mathtt{fupd}\,(\{(\tau', b)\})\,@a.\langle \tau \circledcirc \tau' \rangle @b\rangle}{\xrightarrow{\varepsilon}}}{\dfrac{\langle G \uplus \{(a, \tau, b)\}\ ;\ \langle \tau \circledcirc \tau' \rangle @b\rangle}{\langle G\ ;\ (\nu\ \tau' : \emptyset)\,\mathtt{fupd}\,(\{(\tau', b)\})\,@a.\langle \tau \circledcirc \tau' \rangle @b\rangle}}{\xrightarrow{\varepsilon}}}{\dfrac{\dfrac{\langle G\ ;\ (\nu\ \tau' : \{(a, b)\})\,\langle \tau \circledcirc \tau' \rangle @b\rangle}{\langle G\ ;\ \tau(\tau_1)\,@b.\overline{\tau_1}\ \tau_s@a \parallel ((\nu\ \tau' : \emptyset)\,\mathtt{fupd}\,(\{(\tau', b)\})\,@a.\langle \tau \circledcirc \tau' \rangle @b)\rangle}}{\xrightarrow{\varepsilon}}}{\langle G\ ;\ \tau(\tau_1)\,@b.\overline{\tau_1}\ \tau_s@a \parallel ((\nu\ \tau' : \{(a, b)\})\,\langle \tau \circledcirc \tau' \rangle @b)\rangle}$$

Now the two parallel policies can communicate, extruding τ'. The reception of the envelope by the left policy (by the lambda reaction) performs also the extrusion of the topology associated to τ'. Hence, a can emit signals having τ' to the recipient b.

$$\dfrac{\dfrac{\langle G\ ;\ \tau(\tau_1)\,@b.\overline{\tau_1}\ \tau_s@a\rangle}{\xrightarrow{\tau\ \tau'@b}}\qquad \dfrac{\dfrac{\langle G \uplus \{(a, \tau', b)\}\ ;\ \langle \tau \circledcirc \tau' \rangle @b\rangle}{\xrightarrow{\langle \tau \circledcirc \tau' \rangle @b}}}{\dfrac{\langle G \uplus \{(a, \tau', b)\}\ ;\ \mathbf{0}\rangle}{\langle G\ ;\ (\nu\ \tau' : \{(a, b)\})\,\langle \tau \circledcirc \tau' \rangle @b\rangle}}{\xrightarrow{\langle \tau \circledcirc \tau' : \{(a,b)\} \rangle @b}}}{\dfrac{\left\langle G \uplus \{(a, \tau', b)\}\ ;\ \overline{\tau'}\ \tau_s@a\right\rangle \qquad \langle G \uplus \{(a, \tau', b)\}\ ;\ \mathbf{0}\rangle}{\langle G\ ;\ \tau(\tau_1)\,@b.\overline{\tau_1}\ \tau_s@a \parallel ((\nu\ \tau' : \{(a, b)\})\,\langle \tau \circledcirc \tau' \rangle @b)\rangle}}{\xrightarrow{\varepsilon}}$$
$$\left\langle G\ ;\ (\nu\ \tau' : \{(a, b)\})\,(\overline{\tau'}\ \tau_s@a \parallel \mathbf{0})\right\rangle$$

Notice that when the rule `close` is applied, the bound event notification chooses the right input transition, namely, the one the extends the network topology with

the right graph for the extruded topic.

4.1.3 Bisimulation Semantics

We introduce the notion of observational equivalence for *NCP*. Honda-Tokoro and Amadio et alia have studied bisimilarity for asynchronous calculi [54; 55]. We use these results (in particular, the directed HT labeled transition systems presented in Section 2.5) to define our bisimulation semantics.

Following this approach, in the bisimulation game, any process can act as a buffer that reads any possible message and stores it without consuming the message. This is done, in our case, by rule `async`. On the other hand, "effective" inputs that actually consume messages are not observed at all in the bisimulation game, whereas synchronizations induced by these inputs are. Thus, in defining bisimilarity, we keep into account the transitions induced by the rule `async`, but not those obtained by `check` or `lambda`.

Definition 11 *A symmetric binary relation $\mathcal{B}$ over* NCP *states is an* NCP*-bisimulation if whenever* $\langle G_1 \,;\, P_1\rangle\, \mathcal{B}\, \langle G_2 \,;\, P_2\rangle$ *and* $\langle G_1 \,;\, P_1\rangle \xrightarrow{\alpha} \langle G_1' \,;\, P_1'\rangle$

- *if* $\alpha \in \{\varepsilon, \langle \tau_{\copyright} \tau'\rangle @a, (\tau\,\tau'@a)\}$ *and* $a \notin bn(G_1)$*, there is* $\langle G_2 \,;\, P_2\rangle \xrightarrow{\alpha} \langle G_2' \,;\, P_2'\rangle$ *and* $\langle G_1' \,;\, P_1'\rangle\, \mathcal{B}\, \langle G_2' \,;\, P_2'\rangle$
- *if* $\alpha = \langle \tau_{\copyright}(\tau' : T)\rangle @a$ *with* $\tau' \notin fn(G_2, P_2)$ *and* $a \notin bn(G_1)$*,*
  there is $\langle G_2 \,;\, P_2\rangle \xrightarrow{\langle \tau \copyright (\tau' : T')\rangle @a} \langle G_2' \,;\, P_2'\rangle$ *and* $\langle G_1' \,;\, P_1'\rangle\, \mathcal{B}\, \langle G_2' \,;\, P_2'\rangle$.

The bisimilarity relation is obtained as usual and denoted by $\sim$. The definition of weak bisimulation is defined in the standard way by considering the *weak transition relation* defined as the union of $\Longrightarrow$ and $\bigcup_{\alpha \neq \varepsilon} \Longrightarrow \xrightarrow{\alpha} \Longrightarrow$, where $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\varepsilon}$. We define $\approx$ as the largest weak bisimulation.

A key difference between *NCP* and the asynchronous π-calculus is the awareness of topic topologies in the semantics. However, it would be too restrictive to require that only policies with the same topology can be bisimilar. For example, the empty network is bisimilar to itself under any topology. This is also reflected in the definition of the clause for the bound output: when a bound output transition is matched in the bisimulation relation, the two hidden topologies associated to the transition are not taken in account, and, therefore, can be different.

The following theorems characterize properties of the *NCP* bisimulation and are used to define the compositionality of *NCP*.

Theorem 3 *Let* $\langle G_1 \,;\, P_1\rangle$ *and* $\langle G_2 \,;\, P_2\rangle$ *be two* NCP *state such that* $\langle G_1 \,;\, P_1\rangle \sim \langle G_2 \,;\, P_2\rangle$ *and* $\langle G_1 \,;\, P_1\rangle \xrightarrow{\tau\ \tau'@a} \langle G_1 \,;\, P_1'\rangle$*, then and one of the following two statements must hols:*

- $\langle G_2 \,;\, P_2\rangle \xrightarrow{\tau\ \tau'@a} \langle G_2 \,;\, P_2'\rangle$ *and* $\langle G_1 \,;\, P_1'\rangle \sim \langle G_2 \,;\, P_2'\rangle$
- $\langle G_2 \,;\, P_2\rangle \xrightarrow{\varepsilon} \langle G_2' \,;\, P_2'\rangle$ *and* $\langle G_1 \,;\, P_1'\rangle \sim \langle G_2' \,;\, P_2' \parallel \langle \tau_{\circledcirc}\tau'\rangle@a\rangle$

We remark that *NCP* bisimulation does not take into account input transitions, since it has been studied for an asynchronous calculus. However, the theorem shows how the bisimulation characterizes two bisimilar states, when one of them is ready to perform an input. Informally, two cases are possibles; both states are ready to receive the same event notification, or the receiver one *re-spawns* the received notification immediately. The proof of the theorem is given in Appendix A.3 and exploits the `async` transition rule to compose the two policies with the same envelopes an to infer their behavior.

Theorem 4 *Let* $\langle G \,;\, P\rangle$ *and* $\langle G' \,;\, P'\rangle$ *be two* NCP *states such that* $\langle G \,;\, P\rangle \sim \langle G' \,;\, P'\rangle$*,* $\tau \in \mathcal{T}$ *be a topic,* $T = G(\tau)$ *and* $T' = G'(\tau)$ *be the topic graphs of* τ *in the two network topologies* G *and* G' *respectively, then:*

1. *if* $G = (\vec{b}, E)$ *and* $G' = (\vec{b}', E')$ *then* $\left\langle (\vec{b} \cup \vec{a}, E) \,;\, P \right\rangle \sim \left\langle (\vec{b}' \cup \vec{a}, E') \,;\, P' \right\rangle$
2. $\langle G \setminus \tau \boxdot T \,;\, (\nu\ \tau : T)\, P\rangle \sim \langle G' \setminus \tau \boxdot T' \,;\, (\nu\ \tau : T')\, P'\rangle$

Theorem 4 describes which changes to the network topologies (G and G') of two *NCP* states can be applied without violating their bisimilarity:

1. the restriction of the set of component names $\vec{a}$ for both topologies, making all components $\vec{a}$ not reachable from outside agents,
2. the restriction of a topic name τ, without changing the internal topology, inhibiting outside agents to notify this kind of events.

The proof of the theorem is reported in Appendix A.4. Our strategy is to check that the following relations are *NCP*-bisimulations:

$$\mathcal{B} = \left\{ \left(\left\langle (\vec{b} \cup \vec{a}, E) \,;\, P \right\rangle, \left\langle (\vec{b}' \cup \vec{a}', E') \,;\, P' \right\rangle \right) \mid \left\langle (\vec{b}, E) \,;\, P \right\rangle \sim \left\langle (\vec{b}', E') \,;\, P' \right\rangle \right\}$$

$$\mathcal{B} = \left\{ \begin{array}{c} (\langle G \setminus \tau \boxdot T \,;\, (\nu\ \tau : T)\, P\rangle, \langle G' \setminus \tau \boxdot T' \,;\, (\nu\ \tau : T')\, P'\rangle) \mid \\ \langle G \,;\, P\rangle \sim \langle G' \,;\, P'\rangle, T = G(\tau) \text{ and } T' = G'(\tau) \end{array} \right\} \cup \sim$$

Theorem 5 *Let* $S_{P_1} = \langle G_1 \,;\, P_1 \rangle$, $S_{P_2} = \langle G_2 \,;\, P_2 \rangle$, $S_{Q_1} = \langle I_2 \,;\, Q_2 \rangle$ *and* $S_{Q_2} = \langle I_2 \,;\, Q_2 \rangle$ *be* NCP *states, such that* $S_{P_1} \sim S_{Q_2}$ *and* $S_{Q_1} \sim S_{Q_2}$ *If* $S_{P_1} \perp S_{Q_1}$ *and* $S_{P_2} \perp S_{Q_2}$ *then* $\langle G_1 \uplus I_1 \,;\, P_1 \parallel Q_1 \rangle \sim \langle G_2 \uplus I_2 \,;\, P_2 \parallel Q_2 \rangle$

The composition of two *NCP* states (representing two formal specifications) is obtained by the union of their network topologies ($G_1 \uplus I_1$ and $G_2 \uplus I_2$) and the parallel composition of their policies ($P_1 \parallel Q_1$ and $P_2 \parallel Q_2$). Starting from two bisimilar ncp states (S_{P_1} and S_{P_2}), their compositions with a new specifications yield their bisimililarity equivalence under the assumption that $S_{P_1} \perp S_{Q_1}$ and $S_{P_2} \perp S_{Q_2}$. The theorem provides the sufficient condition on which relies the core of compositinality of *NCP*. We prove the theorem in Appendix A.5 by showing that the following relation is a bisimulation:

$$\mathcal{B} = \left\{ \begin{array}{r} (\langle G_1 \uplus I_1 \,;\, P_1 \parallel Q_1 \rangle, \langle G_2 \uplus I_2 \,;\, P_2 \parallel Q_2 \rangle) \\ \mid \langle G_1 \,;\, P_1 \rangle \sim \langle G_2 \,;\, P_2 \rangle \wedge \langle I_1 \,;\, Q_1 \rangle \sim \langle I_2 \,;\, Q_2 \rangle \\ \wedge\, Sbj(\langle G_1 \,;\, P_1 \rangle) \cap Sbj(\langle I_1 \,;\, Q_1 \rangle) = \emptyset \\ \wedge\, Sbj(\langle G_2 \,;\, P_2 \rangle) \cap Sbj(\langle I_2 \,;\, Q_2 \rangle) = \emptyset \end{array} \right\}$$

4.2 Checking Choreography

We introduce a formal methodology to verify correctness of a network of *SC* components against global coordination policies as given by *NCP* specifications. The first step of our methodology consists of providing an encoding from *SC* networks to *NCP* policies. The basic idea of the encoding is to transform *SC* reductions into *NCP* transitions labeled with ε.

The encoding of an *SC* network into an *NCP* state exploits the following functions:

- the function $[\![B]\!]_a$ defined in Table 4.10 , which takes a *SC* behavior B, localized within the component a, and maps it into an *NCP* policy
- the function $[\![R]\!]_a$ defined in Table 4.11, which takes a *SC* reaction R, installed in the interface of the component a, and maps it into a policy
- the function $[\![N]\!]$ defined in Table 4.12, which takes a *SC* network N and maps it into a *NCP* state.

The correctness of the encoding is "up-to" bisimilarity as shown by the following theorem.

Theorem 6 *Let* N *and* N' *be* SC *networks. It holds that* $N \rightarrow N'$ *if and only if* $[\![N]\!] \xrightarrow{\varepsilon} (G,P)$ *and* $(G,P) \sim [\![N']\!]$

$$[\![\varepsilon;B]\!]_a = \iota.[\![B]\!]_a$$

$$[\![0]\!]_a = \mathbf{0} \qquad [\![B \mid B']\!]_a = [\![B]\!]_a \parallel [\![B']\!]_a$$

$$[\![(\nu\tau)B]\!]_a = (\nu\ \tau : \emptyset)\,[\![B]\!]_a \qquad [\![\mathtt{out}\langle\tau_{\circledcirc}\tau'\rangle B]\!]_a = \overline{\tau}\ \tau'@a.\,[\![B]\!]_a$$

$$[\![\mathtt{rupd}\,(R)\,;B]\!]_a = \iota.\,[\![R]\!]_a \parallel [\![B]\!]_a \qquad [\![\mathtt{fupd}\,(F)\,;B]\!]_a = \mathtt{fupd}\,(F)\,@a.\,[\![B]\!]_a$$

Table 4.10: Encoding the behavior B executed within the component a: $[\![B]\!]_a$

$$[\![0]\!]_a = \mathbf{0} \qquad [\![R|R']\!]_a = [\![R]\!]_a \parallel [\![R']\!]_a$$

$$[\![\tau_{\circledcirc}\tau'_{\circledcirc}B\gg]\!]_a = \tau\ \tau'@a.\,[\![B]\!]_a \qquad [\![\tau\ \lambda\ \tau' \gg B]\!]_a = \tau\,(\tau')\,@a.\,[\![B]\!]_a$$

Table 4.11: Encoding the reaction R installed in the interface of the component a: $[\![R]\!]_a$

The proof of the theorem is described in Appendix B.5

The theorem allows us to derive the choreography model of a *SC* network. The next step of our methodology consists of making verification to be *compositional*. Once a choreography has been verified, it should be possible to "plug" it into a distributed network of components, without altering verified properties. This is formalized in the rest of this section.

First, we have to define *SC* network contexts. We use the notions of *occurrence* of a symbol in a term, and of substitution that can be defined in the standard way. The set $\mathcal{C}$ of one-hole *SC* network contexts is defined as the least subset of terms generated by the grammar in Table 4.13, where the number of occurrences of the placeholder $*$ is one.

Assume that $C \in \mathcal{C}$, and let N be a *SC* network. The application $C[N]$ of C to N is defined as the syntactic substitution of the single occurrence of $*$ in C with N. As usual we focus only on $C[N]$ well formed. We have the following compositionality result.

Theorem 7 *Let N_1 and N_2 be* SC *networks such that $[\![N_1]\!] \sim [\![N_2]\!]$. For all $C \in \mathcal{C}$, it holds that $[\![C[N_1]]\!] \sim [\![C[N_2]]\!]$.*

$$[\![\emptyset]\!] = \langle \mathbf{0} \,;\, \mathbf{0} \rangle \qquad [\![\langle \tau_{\circledcirc} \tau' \rangle @ a]\!] = \langle \mathbf{0} \,;\, \langle \tau_{\circledcirc} \tau' \rangle @ a \rangle$$

$$\frac{[\![N]\!] = \langle G \,;\, P \rangle \quad [\![N']\!] = \langle G' \,;\, P' \rangle}{[\![N \parallel N']\!] = \langle G \uplus G' \,;\, P \parallel P' \rangle} \qquad \frac{[\![N]\!] = \langle G \,;\, P \rangle \quad T = G(\tau)}{[\![(\nu \tau) N]\!] = \langle G \setminus (\tau \boxdot T) \,;\, (\nu\, \tau : T)\, P \rangle}$$

$$\frac{[\![N]\!] = \left\langle (\vec{b}, G) \,;\, P \right\rangle \quad a \notin \vec{b}}{[\![(\nu a) N]\!] = \left\langle (\vec{b} \cup \{a\}, G) \,;\, P \right\rangle}$$

$$\left[\!\left[a\,[B]_F^R \right]\!\right] = \langle G \,;\, [\![B]\!]_a \parallel [\![R]\!]_a \rangle \text{ where } G = a \boxtimes F$$

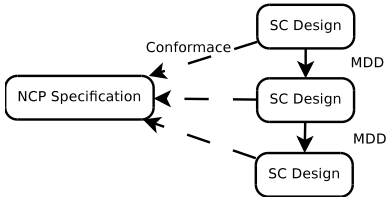
Table 4.12: Encoding the network N: $[\![N]\!]$

$$C ::= \emptyset \;\mid\; a\,[B]_F^R \;\mid\; C \parallel C \;\mid\; \langle \tau_{\circledcirc} \tau' \rangle @ a \;\mid\; (\nu n) C \;\mid\; *$$

Table 4.13: Context $\mathcal{C}$ syntax

The proof of the theorem is straightforward and can be done by induction over the context structure. If the context is a name restriction ($(\nu n)C$), we can directly exploit the Theorems 4. If the context is a parallel composition ($C_1 \parallel C_2$) we can use the Theorem 5. In fact, the subjects of the encodings ($[\![C_1]\!]$ and $[\![C_2]\!]$) must be disjoint, because we assume context well formed.

Putting the contents of this section together, we have a definition of satisfaction of a policy: let N be a *SC* network, P be a *NCP* policy and let G denote a topic-driven topology. We say that N implements the choreography (G,P) provided that $[\![N]\!] \approx (G,P)$. This is a semantic notion of satisfaction, since we define it up-to weak bisimulation, and can be mechanically checked in the finite-state case exploiting bisimulation-checking techniques such as those of [73]. This notion of satisfaction assists the development of systems in a Model Driven Development fashion. The designer can define successive *SC* models that implement the system, each of them is obtained refining the previous one to add more details. Moreover, the conformance of each model with respect to a *NCP* specification can be formally verified.



$$\left\langle (\emptyset, \{(b, \tau', \vec{c})\}) \; ; \; \tau\left(\tau'\right) @a.\iota.\overline{\tau_1}\; \tau' @b \right\rangle$$

Figure 4.2: *NCP* specification

Moreover, the presented characterization of the *NCP* bisimulation can be used to formally describe how designs must be updated to reflect changes to the specifications. For example, the Theorem 4 states that if a component or a topic is restricted in the specification, the implementation must ensure that external agents cannot deliver signals targeted to the restricted component or having the restricted topic. These results provide a starting infrastructure for a refactoring tools of *SC* designs and *NCP* policies. In Section 4.4 we will describe such kind of tool tailored for long running transactions.

4.2.1 Example of verifying *SC* designs

As described above, checking if a *SC* system respect an *NCP* specification is performed by verifying the weak bismimilarity between the specification and the translation of the system. We illustrate this methodology by verifying if some *SC* designs satisfy the *NCP* specification described in Section 4.1.2.

This *NCP* state specifies a system composed by two components know globally (a and b). The component a must be able to consume any envelope targeted to it and having the schema τ, regardless the session of the envelope. Moreover, after some internal computation of the system (ι.), the component b must raise an event of kind τ_1 having the same session of the one received by a. The topic names τ and τ_1 are free, representing that this kinds of event are globally know. Notice that the specification does not imposes any restriction regarding the mechanism used by a to inform b about the received session. Finally, the policy specifies that the flow of the component b does not change for all topic globally know.

$$a\,[0]^{\tau\ \lambda\ \tau' \gg \mathtt{out}\langle \tau \copyright \tau' \rangle}_{\tau \rightsquigarrow \{b\}} \parallel b\,[0]^{\tau\ \lambda\ \tau' \gg \mathtt{out}\langle \tau_1 \copyright \tau' \rangle}_{\tau_1 \rightsquigarrow \vec{c}}$$

(a) *SC* model

$$\langle (\emptyset, \{(a, \tau, \{b\}), (b, \tau_1, \vec{c})\})\ ;\ \tau(\tau')\,@a.\overline{\tau}\ \tau'@a \parallel \tau(\tau')\,@b.\overline{\tau_1}\ \tau'@b \rangle$$

(b) *NCP* encoding

Figure 4.3: Wrong implementation

$$(\nu \tau_2)\left(a\,[0]^{\tau\ \lambda\ \tau' \gg \mathtt{out}\langle \tau_2 \copyright \tau' \rangle}_{\tau_2 \rightsquigarrow \{b\}} \parallel b\,[0]^{\tau_2\ \lambda\ \tau' \gg \mathtt{out}\langle \tau_1 \copyright \tau' \rangle}_{\tau_1 \rightsquigarrow \vec{c}}\right)$$

(a) *SC* model

$$\langle (\emptyset, \{(b, \tau_1, \vec{c})\})\ ;\ (\nu\ \tau_2 : (a, b))\,(\tau(\tau')\,@a.\overline{\tau_2}\ \tau'@a \parallel \tau_2\,(\tau')\,@b.\overline{\tau_1}\ \tau'@b) \rangle$$

(b) *NCP* encoding

Figure 4.4: Correct implementation

The *SC* network in Figure 4.3a models a system that attempts to implement the specification. The network is composed only by the two component a and b. The component a is able to receive the τ envelopes by its lambda reaction, which simply forwards the envelope to the component b.

It is trivial to verify that the system does not correctly implement the specification. Intuitively, the lambda reaction of the component b reacts to any signal having the global topic τ, regardless the notifier component. If an external agent sends directly to b an envelope having this topic, b raises the event τ_1; this behavior is not prescribed by the specification. The formal verification is performed by translating the *SC* network, obtaining the *NCP* state in Figure 4.3b. Both the specification and the translation of the network can perform an `asinc` action $(\tau\ \tau'@b)$, storing concurrently with the policy the envelope $\langle \tau \copyright \tau' \rangle @b$. However the resulting policies do not continue in the same way. In fact, the specification cannot consume the pending envelope, enabling observable only the action $\langle \tau \copyright \tau' \rangle @b$ (corresponding to the signal). Instead, the implementation can consume the envelope via the reaction of b, activates the corresponding behavior and then deliver τ_1 envelopes to the components $\vec{c}$.

The issues of the previous implementation can be solved by using a private topic, shared between a and b. In the *SC* network of Figure 4.4a, the component a employees the private topic τ_2 to notify to b the reception of the session. Intu-

itively, the component b cannot be forced by an external agent to raise a τ_1 event without previously involve a. The *SC* network is translated to the *NCP* state in Figure 4.4b.

4.3 Encoding saga in *SC*

In Sections 2.2 and 2.4 we described how the BPMN notations and saga processes can be used to design transactional properties of business processes. Now, we provide an *SC* implementation of the saga processes (i.e. the subset of BPMN transactional designs). The starting point of our work is the implementation of simple BPMN processes described in Sections 3.1.2 and 3.2.2.

We assume that an execution of the business process is identified by a specific session, namely all events raised during one execution will be annotated with the corresponding session topic. The notification to a component of a *forward* event (having topic f) represents that all previous stages of the forward-flow have completed their execution. Similarly, when an error occurs, the notification of a *rollback* event (having topic r) represents that all next stages have completed their compensation.

Without loss of generality, we assume that each saga atomic activity has a unique name, ranged by $A, B, \ldots$. Let S be a saga we denote with $\mathcal{A}(S)$ the set of names of all atomic activities in S. We implement each saga step with an *SC* component. The mapping function $SC_{name} : \mathcal{A}_{ct} \rightarrow \mathcal{A}$ retrieves the *SC* component name $a \in \mathcal{A}$ that must implement the saga step containing the atomic activity $A \in \mathcal{A}_{ct}$.

The model transformation is provided by the function $[\![P]\!] = N, a, b, \vec{a}, \vec{b}$, which takes the saga process P and returns its implementation. Informally, the network N is a reference *SC* implementation of the saga process, the names a and b (called *sequential entry/exit points*) are the components that handle the start and the termination of the process, while the sets of component names $\vec{a}$ and $\vec{b}$ (called *parallel entry/exit points*) are the components that handle the start and termination of each independent concurrent branch in P. The meaning of sequential and parallel entry/exit points will be discussed later, when we will describe the implementation of sequential and concurrent saga processes, respectively.

The auxiliary functions $[\![A]\!]$ and $[\![A]\!]_c$ map an atomic activity and a compensation (A) to an *SC* behavior. We do not directly implement this function, since the action performed by A are not described in saga. We assume that after the termination of an atomic activity, the mapped behavior raises an *ok* event to notify its successful termination or an *ex* event to notify its failure. Moreover we assume that after a compensation has been completed, the corresponding behavior

$$[\![A \div B]\!] \triangleq (\nu ok, ex) SC_{name}(A)\,[0]_{ok \rightsquigarrow \{a\} | ex \rightsquigarrow \{a\}}\,, a, a, \{a\}, \{a\}$$

$$f\ \lambda\ s \gg \begin{array}{l} \texttt{rupd}\,(ok_{\circledcirc} s \gg \texttt{rupd}\,(r_{\circledcirc} s \gg [\![B]\!]_c)\mid \texttt{out}\langle f_{\circledcirc} s\rangle) \\ \mid \texttt{rupd}\,(ex_{\circledcirc} s \gg \texttt{out}\langle r_{\circledcirc} s\rangle) \\ \mid [\![A]\!] \end{array}$$

Where $a = SC_{name}(A)$.

Table 4.14: The transactional component

raises an r event. Formally, let $A \div B$ be a saga step and $SC_{name}(A)$ the component implementing it, if $A \mapsto \boxdot$ then

$$\left\langle (\emptyset, E)\ ;\ [\![[\![A]\!]]\!]_{SC_{name}(A)} \right\rangle \approx \left\langle (\emptyset, E)\ ;\ [\![\texttt{out}\langle ok_{\circledcirc} s\rangle]\!]_{SC_{name}(A)} \right\rangle$$

, otherwise

$$\left\langle (\emptyset, E)\ ;\ [\![[\![A]\!]]\!]_{SC_{name}(A)} \right\rangle \approx \left\langle (\emptyset, E)\ ;\ [\![\texttt{out}\langle ex_{\circledcirc} s\rangle]\!]_{SC_{name}(A)} \right\rangle$$

Since we assumed that a compensation always successes, the following statement must holds

$$\left\langle (\emptyset, E)\ ;\ [\![[\![B]\!]_c]\!]_{SC_{name}(A)} \right\rangle \approx \left\langle (\emptyset, E)\ ;\ [\![\texttt{out}\langle r_{\circledcirc} s\rangle]\!]_{SC_{name}(A)} \right\rangle$$

Notice that we do not specify the structure of E, meaning that the statements must hold for any flows of involved components.

4.3.1 The transactional component

We start implementing a single saga step, namely an atomic activity and its compensation. Let be $A \div B$ a saga step, its implementation is obtained by the *transactional component* $[\![A \div B]\!]$ defined in Table 4.14.

Initially, the component can react only to f events that notify the activation of the forward-flow of the task. Upon reception of such a signal, the component executes $[\![A]\!]$ and installs a check reaction; noteworthy, the lambda reaction binds the session s that uniquely identifies the envelopes of the current work-flow instance, to avoid that signals of another instances are intercepted.

The installed check reaction captures the signals $ok_{\circledcirc} s$ and $ex_{\circledcirc} s$ from $[\![A]\!]$. If ex is emitted, the corresponding reaction simply emits the rollback signal $r_{\circledcirc} s$ so that

the backward-flow is initiated. If *ok* is emitted, the forward flow continues because $f_{\circledcirc}s$ is emitted toward the components of the next stage. Notice that, if $[\![A]\!]$ emits *ok* the compensation is installed: in fact, the check reaction handling the *ok* events installs another check reaction that waits for rollback notifications $r_{\circledcirc}s$. If the backward-flow is activated, the compensation is executed and the rollback signal $r_{\circledcirc}s$ is propagated to the previous stages. According with the semantics of saga, rollback signals can be consumed only by components that successfully executed their main activity (and therefore installed their compensation). This implies that components execute their compensations only if their main activity ended correctly.

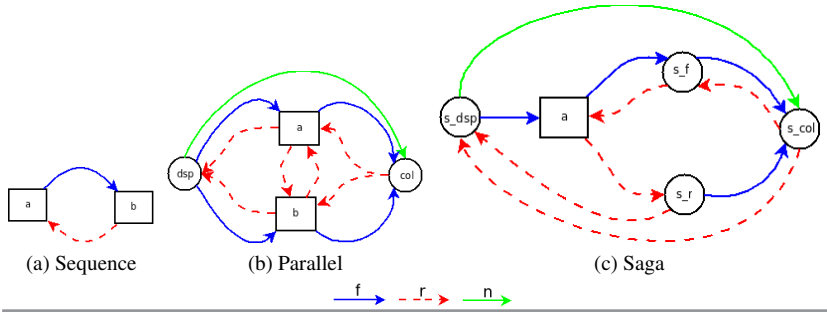
Initially, the transactional component has only two flows, both targeted to itself. These are used to notify to the component about the result of the execution of the main activity. Flows of the transactional component are rearranged later, when it is used to implement sequential and parallel compositions of saga processes. To update the flows of components we define the operator $N \oplus \{\vec{a} : F\}$, which adds to all components $\vec{a}$ contained in the network N the flow F. The operator is defined as follows:

$\emptyset \oplus \{\vec{a} : F\}$	$= \emptyset$
$\langle \tau_{\circledcirc} \tau' \rangle @ b \oplus \{\vec{a} : F\}$	$= \langle \tau_{\circledcirc} \tau' \rangle @ b$
$N_1 \parallel N_2 \oplus \{\vec{a} : F\}$	$= N_1 \oplus \{\vec{a} : F\} \parallel N_2 \oplus \{\vec{a} : F\}$
$b\,[B]^R_{F_1} \oplus \{\vec{a} : F\} =$	$b\,[B]^R_{F_1}$ *if* $b \notin \vec{a}$
$b\,[B]^R_{F_1} \oplus \{\vec{a} : F\} =$	$b\,[B]^R_{F_1 \mid F}$ *if* $b \in \vec{a}$

Table 4.15: The operator $N \oplus \{\vec{a} : F\}$

4.3.2 Sequential composition

Since the coding of a saga process always produces a sequential entry point and an exit point, the implementation of sequential composition simply requires to connect the exit point of the first process with the entry point of the second one. Figure 4.5a reports a graphical representation of the flows of a sequential composition. Formally, let $P_1;P_2$ be a saga sequential composition of two processes, $[\![P_1]\!] = N_1, a_1, b_1, \vec{a_1}, \vec{b_1}$ and $[\![P_2]\!] = N_2, a_2, b_2, \vec{a_2}, \vec{b_2}$ be the implementations of the two processes, we connect the forward-flow of the exit point b_1 with the entry point a_2 and the backward-flow reversely. The rule in Table 4.16 formally describes the methodology.



(a) Sequence (b) Parallel (c) Saga

Figure 4.5: Flows created by the coding of saga processes

$$\frac{\begin{array}{c}[\![P_1]\!] = N_1, a_1, b_1, \vec{a_1}, \vec{b_1} \\ [\![P_2]\!] = N_2, a_2, b_2, \vec{a_2}, \vec{b_2} \\ N_1 \parallel N_2 \; \mathit{Well\ formed}\end{array}}{[\![P_1;P_2]\!] = N_1 \oplus \{\{b_1\} : f \rightsquigarrow a_2\} \parallel N_2 \oplus \{\{a_2\} : r \rightsquigarrow b_1\}, a_1, b_2, \vec{a_1}, \vec{b_2}}$$

Table 4.16: Sequential composition

The constraint $N_1 \parallel N_2$ *Well formed* simply requires that the set of component names used by the two networks must be disjoint. The sequential composition acquires the entry points (both sequential and parallel) of the first process and the exit points of the second one $(a_1, b_2, \vec{a_1}, \vec{b_2})$.

4.3.3 Parallel composition

The encoding of a parallel composition of processes requires auxiliary components called *dispatcher* and *collector* to model the fork and join of the flow. Dispatchers are responsible to collect notifications of the forward flow (signals of topic f) and to dispatch them to the parallel entry points. Dispatchers also bounce rollback signals of topic r when the backward flow is executed. Analogously, collectors propagates forward and backward-flows by sending the signals of topic f or r as appropriate. Fig 4.5b yields a pictorial representation of the forward and backward flows; the dispatcher *dsp*, the collector *col* and the parallel compo-

nents a and b are coordinated using the f and r signals. Notice, that a and b have rollback flows connecting each other; in fact, the semantics of saga imposes that, when the main activity of a parallel component fails, the other components must be notified and start their compensations. The rule in Table 4.17 formally describes the coding of the parallel composition of two processes P_1 and P_2.

$$\frac{\begin{array}{l}[\![P_1]\!] = N_1, a_1, b_1, \vec{a_1}, \vec{b_1} \\ [\![P_2]\!] = N_2, a_2, b_2, \vec{a_2}, \vec{b_2} \\ N_1 \parallel N_2 \parallel Par_{dsp} \parallel Par_{col} \; Well\; Formed\end{array}}{\begin{array}{l} exi \\ \quad [\![P_1|P_2]\!] = \begin{array}{l} N_1 \oplus \{\{a_1\} : r \rightsquigarrow \{dsp\}\}\} \oplus \{\vec{a_1} : r \rightsquigarrow \vec{b_2}\} \oplus \{\{b_1\} : f \rightsquigarrow col\} \\ \parallel N_2 \oplus \{\{a_2\} : r \rightsquigarrow \{dsp\}\}\} \oplus \{\vec{a_2} : r \rightsquigarrow \vec{b_1}\} \oplus \{\{b_2\} : f \rightsquigarrow col\} \\ \parallel Par_{dsp} \parallel Par_{col} \end{array} \\ \quad , dsp, col, \vec{a_1} \cup \vec{a_2}, \vec{b_1} \cup \vec{b_2} \\ Where: \\ \quad Par_{dsp} = dsp\,[0]^{f \lambda\, s \gg \begin{array}{l} \texttt{rupd}\,(r \copyright s \gg \texttt{rupd}\,(r \copyright s \gg \texttt{out}\langle r \copyright s \rangle)) \\ \mid \texttt{out}\langle f \copyright s \rangle \\ \mid \texttt{out}\langle n \copyright s \rangle \end{array}}_{f \rightsquigarrow \{a_1, a_2\} \mid n \rightsquigarrow \{col\}} \\ \quad Par_{col} = col\,[0]^{n \lambda\, s \gg \texttt{rupd}\left(f \copyright s \gg \texttt{rupd}\left(f \copyright s \gg \begin{array}{l} \texttt{out}\langle f \copyright s \rangle \\ \mid \texttt{rupd}\,(r \copyright s \gg \texttt{out}\langle r \copyright s \rangle) \end{array}\right)\right)}_{r \rightsquigarrow \{b_1, b_2\}} \end{array}}$$

Table 4.17: Parallel composition

The last constraint simply requires that all component names of the resulting network, including the dispatcher and the collector, are different. The dispatcher and collector are the sequential entry point and exit point of the whole implementation, respectively. Moreover, the set of component that start ($\vec{a_1} \cup \vec{b_1}$) and end ($\vec{b_1} \cup \vec{b_2}$) concurrent branches are obtained by the union of the concurrent branches implementing the two saga processes. Notice, that the dispatcher communicates to the collector the work-flow session via a event tagged with the topic n. The session notification is mandatory to perform the synchronization of the forward-flow. This synchronization mechanism has been previously described in Section 3.2.2.

$$[\![P]\!] = N, a, b, \vec{a}, \vec{b}$$
$$N \parallel Saga_{dsp} \parallel Saga_f \parallel Saga_r \parallel Saga_{col} \textit{ Well Formed}$$

$$\begin{array}{ll}
[\![\{|P|\}]\!] = & (\nu n, c\vec{m}p) \\
& N \oplus \{\{b\} : f \rightsquigarrow \{s_f\}\} \oplus \{\{a\} : r \rightsquigarrow \{s_r\}\} \\
& \parallel Saga_{dsp} \parallel Saga_f \parallel Saga_r \parallel Saga_{col} \\
& , s_{dsp}, s_{col}, \{s_{dsp}\}, \{s_{col}\} \\
\textit{where}: & \\
Saga_{dsp} = & s_{dsp}\,[0]^{f\ \lambda\ s \gg \mathtt{out}\langle f \copyright s\rangle \,|\, \mathtt{out}\langle n \copyright s\rangle \,|\, \mathtt{rupd}(r \copyright s \gg \mathtt{rupd}(r \copyright s \gg \mathtt{out}\langle r \copyright s\rangle))}_{f \rightsquigarrow \{a\} | n \rightsquigarrow \{s_{col}\}} \\
Saga_f = & s_f\,[0]^{f\ \lambda\ s \gg \mathtt{out}\langle f \copyright s\rangle \,|\, \mathtt{rupd}(r \copyright s \gg \mathtt{out}\langle r \copyright s\rangle)}_{r \rightsquigarrow \{b\} | f \rightsquigarrow \{s_{col}\}} \\
Saga_r = & s_r\,[0]^{r\ \lambda\ s \gg \mathtt{out}\langle f \copyright s\rangle \,|\, \mathtt{out}\langle r \copyright s\rangle}_{r \rightsquigarrow \{s_{dsp}\} | f \rightsquigarrow \{s_{col}\}} \\
Saga_{col} = & s_{col}\,[0]^{n\ \lambda\ s \gg \mathtt{rupd}(f \copyright s \gg \mathtt{out}\langle f \copyright s\rangle \,|\, \mathtt{rupd}(r \copyright s \gg \mathtt{out}\langle r \copyright s\rangle))}_{r \rightsquigarrow \{s_f, s_{col}\}} \\
c\vec{m}p = & \{s_f, s_r\} \cup (fn(N) \cap \mathcal{A}) \setminus \{a \in \mathcal{A} \mid \exists A \in \mathcal{A}(P) \textit{ and } SC_{name}(A) = a\}
\end{array}$$

Table 4.18: Nested transaction

4.3.4 Transactions

Finally, we present the implementation of a saga $\{|P|\}$. The main goal of the transformation function is to hide the result of the execution of the contained process P. Namely, independently by the termination of P, external agents are always notified of a successful execution of the whole saga. With this purpose, we use four components; $Saga_{dsp}$ implements the entry point of the whole saga, $Saga_f$ and $Saga_r$ verify the result of the internal process P, finally $Saga_{col}$ represents the exit point of the whole transaction.

The restriction of the names $c\vec{m}p$ permits to hide to external agents all details of the implementation of the internal process P. Namely, the transformation function creates several components to synchronize forward-flow (the parallel collectors) and backward-flow (the parallel dispatchers) of tasks involved by the process P. The set of names created for this purpose is represents by $c\vec{m}p$. In fact, the set of all component names used to implement the process P is $fn(N) \cap \mathcal{A}$, while the set of component names the are created to implement the tasks (the transactional components) is $\{a \mid \exists A \in \mathcal{A}(P) \textit{ and } SC_{name}(A) = a\}$.

The forward-flow is initiated by the entry point of the transaction (s_{dsp}), which waits the notification of a f event. The entry point forwards the notification to the entry point (a) of the process to isolate (P). The two components s_f and s_r handle the termination status of the process P. If the process P successes, its exit point b notifies the termination to s_f, using an f-event, otherwise, the entry point a notifies the failure to s_r, using a r-event. Notice that, independently by the termination status of the process P, the component s_{col} is notified about an f-event. This permits to hide failures of the contained process to any external agent. The forward-flow is completed by the rising of the event to the collector s_{col}, which represents the end-point of the transaction.

The backward-flow is initiated by the exit point of the transition (s_{col}), which waits the notification of a r-event. The exit point forward the request notification to both components s_f and s_{dsp}. The component s_f is responsible to forward the compensation demand to the exit point b. Thus the internal implementation can perform its compensations. Notice that the component s_{dsp} implements a synchronization of the backward flow. In fact it is notified twice about the r-events.

- If the internal implementation fails, s_{dsp} is immediately notified by s_r. However, before propagating the backward-flow it waits for a second notification from s_{col}. In fact, also if the internal implementation fails, the forward-flow must be propagated and the backward-flow delayed until the occurrence of a failure of external activities.

- If the internal implementation successes, s_{dsp} is notified by s_r only after that the internal compensations have been completed. However, it is also notified by s_{col}, thus informing the dispatcher that the rollback request is coming from outside the saga.

4.4 Refactoring LRT

A few aspects of SOC systems evident in *SC*, such as, the component distribution on the network, are not explicitly modeled in saga. In fact, either saga processes are not concerned with such aspects or, more pragmatically, they can more suitably considered at later stages of the development. For example,saga processes sketches how the overall transaction among transactional tasks should proceed without making any further assumption on which services implement such components (or where they are located). saga processes (i) neglect distribution aspects of the transactional activities, (ii) does not specify if activities are atomic or consisting of hidden sub-activities, (iii) delegate activities or compensations

(e.g., according to patterns like farm). Arguably, refinement does not have to alter the intention of the designer, namely, refinement must preserve the intended semantics.

Our aim is to study the formal properties of some refactoring rules applied to transactional behaviors for which long running transactions (LRT) have been proposed. Our refactoring rules are proved sound by showing that they preserve (weak) bisimulation. The proof relies on a bisimulation preserving mapping from *SC* to its choreographic view expressed in the *NCP*. More precisely, we show that the *NCP* image of an *SC* system is weak bisimilar to a refactoring obtained by applying any of our rules. Albeit the proof could have been given directly at the *SC* level, we prefer to deviate through *NCP* for simplicity and, more importantly, because *NCP* provides a choreographic view of the *SC* system that is closer to the original BPMN design. Hence, *NCP* images of *SC* processes can help to change the BPMN design if problems spotted at the *SC* level require to modify the original BPMN design.

We argue that the translation of saga transactions into *SC* networks provides the suitable level of abstraction to which further refactoring steps can be applied. For example, deployment of distributed components or rearrangement of points of control can be automatically transformed at the *SC* level respecting the original semantics of automatically translated designs.

In the following, we presents some useful refactoring rules by proving, through bisimulation, that they produce new networks without interfering on their semantics.

4.4.1 Refactoring transactional components

We introduce now our first refactoring rule that can be applied to any *SC* component obtained by translating a saga step as shown in Section 4.3.1.

As said, both the main activity and the compensation of a saga step are embedded into a single *SC* component that has then to manage *ok* and *ex* signals in order to propagate forward or backward flows. However can be useful to assign the compensation task to a different agent.

For example, it might be necessary to execute the compensation `CompA` in Figure 2.3a on a different host than the one of `A`, because it involve a remote service. This cannot be specified in saga. Instead, when the business process is mapped in an *SC* network, it is possible to allocate `CompA` on a different host by taking advantage of the JSCL (see Section 6.2) implementation, which permits to distribute the deployment orthogonally to how the network is generated.

The delegation of the compensation of a transactional component a to a component b produces the *SC* in Figure 4.6.

$$TC = (\nu ok, ex)\left(\begin{array}{l} f\ \lambda\ s \gg \left(\begin{array}{l} \texttt{rupd}\left(ok_{\circledcirc} s \gg \begin{array}{l}\texttt{rupd}(r_{\circledcirc} s \gg [\![CompA]\!]) \\ \mid \texttt{out}\langle f_{\circledcirc} s\rangle\end{array}\right) \\ \mid \texttt{rupd}\left(ex_{\circledcirc} s \gg \texttt{out}\langle r_{\circledcirc} s\rangle\right) \\ \mid [\![A]\!] \end{array}\right) \\ a\,[0]_{\{ok \rightsquigarrow a, ex \rightsquigarrow a, f \rightsquigarrow \vec{c_1}, r \rightsquigarrow \vec{c_2}\}} \end{array}\right)$$

$$DelegatedTC = (\nu b, ok, ex)\left(\begin{array}{l} f\ \lambda\ s \gg \left(\begin{array}{l} \texttt{rupd}\left(ok_{\circledcirc} s \gg \begin{array}{l}\texttt{rupd}(r_{\circledcirc} s \gg \texttt{out}\langle r_{\circledcirc} s\rangle) \\ \mid \texttt{out}\langle f_{\circledcirc} s\rangle\end{array}\right) \\ \mid [\![A]\!] \end{array}\right) \\ a\,[0]_{\{ok \rightsquigarrow a, ex \rightsquigarrow b, f \rightsquigarrow \vec{c_1}, r \rightsquigarrow b\}} \\ \parallel b\,[0]^{R_b}_{\{r \rightsquigarrow \vec{c_2}\}} \end{array}\right)$$

where $R_b = ex\ \lambda\ s \gg \texttt{out}\langle r_{\circledcirc} s\rangle \mid\ r\,\lambda\, s \gg [\![CompA]\!]$

Figure 4.6: Delegate compensation to a new component

The refactoring rule uses a restricted component b (where $b \in A$ is fresh) responsible to perform the compensation and manage the backward flow. For this reason, the compensation of a is moved on b towards which a directs r and ex signals as specified in the refactored set of flows of a in Figure 4.6. The refactored a component needs only to check the successful termination of its main activity. In fact, the check reaction of a in Figure 4.6 propagates the forward flow and activates a listener for the rollback signals possibly raised by subsequent transactional components. Notice that a delegates the execution of the compensation $[\![CompA]\!]$ to the new component b. This permits to the new component to be informed if something goes wrong either during the execution of the main activity (ex signals) or, after a successful execution of $[\![A]\!]$ when r signals may be delivered by other components.

The initial reactions of b are given by R_b; namely, b waits the notification of an exception from $[\![A]\!]$ or a rollback signal from subsequent components. In the former case, b simply activates the backward flow (as per the reaction migrated from a) while, in the latter case, b executes $[\![CompA]\!]$ that, as said, upon termination starts the backward flow.

Theorem 8 proves that this refactoring rule is safe as it preserves weak bisimulation.

Theorem 8 *Let TC and DelegetedTC be as in Figure 4.6 then*

$$[\![TC]\!] \approx [\![DelegatedTC]\!]$$

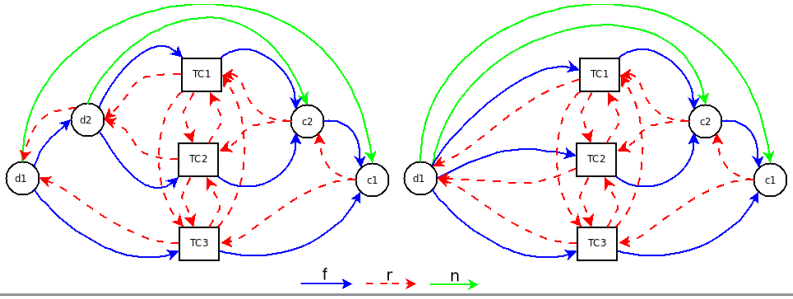


Figure 4.7: Parallel composition and its refactoring

The theorem can be proved verifying that each transition of TC is (weakly) matched by a transition of *DelegatedTC*, and viceversa. The proof of the theorem is reported in Appendix C.1.

4.4.2 Refactoring parallel composition

Figure 4.7a depicts the flows and components required to implement the parallel composition of three saga steps.

Two distinct dispatchers (d_1 and d_2) are involved in the coordination. Dispatcher d_2 is responsible to forward the received requests to components TC_1 and TC_2 and results externally the entry point of their parallel composition. As result, the dispatcher d_1 is connected to TC_3 and to d_2 acting as entry point for the whole parallel block. Similar considerations can be made for the exit points c_1 and c_2. The notification of events to these dispatchers are not relevant to the semantic of the implementing network (more precisely these are hidden notification, since the dispatcher components should be hidden out of the scope of the network itself). The generation of two different dispatchers can provide a mechanism to optimize the communications among components. For example, if the component d_2, TC_1 and TC_2 reside on the same host, the generated dispatcher permits to reduce the inter-host communications for the forward and backward flow, since it receives only one inter-host envelope and then generates two intra-host envelopes to the components TC_1 and TC_2. If the distribution of the components cannot take advantage from this feature (e.g. TC_1, TC_2 and TC_3 have to reside on different hosts) the two dispatcher should be fused. The transformation that we explain hereafter is twofold; (i) it can merge two parallel dispatchers

into one, simplifying the *SC* design, (ii) it can split a parallel dispatcher, refining the communication hierarchy among hosts. In the following we prove the correctness of this refactoring only for parallel dispatcher. Noteworthy, the same proof strategy can be applied to provide a similar refactoring mechanism for the collectors.

Let $Sync_n(\tau_{\circledcirc}s)(B)$ be the behavior that synchronizes n reception of signals $\tau_{\circledcirc}s$:

$$Sync_0(\tau_{\circledcirc}s)(B) = B \quad \text{and} \quad Sync_n(\tau_{\circledcirc}s)(B) = \texttt{rupd}\,(\tau_{\circledcirc}s \gg Sync_{n-1}(\tau_{\circledcirc}s)(B))$$

Any *SC* network with a dispatcher d_1 triggering a dispatcher d_2 can be written as

$$\begin{array}{l} N_{d_1,d_2} = (\nu d_1)(\nu d_2)(N \parallel D) \\ \textit{where}: \\ \qquad D = d_2\,[0]_{f \rightsquigarrow \vec{a_2} | r \rightsquigarrow \{d_1\} | n \rightsquigarrow \vec{c_2}}^{f\ \lambda\ s \gg \begin{array}{r} Sync_{k_2}(r_{\circledcirc}s)(\texttt{out}\langle r_{\circledcirc}s\rangle) \\ \mid \texttt{out}\langle f_{\circledcirc}s\rangle \\ \mid \texttt{out}\langle n_{\circledcirc}s\rangle \end{array}} \parallel d_1\,[0]_{f \rightsquigarrow \vec{a_1} \cup \{d_2\} | r \rightsquigarrow \vec{b} | n \rightsquigarrow \vec{c_1}}^{f\ \lambda\ s \gg \begin{array}{r} Sync_{k_1}(r_{\circledcirc}s)(\texttt{out}\langle r_{\circledcirc}s\rangle) \\ \mid \texttt{out}\langle f_{\circledcirc}s\rangle \\ \mid \texttt{out}\langle n_{\circledcirc}s\rangle \end{array}} \end{array} \tag{4.1}$$

We can merge the two parallel dispatcher, migrating the flows of the component d_2 to the component d_1 and adding to it the synchronizations of d_2:

$$\begin{array}{l} N'_{d_1,d_2} = (\nu d_1)\,(\{d_1/d_2\}N \parallel D') \\ \textit{where}\ D' = d_1\,[0]_{f \rightsquigarrow (\vec{a_1} \cup \vec{a_2}) | r \rightsquigarrow, \vec{b} | n \rightsquigarrow (\vec{c_1} \cup \vec{c_2})}^{f\ \lambda\ s \gg Sync_{k_1+k_2-1}(r \copyright s)(\texttt{out}\langle r \copyright s\rangle) \mid \texttt{out}\langle f \copyright s\rangle \mid \texttt{out}\langle n \copyright s\rangle} \end{array} \tag{4.2}$$

We start characterizing the behavior of the two systems N_{d_1,d_2} and N'_{d_1,d_2}. The starting system N_{d_1,d_2} consists of dispatchers d_1 and d_2, namely D, and of all other components, namely N. Our refactoring changes only the flows of N by migrating all flows towards d_2 onto d_1. The resulting network $\{d_1/d_2\}N$ performs the same actions as the original one, but for the notifications to d_2, that are delivered to d_1. In this case, we say that for the policy corresponding to N the name d_2 can be *fused* with d_1.

The network D should be refactored according to the changes applied to the network N. Since the N refactoring will notify the same events respect to the starting network, the D refactoring should be able to consume the same events of the starting dispatcher network. However all events that in D where consumed by the component d_1 or d_2 will be consumed in the refactoring only by the component a. Moreover, the refactored network must deliver the same envelopes to the network N. If the refactoring is correct, we say that it merges of the behavior of the two dispatcher.

Theorem 9 *Let N_{d_1,d_2} be as in 4.1 and N'_{d_1,d_2} as in (4.2) then* $[\![N_{d_1,d_2}]\!] \approx \left[\!\left[N'_{d_1,d_2}\right]\!\right]$

The theorem can be proved verifying that each transition of $[\![N_{d_1,d_2}]\!]$ is (weakly) matched by a transition of $\left[\left[N'_{d_1,d_2}\right]\right]$, and viceversa. The sketch of the proof of the theorem is reported in Appendix C.2.

4.5 Related Works

In this Chapter we presented our reasoning techniques for the *SC* framework. The first contribution of this Chapter is the definition of the choreography model, called Network Coordination Policies calculus (*NCP*). *NCP*exploits the interaction pattern of *SC*by a global point of view. Since *NCP* is intended as a specification formalism, we formalize its abstract semantics to provide a verification strategy. The problem of checking if two specification are *compatible* is solved by verifying their weak bisimilarity.

To fill the gap between the choreography model and the system design, we provided a semantic transformation that derive the *NCP* state starting from the *SC* design. These results provide a formal definition of satisfaction of a specification in term of weak bisimilarity checking.

To highlight our theoretical approach, we introduced a formal transformation that provide a reference *SC* design starting from a saga model. Then we presented the formal properties of some refactoring rules applied to *SC* models that implement saga transactional behaviors. We verified that the refactoring transformation preserve weak bisimilarity of the implementing *SC* models.

Several other models have been proposed to model choreography (e.g. the Global Calculus [19] and the work presented in [28]). A distinguished feature of *NCP* is that it can handle multicast interaction in a natural way. *NCP* is centered around the notion of *network topology*, which permits to represents the component subscriptions by a global point of view. The *NCP* abstract semantics handles a semantic context (the topology) that affects and can be update by processes (*NCP* policies). Moreover, the interplay between the topologies and the name restrictions allows *NCP* to hide non only names, but also a whole part of the component connections.

A key difference between *NCP* and the asynchronous π-calculus (presented in Section 2.3.5) is the awareness of topic topologies in the semantics. However, it would be too restrictive to require that only policies with the same topology can be bisimilar. For example, the empty network is bisimilar to itself under any topology.

Chapter 5

The *SC* practice

In this Chapter we illustrate the usage of our framework through a case study [32] borrowed by the SENSORIA Project [33]. We address the problem of developing a service oriented application for an automotive system that involves several services, which should be provided by several providers.

We assume a car equipped with a diagnostic system that continuously reports on the status of the vehicle. When the car experiences some major failure (e.g. engine overheating, exhausted battery, flat tires) the on-board emergency service is invoked to

1. locate a garage,
2. locate a tow truck and
3. locate a rental car service

so that the car is towed to the garage and repaired meanwhile the customer may continue his/her travel. Moreover, the system informs the customer company about its delay, so that the company information system can handle the issue.

The inter-dependencies among the services are summarized as follows:

- the first step is to charge the credit card with a security amount;
- before looking for a tow truck, a garage must be found as it poses additional constraints to the candidate tow trucks;
- if finding a tow truck fails, the garage appointment must be revoked;
- if renting a car succeeds and finding a tow truck appointment fails, the car rental must be redirected to the broken down car's actual location;

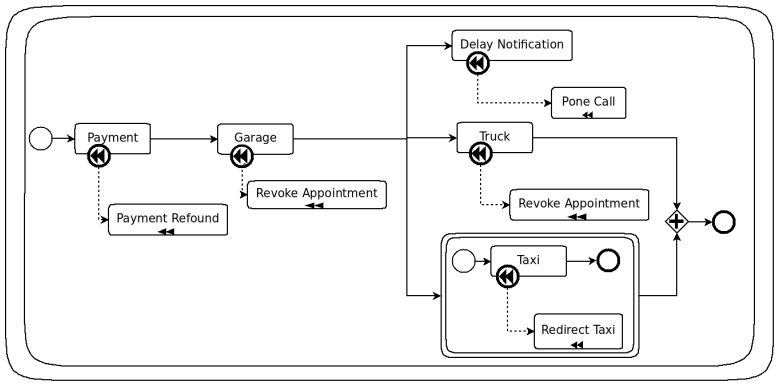


Figure 5.1: The BPMN case study model

- if the car rental fails, it should not affect the other services, since the customer can be transported by the tow truck to the garage.
- if the truck reservation fails a phone call have to be instantiated between the customer and his company, so that the customer can requires an additional support from its company.
- the taxi is ordered to reach the garage location.
- if the truck reservation fails and a taxi has been reserved, it must be redirected to the car location.

5.1 Model transactional properties

We specify the automotive system by exploiting the BPMN design of Figure 5.1. BPMN permits to design the work-flow of a distributed system and its transactional properties by a global point of view. The software designer can abstract from the distribution of the processes, the communication mechanisms and the technologies that are used to implement the processes. The resulting abstract specification can also be reused if further architectural design changes.

The BPMN design of Figure 5.1 is a semi-formal specification of the system that describes the transactional requirements and temporal inter-dependencies among tasks. In Figure 5.2 we exploit the saga calculus to formally describe this properties of the system. We will use the BPMN task names (e.g. *Payment*, $PaymentRefound$, *Taxi*) to identify the saga atomic activities and compensations.

$$
\begin{array}{ll}
P_{Payment} = & Payment \div PaymentRefound \\
P_{Garage} = & Garage \div RevokeAppointment \\
P_{DelayNotification} = & DelayNotification \div PhoneCall \\
P_{Truck} = & Truck \div CancelTruck \\
P_{Taxi} = & Taxi \div RedirectTaxi \\
S_{Taxi} = & \{|P_{Taxi}|\} \\
S = & \left\{\left|P_{Payment}; P_{Garage}; \left(P_{DelayNotification} | P_{Truck} | S_{Taxi}\right)\right|\right\}
\end{array}
$$

Figure 5.2: The saga formalization of the case study

We first formalize each BPMN compensable task by using a saga step, which is composed by an atomic activity and its compensation. We use the name P_A to identify the step whose main activity is A.

To represent the constraint that a failure of a taxi reservation has no effect on the system, we "box" the corresponding step P_{Taxi} into the saga $S_{Taxi} = \{|P_{Taxi}|\}$. Finally the whole system is formalized by the saga S, which contains the elements defined in the previous stages.

5.2 The reference *SC* implementation

We use the encoding of saga into *SC* described in Section 4.3 to provide a reference implementation of the saga S. As usual we start describing the implementation of the main building blocks of the system. We recall that the reference implementation requires the definition of the mapping function $SC_{name} : \mathcal{A} \to \mathcal{A}$, which retrieves the *SC* component name $a \in \mathcal{A}$ that must implement the saga step containing the atomic activity $A \in \mathcal{A}$. We assume that the function satisfies the following constraints:

$$SC_{name}(Payment) = pa \quad SC_{name}(Garage) = ga \quad SC_{name}(Truck) = tr$$

$$SC_{name}(DelayNotification) = dn \quad SC_{name}(Taxi) = ta$$

We use $N_{SC_{name}(A)}$ to identify the *SC* network implementing the saga step whose main activity is A. For example N_{ga} represents the network implementing the saga step $Garage \div RevokeAppointment$. Let $A \div B$ be a saga step, its *SC* reference implementation is obtained by the transformation $[\![A \div B]\!]$ described in

$$[\![Garage \div RevokeAppointment]\!] = N_{ga}, ga, ga, \{ga\}, \{ga\}$$

$$f\ \lambda\ s \gg \begin{array}{l} \texttt{rupd}\left(ok_{\circledcirc} s \gg \begin{array}{l}\texttt{rupd}(r_{\circledcirc} s \gg [\![RevokeAppointment]\!]) \\ \mid \texttt{out}\langle f_{\circledcirc} s\rangle\end{array}\right) \\ \mid \texttt{rupd}\left(ex_{\circledcirc} s \gg \texttt{out}\langle r_{\circledcirc} s\rangle\right) \\ \mid [\![Garage]\!] \end{array}$$

$$where\ N_{ga} = (\nu ok, ex)\, ga\,[0]_{ok \rightsquigarrow \{ga\} \mid ok \rightsquigarrow \{ga\}}$$

Figure 5.3: The *SC* implementation of the step P_{Garage}

$$[\![S_{ta}]\!] = N_{S_{ta}}, s_{ta_{dsp}}, s_{ta_{col}}, \{s_{ta_{dsp}}\}, \{s_{ta_{col}}\}$$

$$where\ N_{S_{ta}} = (\nu n, s_{ta_f}, s_{ta_r}) \left(\begin{array}{l} N_{ta} \oplus \{\{ta\} : f \rightsquigarrow \{s_{ta_f}\}\} \oplus \{\{ta\} : r \rightsquigarrow \{s_{ta_r}\}\} \\ \parallel Saga_{ta_{dsp}} \parallel Saga_{ta_f} \parallel Saga_{ta_r} \parallel Saga_{ta_{col}} \end{array} \right)$$

Figure 5.4: The *SC* implementation of the saga S_{Taxi}

Section 4.3.1. Each step is implemented through a dedicated transactional component, which is responsible to implement both the main activity and its compensation. Figure 5.3 illustrates the implementation of P_{Garage}, which models the booking of a garage and its compensation: revoking the appointment. We assume that the implementation of the main activity ($[\![Garage]\!]$) raises the *ok* event to notify its successful termination or the *ex* event to notify its failure. Moreover, we assume that the compensation ($[\![RevokeAppointment]\!]$) always successes and terminates its execution by rising the *r* event. Initially, each transactional component has only two flows, both targeted to itself, whose are used to retrieve the termination status of the main activity. The flows of the component will be updated later, when the transactional components are composed to implement sequential and parallel sagas. Since the produced network contains only the component *ga*, it represent both entry and exit points of the mapping.

We continue implementing the sub-transaction S_{Taxi}, which isolates the termination status of the taxi activity. The transformation function produces the reference implementation $N_{S_{ta}}$ exploiting the transactional component N_{Ta} (see Figure 5.4). The networks $Saga_{ta_{dsp}}$, $Saga_{ta_f}$, $Saga_{ta_r}$ and $Saga_{ta_{col}}$ contains the four auxiliary component described in Section 4.3.4, which names are assumed to be $s_{ta_{dsp}}$, s_{ta_f}, s_{ta_r} and $s_{ta_{col}}$, respectively. Notice that the component names

$$\left[\left[(P_{DelayNotification}|P_{Truck})|S_{taxi}\right]\right] = N_{par}, dsp_2, col_2, \{dn, tr, s_{ta_{dsp}}\}, \{dn, tr, s_{ta_{col}}\}$$

$$\text{where } N_{par} = \begin{array}{l} N_{dn} \oplus \{\{dn\} : r \rightsquigarrow \{dsp_1, tr, s_{ta_{dsp}}\}\} \oplus \{\{dn\} : f \rightsquigarrow \{col_1\}\} \\ N_{tr} \oplus \{\{tr\} : r \rightsquigarrow \{dsp_1, dn, s_{ta_{dsp}}\}\} \oplus \{\{dn\} : f \rightsquigarrow \{col_1\}\} \\ N_{S_{ta}} \oplus \{\{s_{ta_{dsp}}\} : r \rightsquigarrow \{dsp_2, dn, tr\}\} \oplus \{\{s_{ta_{col}}\} : f \rightsquigarrow \{col_2\}\} \\ \| \, N_{dsp_1} \, \| \, N_{dsp_2} \, \| \, N_{col_1} \, \| \, N_{col_2} \end{array}$$

$$N_{dsp_1} = dsp_1\,[0]^{f \,\lambda\, s \gg \; Sync_2(r \circledcirc s)(\mathtt{out}\langle r \circledcirc s\rangle) \;|\; \mathtt{out}\langle f \circledcirc s\rangle \;|\; \mathtt{out}\langle n \circledcirc s\rangle}_{f \rightsquigarrow \{dn,tr\} | r \rightsquigarrow \{dsp_2\} | n \rightsquigarrow \{col_1\}}$$

$$N_{col_1} = col_1\,[0]^{n \,\lambda\, s \gg \; Sync_2(f \circledcirc s)\left(\begin{smallmatrix} \mathtt{out}\langle f \circledcirc s\rangle \\ \mathtt{rupd}(r \circledcirc s \gg \mathtt{out}\langle r \circledcirc s\rangle)\end{smallmatrix}\right)}_{f \rightsquigarrow \{col_2\} | r \rightsquigarrow \{dn,tr\}}$$

$$N_{dsp_2} = dsp_2\,[0]^{f \,\lambda\, s \gg \; Sync_2(r \circledcirc s)(\mathtt{out}\langle r \circledcirc s\rangle) \;|\; \mathtt{out}\langle f \circledcirc s\rangle \;|\; \mathtt{out}\langle n \circledcirc s\rangle}_{f \rightsquigarrow \{dsp_1, s_{ta_{dsp}}\} | n \rightsquigarrow \{col_2\}}$$

$$N_{col_2} = col_2\,[0]^{n \,\lambda\, s \gg \; Sync_2(f \circledcirc s)\left(\begin{smallmatrix} \mathtt{out}\langle f \circledcirc s\rangle \\ \mathtt{rupd}(r \circledcirc s \gg \mathtt{out}\langle r \circledcirc s\rangle)\end{smallmatrix}\right)}_{r \rightsquigarrow \{col_1, s_{ta_{col}}\}}$$

Figure 5.5: The *SC* implementation of the process $(P_{DelayNotification}|P_{Truck})|S_{taxi}$

s_{ta_f} and s_{ta_r} are under the scope of a restriction. The intuition behind this is that any external component cannot interact with s_{ta_f} and s_{ta_r}. We also need to update the flows of the transactional component ta, to inform the *auxiliary* components about its termination. The graphical representation of the flows of the whole network is depicted in Figure 5.8a.

The above saga and the two network N_{dn} and N_{tr} must be composed to implement the parallel composition $P_{DelayNotification}|P_{Truck}|S_{taxi}$. We exploit the distributive property of the parallel operator to transform the starting process into $(P_{DelayNotification}|P_{Truck})|S_{taxi}$. We create four auxiliary components; a parallel dispatcher and a parallel collector for each saga parallel operator in the process. We assume that the names dsp_1, col_1, dsp_2 and dsp_2 identify the dispatcher and collector for the first operator $(P_{DelayNotification}|P_{Truck})$ and the dispatcher and collector of the second one $((\ldots)|S_{taxi})$. Let N_i be the *SC* network containing the auxiliary component named i, the encoding of the whole process is reported in

$$N_{seq}, pa, col_2, \{pa\}, \{col_2\}$$

$$\text{where } N_{seq} = \begin{array}{l} N_{pa} \oplus \{\{ga\} : f \rightsquigarrow \{dsp_1\}\} \\ N_{ga} \oplus \{\{ga\} : f \rightsquigarrow \{dsp_2\}\} \oplus \{\{ga\} : r \rightsquigarrow \{pa\}\} \\ N_{par} \oplus \{\{dsp_2\} : r \rightsquigarrow \{ga\}\} \end{array}$$

Figure 5.6: The *SC* implementation of the sequential composition

$$[\![S]\!] = \quad N_S, s_{dsp}, s_{col}, \{s_{dsp}\}, \{s_{col}\}$$

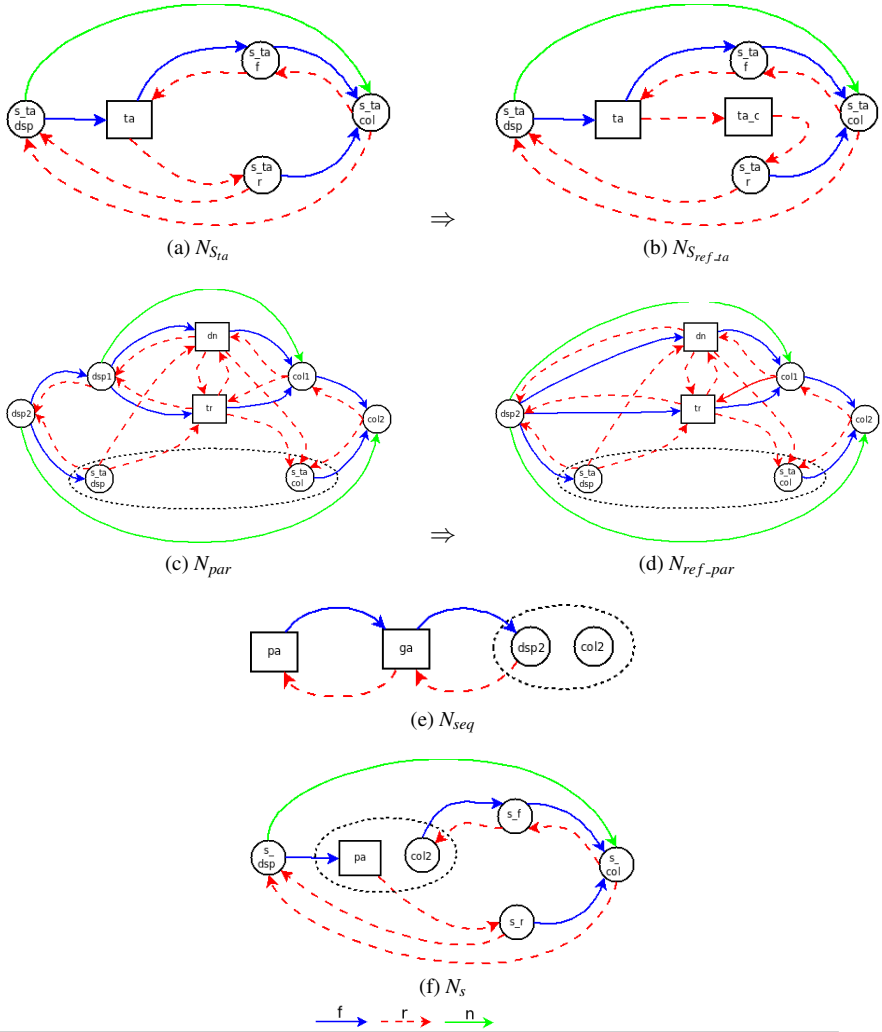
$$\text{where } N_S = \quad (\nu n, dsp_1, dsp_2, col_1, col_2, s_{ta_{dsp}}, s_{ta_{col}}, s_f, s_r) \left(\begin{array}{l} N_{seq} \oplus \{\{col_2\} : f \rightsquigarrow \{s_f\}\} \oplus \{\{pa\} : r \rightsquigarrow \{s_r\}\} \\ \parallel Saga_{dsp} \parallel Saga_f \parallel Saga_r \parallel Saga_{col} \end{array} \right)$$

Figure 5.7: The *SC* implementation of the saga S

Figure 5.5. The flows of the resulting network are depicted in Figure 5.8c. Notice that the three networks implementing the three concurrent saga processes are interconnected for the backward-flow, to start the compensation of concurrent saga branches if at least one of them fails.

The sequential process $P_{Payment}; P_{Garage}; \left(P_{DelayNotification} | P_{Truck} | S_{Taxi}\right)$, is implemented by composing N_{pa}, N_{ga} and the network N_{par}. The encoding simply connects the forward-flow and backward-flow of sequential entry/exit points as depicted in Figure 5.8e. The resulting network N_{seq} is described in Figure 5.6.

Finally, we can implement the saga S by boxing the network N_{seq} with the same technique used for the inner transaction (see Figure 5.7). The networks $Saga_{dsp}$, $Saga_f$, $Saga_r$ and $Saga_{col}$ contain the four auxiliary component described in Section 4.3.4, which names are assumed to be s_{dsp}, s_f, s_r and s_{col}. Notice that the component names $\{dsp_1, dsp_2, col_1, col_2, s_{ta_{dsp}}, s_{ta_{col}}, s_f, s_r\}$ are restricted, representing that this components are used to implement the internal synchronizations and that any external agent cannot interact with them. Finally, we update the flows of the network to allow *auxiliary* components to keep track the forward and backward flows. The graphical representation of the connections is depicted in Figure 5.8f.



(a) $N_{S_{ta}}$ ⇒ (b) $N_{S_{ref\_ta}}$

(c) N_{par} ⇒ (d) N_{ref_par}

(e) N_{seq}

(f) N_s

Figure 5.8: Flows of the reference implementation and of its refactoring

$$N_{ta_{ref}} = (\nu\, ta_c, ok, ex)\left(\begin{array}{l} f\ \lambda\ s\gg\left(\begin{array}{l}\texttt{rupd}\left(ok_{\copyright}s \gg \begin{array}{l}\texttt{rupd}(r_{\copyright}s \gg \texttt{out}\langle r_{\copyright}s\rangle) \\ \mid \texttt{out}\langle f_{\copyright}s\rangle\end{array}\right) \\ \mid [\![Taxi]\!]\end{array}\right) \\ ta\,[0]_{\{ok\rightsquigarrow\{ta\},ex\rightsquigarrow\{ta_c\},f\rightsquigarrow\{s_{ta_f}\},r\rightsquigarrow\{ta_c\}} \\ \\ \parallel ta_c\,[0]^{ex\ \lambda\ s\gg\texttt{out}\langle r_{\copyright}s\rangle \mid\ r\,\lambda\,s\gg[\![RedirextTaxi]\!]}_{r\rightsquigarrow\{s_{tar}\}} \end{array}\right)$$

(a) Delegate the compensation of taxi

$$N_{dsp2_{ref}} = dsp_2\,[0]_{f\rightsquigarrow\{dn,tr,s_{ta_{dsp}}\}|n\rightsquigarrow\{col_1,col_2\}|r\rightsquigarrow\{ga\}}^{f\ \lambda\ s\gg\ \begin{array}{l}Sync_3(r_{\copyright}s)(\texttt{out}\langle r_{\copyright}s\rangle) \\ \mid \texttt{out}\langle f_{\copyright}s\rangle \\ \mid \texttt{out}\langle n_{\copyright}s\rangle\end{array}}$$

(b) Merge the two parallel dispatchers

Figure 5.9: Refactoring the reference implementation

5.3 Refinement via refactoring

The *SC* reference implementation exploits a simple strategy to satisfy the business process: each activity and the corresponding compensation is managed by a dedicated transactional component. It handles *ok* and *ex* events in order to propagate forward and backward-flows. However, the deployment of the system should reflect some additional aspects that does not affect the transactional properties of the process. Let us assume that the reservation of the taxi is performed by the information system of the taxi company. Moreover, assume that the request to redirect the taxi (the compensation) must be sent to an on-board system hosted on the taxi itself. The simplest way to refine the *SC* implementation is to delegate the compensation task to a different component, which represents the on-board service. To this purpose, we use the refactoring rules presented in Section 4.4.1. Informally, we replace the starting network that implements the task *Taxi* (N_{ta}) with the network presented in Figure 5.9a ($N_{ta_{ref}}$).

The refactoring yields a restricted component ta_c that represent the on-board system. It is responsible to perform the compensation and to manage the backward-flow. In fact, *RedirectTaxy* is moved on ta_c towards which *ta* directs *r* and *ex* signals, in accordance with this transformation. Now, the information system of the taxi company (represented by *ta*) needs only to check the successful termination of the reservation, delegating to the on-board system any further exception handling. The refactoring also updates the flows of the components involved in

the taxi reservation (ta, ta_c, $s_{ta_{dsp}}$, s_{ta_f}, s_{ta_r}, $s_{ta_{col}}$), producing the connections depicted in Figure 5.8b.

Now, we focus on the management of the concurrent execution of the activities $DelayNotification$, $TruckReservation$ and $Taxi$. The reference implementation exploits two distinct dispatchers (dsp_1 and dsp_2). The notification of events to these dispatchers are not relevant to the semantic of the network. Indeed, their names are restricted in the network N_S. Let us assume that all components that implement the three involved activities reside on different hosts. In this scenario, the usage of two dispatcher cannot reduce the inter-host communications.

We can merge two parallel dispatchers into one to simplify the *SC* design. We refine the *SC* network by exploiting the refactoring presented in Section 4.4.2. We merge the two parallel dispatcher, migrating the flows of the component dsp_1 to the component dsp_2 and adding the synchronizations of dsp_1. Then we substitute the networks N_{dsp_1} and N_{dsp} with the network $N_{dsp2_{ref}}$ of Figure 5.9. Finally, we update the flows of the system obtaining the connections depicted in Figure 5.8d.

We recall that the theorems presented in Section 4.4 ensure that the refinement is sound respect to the starting reference implementation.

Chapter 6

The Event based Service Coordination framework

The programming model of *SC* presented in Section 3.2 has driven the prototype implementation of the framework Event based Service Coordination (ESC). The name of the framework highlights the adoption of the event notification paradigm to coordinate distributed services. ESC is composed by three main components: a programming language, a run-time that implements the communications and an integrated development environment.

In this chapter we describe the ESC framework and we focus on the interplay between the actual programming primitives and our theoretical results. The full description of the implementation details is out of the scope of this thesis and can be found in [27].

In Section 6.1 we present the Signal Core Language (*SCL*). *SCL* is a Domain Specific Language (DSL) that allows the designer to program distributed systems using the *SC* programming model. The language permits to model an *SC* network, defining the involved components, their behaviors and their interfaces.

In Section 6.2 we summarize the Java Signal Core Layer (*JSCL*). It is a Java API to implement distributed components. *JSCL* represents the tun-time of ESC. A key feature of *JSCL* is its two layer architecture. The more abstract layer provides the primitives used by the programmer to coordinate components, while the lower level permits to specialize the system to exploit a specific network infrastructure. This architecture allows components to abstract from the communication protocols that are used to deliver event notifications.

In Section 6.3 we briefly describe the programming environment of ESC,

called JSCL4Eclipse. We implemented the environment on top of the Eclipse platform [74], in order to integrate the development process with existing programming tools. The environment supports the *SCL* language and provides a model transformation tool that compiles *SCL* programs into Java code.

6.1 The language: SCL

The Signal Core Language (*SCL*) is a textual language to program distributed systems. We defined the language to provide a Domain Specific Language (DSL) that exploits the *SC* programming model. The language is tailored to coordinate services using the event notification paradigm. We present the syntax of the *SCL* language and we highlighting the interplay between the *SCL* primitives and the programming model of *SC*. An *SCL* file represents an *SC* network. A file starts with the declaration of the public topics followed by definition of the involved components:

```
1          global SetOfTopics;
2          ListOfComponent
```

Let $\vec{n}$ be the set of free topic names of ListOfComponents, formally

$$\vec{n} = fn(ListOfComponents) \cap \mathcal{T}$$

an *SCL* file represents the *SC* network:

$$(\nu(\vec{n} \setminus SetOfTopics))ListOfComponents$$

Components are the computational units of the language. A component is defined using the following *SCL* primitive:

```
1      component a {
2        local: SetOfTopics;
3        SetOfFlows;
4        ListOfReactions
5        main {
6          Behavior
7        }
8      }
```

The relation with the *SC* model is straightforward. The primitive declares the *SC* component

$$(\nu\, SetOfTopics)\left(a\,[Behavior]_{SetOfFlows}^{ListOfReactions}\right)$$

Notice that topics specified as local represent restrictions, modeling that all other components does not know these names. The $SetOfFlows$ is simply a set of

comma separated of unit flows. An *SC* unit flow $topic \rightsquigarrow a$ is defined by the primitive $[topic- > a]$. The *ListOfReactions* is a *new line* separated list of reactions, which are the event handlers installed on the component. According to the *SC* model, *SCL* provide two kind of handlers: the lambda and the check reaction. The *SC* check reaction $\tau_{\circledcirc}\tau' \gg B$ is defined by the *SCL* fragment:

```
1       reaction check (topic@topic1) {
2         Behavior
3       }
```

The *SC* lambda reaction $\tau\ \lambda\ \tau' \gg B$ is defined by the *SCL* fragment:

```
1       reaction lambda (topic@x) {
2         Behavior
3       }
```

The semantics of the two handler primitives reflect the semantics of *SC* 3.7. A check reaction can be activated only by events having the session $topic1$. Instead, a lambda reaction handles events independently by their sessions. Moreover, a check reaction is consumed when activated, while a lambda reaction is a singleton that persists on the component interface. The identifier x of the lambda reaction is a variable name; when an event is consumed, its session identifier is assigned to x.

Behaviors represent the actions performed by a component. Any *SC* behavioral primitive has been provided as an *SCL* operator. The *SCL* fragment

```
1       split
2           {Behavior1}
3        || {Behavior2}
```

represents the behavior $B_1 \mid B_2$, which executes concurrently the two contained fragments.

The *SCL* fragment

```
1       addReaction (
2         Reaction
3       );
4       Behavior
```

represents the *SC* reaction update $\texttt{rupd}(R);B$.

The *SCL* fragment

```
1       addFlow ( SetOfFlows );
2       Behavior
```

represents the *SC* flow update $\texttt{fupd}(F);B$.

The *SCL* fragment

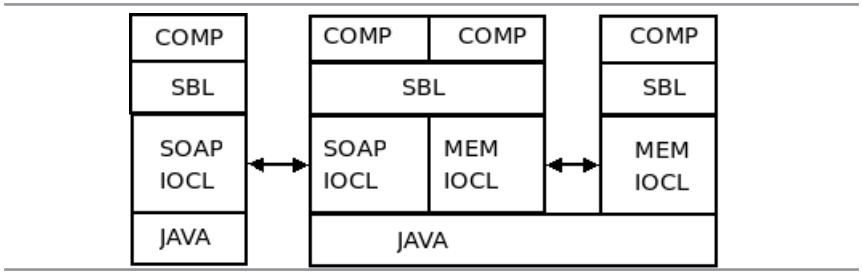


Figure 6.1: Architecture of *JSCL*

```
1       emit ( topic1@topic2 );
2       Behavior
```

represents the *SC* asynchronous signal emission $\mathtt{out}\langle\tau_1 @ \tau_2\rangle; B$.

The *SCL* fragment

```
1       with (topic) {
2         Behavior
3       }
```

represents the *SC* topic generation $(\nu\tau)B$.

Finally, the *SC* internal action $\varepsilon; B$ is represented by the *SCL* fragment

```
1       nop;
2       Behavior
```

6.2 The run-time: *JSCL*

Java Signal Core Layer (*JSCL*) is the run-time of our framework. It has been developed using the two-level architecture depicted in Figure 6.1. The more abstract level is called Signal Based Layer (sbl). It provides to the programmer the primitives to build and coordinate services. The lower level, called Inter Object Communication Layer (iocl), manages the network communication. The role of iocl is to abstract from the network technologies used, in order to hide network heterogeneity to the higher level.

Table 6.1 summarizes the main API of sbl. According to the *SC* programming model, communications are implemented by notification of events. An event is represented as an instance of the class *Signal*. Signals are characterized by their *SignalType*, which embeds the topic and the session identifier of the event.

```
1 SignalType t = new SignalType();
2 t.setTopic(topic);
3 t.setSessionID(topic1);
4 Signal s = new Signal();
5 s.setType(t);
6 component.emit(s);
```

Figure 6.2: Java implementation of $\texttt{out}\langle\tau_{\circledcirc}\tau'\rangle$

According with the *SC* model, both topics and session identifiers are instances of the interface *Topic* and can be freely interchanged.

The class *Component* implements both the interfaces *Handler* and *Emitter*. The former defines the structure of objects able to handle notifications, while the latter specifies objects that can raise events.

A component internally stores its flows, which associate signal types to the addresses of the subscribers. When a component is created, its flows are initially empty. The component can change its flows using the method *addFlow*. Intuitively, the execution of $addFlow(topic, address)$ implements the *SC* behavior $\texttt{fupd}(\tau \rightsquigarrow \{address\})$. The *SC* behavior $\texttt{out}\langle\tau_{\circledcirc}\tau'\rangle$, which raises a new event, can be implemented by Java code In Figure 6.2. When it is executed, the middleware introspects the flows of the emitter to deliver multiple copies of the event to all connected. Signals targeted to components that have not yet published become pending over the network. The infrastructure must implement a buffering mechanism that allows asynchronous communications. This task is delegated to the iocl. It must ensure that pending signals will be consumed when the target component become reachable. Delegate this feature to the iocl permits to specialize the middleware to take advantages of particular IT infrastructure. For example, the iocl could guarantee the persistence and reliability of message queues by serializing them on an available relational database [75].

Each component stores its reactions, which declare how the component handles incoming signals. Analogously to component flows, the set of reactions is initially empty. A component can update its reactions using the method $addReaction(R)$, which implements the *SC* behavior $\texttt{rupd}(R)$. A *JSCL* reaction embeds a *SignalType* and a *Task*. The former defines which events notification can activate the reaction, while the latter defines the behavior to execute. When a reaction consume a signal, the method *handle* of the proper task is invoked by the middleware using a dedicated thread. Reactions are classified into check and lambda, according to their signal type. Reactions having an empty session identifier (null) in the

```
1  class BTask implements Task {
2    void handle(Signal s) {
3      Topic x = s.getSessionID();
4      B
5    }
6  }
7  SignalType t = new SignalType();
8  t.setTopic(topic);
9  t.setSessionID(null);
10 Reaction r = new Reaction();
11 r.setType(t);
12 r.setTask(BTask);
13 component.addReaction(r);
```

Figure 6.3: Java implementation of $\texttt{rupd}\,(\tau\ \lambda\ x \gg B)$

signal type are considered lambda and will not be deleted after their activation. We provide an example to illustrate this mechanism. Let *ex* be the *SC* behavior $\texttt{rupd}\,(\tau\ \lambda\ x \gg B)$ that adds a new lambda reaction. Upon the reception of an event having topic τ, the reaction can be activated and the received session identifier is substituted to x in the behavior B. Finally, B is concurrently executed inside the component. The behavior *ex* can be implemented by Java code in Figure 6.3. Notice that the variable binding is implemented by the assignment to the variable x of the session identifier of the received signal.

The iocl has been introduced to abstract from the peculiarities of the technologies used to distribute the *JSCL* components. The iocl defines the minimal functionalities that a technological infrastructure must provide to allow the interaction of *JSCL* components. The iocl is responsible of the delivery of events raised by components. A key feature of *JSCL* is that several iocls can coexist under the same sbl. Each iocl instance is responsible to manage a specific network infrastructure. The set of all iocl used by a system represents the *SC* notion of network, providing the functionality of publish components, identify them by a unique name and deliver the signal notifications.

Currently, the prototype implementation supports three iocls, allowing to deploy components to as many communication technologies: shared memory, socket and HTTP SOAP. A component must be deployed on at least one iocl in order to be notified about events.

The iocl is also responsible to serialize messages according to the adopted network. For example, if components are deployed as web services, notifica-

tions will be serialized as XML documents and delivered inside a SOAP envelopes. The handling of notifications and their serialization/deserialization allows an high degree of interoperability. Components residing on heterogeneous networks (e.g. mobile phones, sensor networks, web services etc.) can coexist and inter-operate, under the assumption that these systems adhere to a common programming model: the one supplied by the sbl. The separation among the aspects related to the network infrastructure and the behavior of components allows to change the iocls without compromising the design of the components.

6.3 The programming environment: JSCL4Eclipse

The JSCL4Eclipse is our programming environment. It provide the development tools to support the designer to model and implement service oriented applications on top of the ESC framework. The environment is integrated within the Eclipse platform [74] in order to simplify the adoption of the framework. The environment consists of three tools, each of them developed as an independent Eclipse plug-in:

- a visual editor to graphically model the flows of components
- a textual editor for the *SCL* language, supporting code completion and syntax error checking
- a model transformation that "compiles" *SCL* files into Java classes that exploit the *JSCL* API.

The tools ha been developed with the aid of the Graphical Modeling Framework (GMF [76]), which provides a generative infrastructure for developing editors based on the Eclipse Modeling Framework (EMF [77]) and on the Graphical Eclipse Framework (GEF [78]).

The development cycle of distributed applications using ESC can be summarized as follows:

1. The designer exploits the visual editor to graphically model the set of components involved into the coordination. The designer can add reactions, modeling the kind (check or lambda) and the event topics handled by them. It also models the flows of the components drawing arrows among them.

2. The environment associates to each graphical model a *SCL* file. The graphical model and the corresponding *SCL* file are maintained coherent by the tools. Every updates, insertion and deletion on a term on the graphical

model are reflected on the *SCL* file and vice versa. Therefore, after that the designer modeled the components on the visual editor, a skeleton *SCL* code has been automatically generated. Now, the programmer can implement the behaviors of the reactions that cannot be graphically represented.

3. When all missing behavior has been implemented in *SCL*, the programmer can generate the Java source code. The output code exploits the *JSCL* API to implement the coordination specified by the platform independent model. Now, the implementation detail that does not affect the coordination can be directly added into the generated source code using the Java programming language.

6.4 Related Works

Software companies have promoted frameworks to deal with SOA. Here we only consider the SCA proposal. The Service Component Architecture (SCA [79]) and its standard specification Composite Services Architecture (CSA [80]) exploit a component based approach to decouple the service logic from the process logic. Both SCA and ESC exploit a notion of connections among components, allowing to implement complex interactions by configuring the component links. Moreover, both approach exploit a multi-layered architecture to decouple the programming interface from the underlying network technologies used to exchange messages.

However, the distinguished feature of ESC is the strict interplay between the actual programming primitives and the foundational mechanisms. In Section 4.4 we presented some refactoring rules that handle specific aspects of systems. We proved that these transformations are sound, namely the semantics of the system is unchanged. We have already pointed out that the programming frameworks can greatly increase the benefits of the MDD approach only if the semantics of the DSLs are formally defined.

```
1  class Signal {
2     SignalType getType();
3     void setType(SignalType sigT);
4  }
5
6  class SignalType {
7     Topic getTopic();
8     void setTopic(Topic t);
9     Topic getSesisonID();
10    void setSessionID(Topic s);
11 }
12
13 interface Task {
14    void handle(Signal s);
15 }
16
17 class Reaction {
18    setType(SignalType sigT);
19    setTask(Task t);
20 }
21
22 class Handler {
23    void addReaction(Reaction r);
24 }
25
26 class Emitter {
27    void emit(Signal s);
28    void addFlow (Topic sigT, ComponentAddress target);
29 }
30
31 class Component implements Handler, Emitter {
32 }
```

Table 6.1: Sketch of main sbl API

Chapter 7

Concluding remarks and Future works

In this thesis we presented our framework for service coordination. We focused on the theoretical aspects of our work and the benefits provided by the formal methodologies provided to the actual programming environment. The main contribution of this thesis are:

- The definition of the *SC* family of process calculi. Signal Calculus (*SC*) is our proposal to design services coordination using the event notification paradigm. The calculus is the formal machinery that has driven the *SCL* programming language and the its run-time, called *JSCL*.
- The definition of the choreography model of *SC*: the Network Coordination Policy calculus (*NCP*). We described the observational semantics of *NCP* to provide a formal definition correctness. We related the semantics of *SC* and *NCP* to allows *SC* designs to be verified respect to *NCP* policies.
- The implementation of saga calculus using *SC*, which provides a reference implementation of long running transaction.
- The definition of *sound* refactoring methodologies that permit to handle aspects related to the deployment phase of the *SC* models without affecting their semantics

Since we focus on service choreography, the thesis handles the issues specific to the design and the implementation of services. In particular we focus on the

verification of a local design respect to a global specification. We implement the saga calculus using *SC* models to highlight the benefits of a SOA framework with formal grounds.

Future Work

The *SC* design language has been equipped with three styles of event notification interactions, namely topic-based, content-based and type-based. However, some further investigations are still needed. For example, the type-based *SC* dialect uses C-Semiring [63] to represent properties of events. This structures have already been exploited to represent quantitative aspects of systems. This suggests that *SC* can be extended to deal with *quality of service* issues. Moreover, the type-based *SC* dialect could be generalized in order to provide a standard framework to manage type based interactions. For example, the XDuce [81] language could provide the type system needed to filter events using hierarchical structures similar to XML.

The *NCP* calculus is strictly related to the *SC* programming model, allowing to simplify the reasoning techniques. However, some differences between the two approaches can be studied. For example *NCP* could be equipped with fusion-based interactions [82] to provide a further abstraction mechanism for choreography. Moreover, we plan to equip the calculus with a variety of verification mechanisms like model checking and property checking via bisimilarity.

In [40] we proposed a debugging model that exploits causal petri nets to track of the progress of *SC* models. The intuitive idea can be used to provide new tools to simplify the tuning of distributed applications programmed in *JSCL*. Moreover, we are investigating the interplay between the debugging model and long running transactions. Usually, it is not possible to undo all the effects of a debugging session of a SOA application. Thus, we want to apply the notion of compensation to recover partial executions of the debugger. We plan to extend the set of refactoring rules for the *SC* models that implement saga processes to deal with this possibility.

Finally, the multicast interaction pattern of *SC* and *NCP* can be suitably applied in a wide range of systems different from SOA, including grid and clouds computing. We plan to investigate some usage scenario to verify the expressiveness of our framework.

Appendix A

Proof of Theorems in Section 4.1

A.1 Proof of Theorem 1

Let $\langle G\ ;\ P\rangle$ be a *NCP* state and G_1 a network topology, if $P \perp G_1$ then

$$\langle G\ ;\ P\rangle \xrightarrow{\alpha} \langle G'\ ;\ P'\rangle \textit{ if and only if } \langle G \uplus G_1\ ;\ P\rangle \xrightarrow{\alpha} \langle G' \uplus G_1\ ;\ P'\rangle$$

The theorem can be proved by induction over the *NCP* transition rules. We report the proof only for the most interesting cases; `fupd`, `emit` and `open`.

Rule `fupd`

If $\langle G\ ;\ P\rangle$ performs an action by the rule `fupd` then

$$\langle G\ ;\ P\rangle = \langle G\ ;\ \texttt{fupd}\,(F)\,@A.P_0\rangle \xrightarrow{\varepsilon} \langle G \uplus a \boxtimes F\ ;\ P_0\rangle$$

The state $\langle G \uplus G_1\ ;\ P\rangle$ can be written as $\langle G \uplus G_1\ ;\ \texttt{fupd}\,(F)\,@A.P_0\rangle$. This can performs the same action using the rule `fupd`

$$\langle G \uplus G_1\ ;\ \texttt{fupd}\,(F)\,@A.P_0\rangle \xrightarrow{\varepsilon} \langle G \uplus G_1 \uplus a \boxtimes F\ ;\ P_0\rangle = \langle G \uplus a \boxtimes F \uplus G_1\ ;\ P_0\rangle$$

Rule `emit`

If $\langle G\ ;\ P\rangle$ performs an action by the rule `emit` then

$$\langle G\ ;\ P\rangle = \langle G\ ;\ \overline{\tau}\,\tau'@a.P_0\rangle \xrightarrow{\varepsilon} \left\langle G\ ;\ P_0 \parallel \prod_{b\in G(\tau,a)} \langle \tau \circledcirc s\rangle @b \right\rangle$$

The state $\langle G \uplus G_1\ ;\ P\rangle$ can be written as $\langle G\uplus G_1\ ;\ \overline{\tau}\,\tau'@a.P_0\rangle$. This can perform the same action using the rule `emit`

$$\langle G\uplus G_1\ ;\ \overline{\tau}\,\tau'@a.P_0\rangle \xrightarrow{\varepsilon} \left\langle G\uplus G_1\ ;\ P_0 \parallel \prod_{b\in (G\uplus G_1)(\tau,a)} \langle \tau \circledcirc s\rangle @b \right\rangle$$

Since $a \in Sbj(P)$ then $a \notin Sbj(G_1)$. This guarantees that $(G\uplus G_1)(\tau,a) = G(\tau,a)$.

Rule `open`

If $\langle G\ ;\ P\rangle$ perform an action involving by rule `open` then

$$\langle G\ ;\ P\rangle = \langle G\ ;\ (\nu\,\tau : T)\,P_0\rangle \xrightarrow{\langle \tau' \circledcirc (\tau:T)\rangle @a} \langle G\uplus \tau \boxdot T\ ;\ P_0'\rangle$$

$$\textit{and}\ \ \langle G\uplus \tau\boxdot T\ ;\ P_0\rangle \xrightarrow{\langle \tau'\circledcirc\tau\rangle @a} \langle G\uplus\tau\boxdot T\ ;\ P_0'\rangle$$

Since $Sbj(\langle G\ ;\ (\nu\,\tau:T)\,P_0\rangle) = Sbj(\langle G\uplus\tau\boxdot T\ ;\ P_0\rangle)$ then $\langle G\uplus\tau\boxdot T\ ;\ P_0\rangle \perp G_1$. By the induction hypothesis

$$\langle G\uplus\tau\boxdot T\uplus G_1\ ;\ P_0\rangle \xrightarrow{\langle\tau'\circledcirc\tau\rangle@a} \langle G\uplus\tau\boxdot T\uplus G_1\ ;\ P_0'\rangle$$

Then the rule `open` can be applied to $\langle G\uplus G_1\ ;\ P\rangle$, obtaining

$$\langle G\uplus G_1\ ;\ P\rangle = \langle G\uplus G_1\ ;\ (\nu\,\tau:T)\,P_0\rangle \xrightarrow{\langle\tau'\circledcirc(\tau:T)\rangle@a} \langle G\uplus G_1\uplus\tau\boxdot T\ ;\ P_0'\rangle$$

A.2 Lemma 1

Lemma 1 *Let* $\langle G\ ;\ P\rangle$ *be a* NCP *state, if* $\langle G\ ;\ P\rangle \xrightarrow{\langle\tau\circledcirc\tau'\rangle@a} \langle G\ ;\ P'\rangle$ *then*

$$\langle G\ ;\ P\rangle \sim \langle G\ ;\ P' \parallel \langle \tau\circledcirc\tau'\rangle @a\rangle$$

The lemma can be easily proved by induction over *NCP* transition rules.

A.3 Proof of Theorem 3

Let $\langle G_1 \,;\, P_1 \rangle$ and $\langle G_2 \,;\, P_2 \rangle$ be two *NCP* state such that $\langle G_1 \,;\, P_1 \rangle \sim \langle G_2 \,;\, P_2 \rangle$ and $\langle G_1 \,;\, P_1 \rangle \xrightarrow{\tau\ \tau'@a} \langle G_1 \,;\, P_1' \rangle$, then and one of the following two statements must hols:

- $\langle G_2 \,;\, P_2 \rangle \xrightarrow{\tau\ \tau'@a} \langle G_2 \,;\, P_2' \rangle$ and $\langle G_1 \,;\, P_1' \rangle \sim \langle G_2 \,;\, P_2' \rangle$
- $\langle G_2 \,;\, P_2 \rangle \xrightarrow{\varepsilon} \langle G_2' \,;\, P_2' \rangle$ and $\langle G_1 \,;\, P_1' \rangle \sim \langle G_2' \,;\, P_2' \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \rangle$

Since the transition rules `async` can be applied to any *NCP* state, we can compose the first *NCP* state with a compatible envelope

$$\langle G_1 \,;\, P_1 \rangle \xrightarrow{(\tau\ \tau'@a)} \big\langle G_1 \,;\, P_1 \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle$$

Since $\langle G_1 \,;\, P_1 \rangle$ and $\langle G_2 \,;\, P_2 \rangle$ are bisimilar, the latter state must perform the same action and

$$\big\langle G_1 \,;\, P_1 \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle \sim \big\langle G_2 \,;\, P_2 \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle$$

We can exploit the hypothesis of the theorem to apply the transition rule `com` and obtain

$$\big\langle G_1 \,;\, P_1 \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle \xrightarrow{\varepsilon} \big\langle G_1 \,;\, P_1' \big\rangle$$

Then, we use the bisimulation hypothesis to discover that also the other state performs a silent action

$$\big\langle G_2 \,;\, P_2 \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle \xrightarrow{\varepsilon} \big\langle G_2' \,;\, P_2' \big\rangle$$

The last transition can be obtained by using the rules `com` or `par`. We separate the two cases.

Rule `com`

If the rule `com` has been applied, it is trivial to prove

$$\langle G_2 \,;\, P_2 \rangle \xrightarrow{\tau\ \tau'@a} \big\langle G_2 \,;\, P_2' \big\rangle \wedge G_2' = G_2 \wedge \big\langle G_1 \,;\, P_1' \big\rangle \sim \big\langle G_2 \,;\, P_2' \big\rangle$$

Rule `par`

If the rule `par` has been applied then one of the two involved policy must perform a silent action. Since this cannot be performed by the envelope, we must conclude

$$\big\langle G_2 \,;\, P_2' \big\rangle \xrightarrow{\varepsilon} \big\langle G_2' \,;\, P_2'' \big\rangle \wedge \big\langle G_1 \,;\, P_1' \big\rangle \sim \big\langle G_2' \,;\, P_2'' \parallel \langle \tau_{\circledcirc}\tau' \rangle @a \big\rangle$$

A.4 Proof of Theorem 4

Let $\langle G\ ;\ P\rangle$ and $\langle G'\ ;\ P'\rangle$ be two *NCP* states such that $\langle G\ ;\ P\rangle \sim \langle G'\ ;\ P'\rangle$, $\tau \in \mathcal{T}$ be a topic, $T = G(\tau)$ and $T' = G'(\tau)$ be the topic graphs of τ in the two network topologies G and G' respectively, then:

1. if $G = (\vec{b}, E)$ and $G' = (\vec{b}', E')$ then $\left\langle (\vec{b} \cup \vec{a}, E)\ ;\ P \right\rangle \sim \left\langle (\vec{b}' \cup \vec{a}, E')\ ;\ P' \right\rangle$
2. $\langle G \setminus \tau \boxdot T\ ;\ (\nu\ \tau : T)\, P\rangle \sim \langle G' \setminus \tau \boxdot T'\ ;\ (\nu\ \tau : T')\, P'\rangle$

A.4.1 Proof of statement 1 of Theorem 4

We start noticing that if $\vec{a} \cap fn(\alpha) = \emptyset$ then $\left\langle (\vec{b}, E)\ ;\ P \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b}, E_1)\ ;\ P_1 \right\rangle$ if and only if $\left\langle (\vec{b} \cup \vec{a}, E)\ ;\ P \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b} \cup \vec{a}, E_1)\ ;\ P_1 \right\rangle$.
We prove the statement by showing that the following relation is an *NCP*-bisimulation:

$$\mathcal{B} = \left\{ \left(\left\langle (\vec{b} \cup \vec{a}, E)\ ;\ P \right\rangle, \left\langle (\vec{b}' \cup \vec{a}', E')\ ;\ P' \right\rangle \right) \mid \left\langle (\vec{b}, E)\ ;\ P \right\rangle \sim \left\langle (\vec{b}', E')\ ;\ P' \right\rangle \right\}$$

All transition observable transitions, except opens

We test the first condition of bisimilarity relations: let

$$\left\langle (\vec{b} \cup \vec{a}, E)\ ;\ P \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b} \cup \vec{a}, E_1)\ ;\ P_1 \right\rangle\ ,\ \alpha \in \{\varepsilon, \langle \tau \circledcirc \tau' \rangle @ a, (\tau\tau' @ a)\}\ and\ a \notin \vec{b} \cup \vec{a}$$

then

$$\left\langle (\vec{b}, E)\ ;\ P \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b}, E_1)\ ;\ P_1 \right\rangle$$

The bisimilarity condition of the $\mathcal{B}$ relation ensures that

$$\left\langle (\vec{b}', E')\ ;\ P' \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b}', E_1')\ ;\ P_1' \right\rangle\ and\ \left\langle (\vec{b}, E_1)\ ;\ P_1 \right\rangle \sim \left\langle (\vec{b}', E_1')\ ;\ P_1' \right\rangle$$

then

$$\left\langle (\vec{b}' \cup \vec{a}, E')\ ;\ P' \right\rangle \xrightarrow{\alpha} \left\langle (\vec{b}' \cup \vec{a}, E_1')\ ;\ P_1' \right\rangle$$

$$and\ \left(\left\langle (\vec{b} \cup \vec{a}, E_1)\ ;\ P_1 \right\rangle, \left\langle (\vec{b}' \cup \vec{a}, E_1')\ ;\ P_1' \right\rangle \right) \in \mathcal{B}$$

Open transitions

Now we test the second condition of bisimulation relations: let

$$\left\langle (\vec{b}\cup\vec{a},E)\ ;\ P\right\rangle \xrightarrow{\langle \tau \copyright (\tau':T)\rangle @a} \left\langle (\vec{b}\cup\vec{a},E_1)\ ;\ P_1\right\rangle, \tau' \notin fn(E',P') \text{ and } a\notin \vec{b}\cup\vec{a}$$

then

$$\left\langle (\vec{b},E)\ ;\ P\right\rangle \xrightarrow{\langle \tau \copyright (\tau':T)\rangle @a} \left\langle (\vec{b},E_1)\ ;\ P_1\right\rangle$$

The bisimilarity condition of the $\mathcal{B}$ relation ensures that

$$\left\langle (\vec{b}',E')\ ;\ P'\right\rangle \xrightarrow{\langle \tau \copyright (\tau':T')\rangle @a} \left\langle (\vec{b}',E_1')\ ;\ P_1'\right\rangle \text{ and } \left\langle (\vec{b},E_1)\ ;\ P_1\right\rangle \sim \left\langle (\vec{b}',E_1')\ ;\ P_1'\right\rangle$$

then

$$\left\langle (\vec{b}'\cup\vec{a},E')\ ;\ P'\right\rangle \xrightarrow{\langle \tau \copyright (\tau':T')\rangle @a} \left\langle (\vec{b}'\cup\vec{a},E_1')\ ;\ P_1'\right\rangle$$
$$\text{and } \left(\left\langle (\vec{b}\cup\vec{a},E_1)\ ;\ P_1\right\rangle, \left\langle (\vec{b}'\cup\vec{a},E_1')\ ;\ P_1'\right\rangle\right) \in \mathcal{B}$$

A.4.2 Proof of statement 2 of Theorem 4

We prove the statement by showing that the following relation is an *NCP*-bisimulation:

$$\mathcal{B} = \left\{ \begin{array}{c} (\langle G\setminus \tau\boxdot T\ ;\ (\nu\,\tau:T)\,P\rangle, \langle G'\setminus \tau\boxdot T'\ ;\ (\nu\,\tau:T')\,P'\rangle)\ | \\ \langle G\ ;\ P\rangle \sim \langle G'\ ;\ P'\rangle, T = G(\tau) \text{ and } T' = G'(\tau) \end{array} \right\} \cup \sim$$

All observable transitions, except opens

We test the first condition of bisimilarity relations: let

$$\langle G\setminus \tau\boxdot T\ ;\ (\nu\,\tau:T)\,P\rangle \xrightarrow{\alpha} \langle G_1\ ;\ P_1\rangle\ ,\ \alpha\in\{\varepsilon, \langle \tau'\circledcirc\tau''\rangle @a, (\tau'\,\tau''@a)\} \text{ and } a\notin bn(G)$$

The transition can be performed only by applying the `new` transition rule, since the policy is a topic restriction. The hypothesis of the rule ensures

$$\langle G\ ;\ P\rangle \xrightarrow{\alpha} \langle G_2\ ;\ P_2\rangle, \tau\notin n(\alpha), T_2 = G_2(\tau) \text{ and } \langle G_1\ ;\ P_1\rangle = \langle G_2\setminus \tau\boxdot T_2\ ;\ (\nu\,\tau:T_2)\,P_2\rangle$$

The bisimilarity condition of the $\mathcal{B}$ relation ensures

$$\left\langle G'\ ;\ P'\right\rangle \xrightarrow{\alpha} \left\langle G_2'\ ;\ P_2'\right\rangle \text{ and } \left\langle G_2\ ;\ P_2\right\rangle \sim \left\langle G_2'\ ;\ P_2'\right\rangle$$

Let $T_2' = G_2'(\tau)$, we can apply the transition rule `new`, obtaining that

$$\left\langle G'\setminus \tau\boxdot T'\ ;\ \left(\nu\,\tau:T'\right)P'\right\rangle \xrightarrow{\alpha} \left\langle G_2'\setminus \tau\boxdot T_2'\ ;\ \left(\nu\,\tau:T_2'\right)P_2'\right\rangle = \left\langle G_1'\ ;\ P_1'\right\rangle$$

and finally $(\langle G_1\ ;\ P_1\rangle, \langle G_1'\ ;\ P_1'\rangle) \in \mathcal{B}$

Opens Transitions

Now we test the second condition bisimilarity relations: let

$$\langle G \setminus \tau \boxdot T \;;\; (\nu\, \tau : T)\, P\rangle \xrightarrow{\langle \tau' \copyright (\tau'' : T'')\rangle @ a} \langle G_1 \;;\; P_1\rangle, \tau'' \notin fn(G', P') \text{ and } a \notin bn(G)$$

We distinguish the two possible cases: when $\tau \neq \tau''$ and when $\tau = \tau''$. If $\tau \neq \tau''$ the proof is trivial and exploits the strategy presented above.

If $\tau'' = \tau$ then the rule `open` has been applied and $T'' = T$ must holds. The hypothesis of the rule ensures

$$\langle G \;;\; P\rangle \xrightarrow{\langle \tau' \copyright \tau\rangle @ a} \langle G \;;\; P_2\rangle, \tau' \neq \tau \text{ and } \langle G_1 \;;\; P_1\rangle = \langle G \;;\; P_2\rangle$$

The bisimilarity condition of the $\mathcal{B}$ relation ensures that

$$\left\langle G' \;;\; P'\right\rangle \xrightarrow{\langle \tau' \copyright \tau\rangle @ a} \left\langle G' \;;\; P'_2\right\rangle \text{ and } \langle G \;;\; P_2\rangle \sim \left\langle G' \;;\; P'_2\right\rangle$$

We can apply the transition rule open, obtaining that

$$\left\langle G' \setminus \tau \boxdot T' \;;\; \left(\nu\, \tau : T'\right) P'\right\rangle \xrightarrow{\langle \tau' \copyright (\tau : T')\rangle @ a} \left\langle G' \;;\; P'_2\right\rangle = \left\langle G'_1 \;;\; P'_1\right\rangle$$

and finally $(\langle G_1 \;;\; P_1\rangle, \langle G'_1 \;;\; P'_1\rangle) \in \mathcal{B}$, since they are bisimilar.

A.5 Proof of Theorem 5

Let $S_{P_1} = \langle G_1 \;;\; P_1\rangle$, $S_{P_2} = \langle G_2 \;;\; P_2\rangle$, $S_{Q_1} = \langle I_2 \;;\; Q_2\rangle$ and $S_{Q_2} = \langle I_2 \;;\; Q_2\rangle$ be *NCP* states, such that $S_{P_1} \sim S_{Q_2}$ and $S_{Q_1} \sim S_{Q_2}$ If $S_{P_1} \perp S_{Q_1}$ and $S_{P_2} \perp S_{Q_2}$ then $\langle G_1 \uplus I_1 \;;\; P_1 \parallel Q_1\rangle \sim \langle G_2 \uplus I_2 \;;\; P_2 \parallel Q_2\rangle$

We prove the theorem by showing that the following relation is a bisimulation:

$$\mathcal{B} = \left\{ \begin{array}{r} (\langle G_1 \uplus I_1 \;;\; P_1 \parallel Q_1\rangle, \langle G_2 \uplus I_2 \;;\; P_2 \parallel Q_2\rangle) \\ \mid \langle G_1 \;;\; P_1\rangle \sim \langle G_2 \;;\; P_2\rangle \wedge \langle I_1 \;;\; Q_1\rangle \sim \langle I_2 \;;\; Q_2\rangle \\ \wedge\, Sbj(\langle G_1 \;;\; P_1\rangle) \cap Sbj(\langle I_1 \;;\; Q_1\rangle) = \emptyset \\ \wedge\, Sbj(\langle G_2 \;;\; P_2\rangle) \cap Sbj(\langle I_2 \;;\; Q_2\rangle) = \emptyset \end{array} \right\}$$

We report the proof for the most interesting transition rules that can be applied to the state $\langle G_1 \uplus I_1 \;;\; P_1 \parallel Q_1\rangle$: the rules `par` and `com`

Rule `par`

If the state $\langle G_1 \uplus I_1 \; ; \; P_1 \parallel Q_1 \rangle$ performs an action involving the rule `par` then one of the two contained policies performs the same action. We suppose that the action is performed by P_1. The hypothesis of the transition rule ensures

$$\langle G_1 \uplus I_1 \; ; \; P_1 \parallel Q_1 \rangle \xrightarrow{\alpha} \langle G' \; ; \; P_1' \parallel Q_1 \rangle \; \wedge \; \langle G_1 \uplus I_1 \; ; \; P_1 \rangle \xrightarrow{\alpha} \langle G' \; ; \; P_1' \rangle$$

Since we are checking bisimilarity, we can check only transitions $\alpha \neq \tau \; \tau'@a$. We exploit the disjunction of the subjects of the states $\langle G_1 \; ; \; P_1 \rangle$ and $\langle I_1 \; ; \; Q_1 \rangle$ to apply the theorem 1, which guarantees

$$\langle G_1 \; ; \; P_1 \rangle \xrightarrow{\alpha} \langle G_1' \; ; \; P_1' \rangle \; \wedge \; G' = G_1' \uplus G$$

If $\alpha \in \{(\tau \; \tau'@a), \langle \tau_{\circledcirc} \tau' \rangle @a, \varepsilon\}$, the bisimilarity condition of the realtion $\mathcal{B}$ guarantees that

$$\langle G_2 \; ; \; P_2 \rangle \xrightarrow{\alpha} \langle G_2' \; ; \; P_2' \rangle \; \wedge \; \langle G_1' \; ; \; P_1' \rangle \sim \langle G_2' \; ; \; P_2' \rangle$$

We exploit the disjunction of the subjects of the states $\langle G_2 \; ; \; P_2 \rangle$ and $\langle I_2 \; ; \; Q_2 \rangle$ to apply the theorem 1, proving

$$\langle G_2 \uplus I_2 \; ; \; P_2 \rangle \xrightarrow{\alpha} \langle G_2' \uplus I_2 \; ; \; P_2' \rangle$$

Now, we apply the transition rule `par`

$$\langle G_2 \uplus I_2 \; ; \; P_2 \parallel Q_2 \rangle \xrightarrow{\alpha} \langle G_2' \uplus I_2 \; ; \; P_2' \parallel Q_2 \rangle$$

Since the subjects of this *NCP* policies cannot increase (Theorem 2) we argue that

$$\langle G_1' \; ; \; P_1' \rangle \perp \langle I_1 \; ; \; Q_1 \rangle \; \textit{and} \; \langle G_2' \; ; \; P_2' \rangle \perp \langle I_2 \; ; \; Q_2 \rangle$$

then

$$\big(\langle G_1' \uplus I_1 \; ; \; P_1' \parallel Q_1 \rangle, \langle G_2' \uplus I_2 \; ; \; P_2' \parallel Q_2 \rangle \in \mathcal{B} \big)$$

If $\alpha = \langle \tau_{\circledcirc} (\tau' : T_1) \rangle @a$, the bisimilarity condition of the relation $\mathcal{B}$ guarantees that exists T_2 such that

$$\langle G_2 \; ; \; P_2 \rangle \xrightarrow{\langle \tau \copyright (\tau' : T_2) \rangle @a} \langle G_2' \; ; \; P_2' \rangle \; \wedge \; \langle G_1' \; ; \; P_1' \rangle \sim \langle G_2' \; ; \; P_2' \rangle$$

The end of the proof is trivial, because we can exploit the same strategy used for $\alpha \in \{(\tau \; \tau'@a), \langle \tau_{\circledcirc} \tau' \rangle @a, \varepsilon\}$.

Rule COM

If the state $\langle G_1 \uplus I_1 \ ; \ P_1 \parallel Q_1 \rangle$ performs an action involving the rule `com` then one of the two policies must perform an event notification, while the other one must contain an input suitable for the same event. We suppose that the input is performed by the policy P_1 and the signal notification by Q_1. The hypothesis of the rule `com` ensure

$$\langle G_1 \uplus I_1 \ ; \ P_1 \parallel Q_1 \rangle \xrightarrow{\varepsilon} \langle G_1 \uplus I_1 \ ; \ P_1' \parallel Q_1' \rangle \ \wedge$$
$$\langle G_1 \uplus I_1 \ ; \ P_1 \rangle \xrightarrow{\tau \ \tau'@a} \langle G_1 \uplus I_1 \ ; \ P_1' \rangle \ \wedge \ \langle G_1 \uplus I_1 \ ; \ Q_1 \rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle G_1 \uplus I_1 \ ; \ Q_1' \rangle$$

As usual we can use the theorem 1 to separate the network topologies of the policies P_1 and P_2

$$\langle G_1 \ ; \ P_1 \rangle \xrightarrow{\tau \ \tau'@a} \langle G_1 \ ; \ P_1' \rangle \ \wedge \ \langle I_1 \ ; \ Q_1 \rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle I_1 \ ; \ Q_1' \rangle$$

The bisimulation condition of the relation $\mathcal{B}$ ensures that

$$\langle I_2 \ ; \ Q_2 \rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle I_2 \ ; \ Q_2' \rangle \ \wedge \ \langle I_1 \ ; \ Q_1' \rangle \sim \langle I_2 \ ; \ Q_2' \rangle$$

Now, the hypothesis of the Theorem 3 are verified for both the states $\langle G_1 \ ; \ P_1 \rangle$ and $\langle G_2 \ ; \ P_2 \rangle$. We separate the proof for the two statements of the Theorem 3.

Statement 1

If the statement 1 of the Theorem 3 holds then

$$\langle G_2 \ ; \ P_2 \rangle \xrightarrow{\tau \ \tau'@a} \langle G_2 \ ; \ P_2' \rangle \ \wedge \ \langle G_1 \ ; \ P_1' \rangle \sim \langle G_2 \ ; \ P_2' \rangle$$

As usual, we use the theorem 1 extend the network topologies of the policies P_2 and Q_2

$$\langle G_2 \uplus I_2 \ ; \ P_2 \rangle \xrightarrow{\tau \ \tau'@a} \langle G_2 \uplus I_2 \ ; \ P_2' \rangle \ \wedge \ \langle G_2 \uplus I_2 \ ; \ Q_2 \rangle \xrightarrow{\langle \tau \copyright \tau' \rangle @a} \langle G_2 \uplus I_2 \ ; \ Q_2' \rangle$$

Now, we apply the transition rule `com`

$$\langle G_2 \uplus I_2 \ ; \ P_2 \parallel Q_2 \rangle \xrightarrow{\varepsilon} \langle G_2 \uplus I_2 \ ; \ P_2' \parallel Q_2' \rangle$$

Since the subjects of this *NCP* policies are not increases (Theorem 2) we argue that

$$\langle G_1 \ ; \ P_1' \rangle \perp \langle I_1 \ ; \ Q_1' \rangle \ \textit{and} \ \langle G_2 \ ; \ P_2' \rangle \perp \langle I_2 \ ; \ Q_2' \rangle$$

and finally

$$\left(\langle G_1 \uplus I_1 \ ; \ P_1' \parallel Q_1' \rangle, \langle G_2 \uplus I_2 \ ; \ P_2' \parallel Q_2' \rangle \right) \in \mathcal{B}$$

Statement 2

If the statement 1 of the Theorem 3 holds then

$$\langle G_2 \,;\, P_2\rangle \xrightarrow{\varepsilon} \langle G_2' \,;\, P_2''\rangle \;\wedge\; \langle G_1 \,;\, P_1'\rangle \sim \langle G_2' \,;\, P_2'' \parallel \langle \tau_{\circledcirc}\tau'\rangle @ a\rangle$$

As usual, we use the Theorem 1 to extend the context of the policies

$$\langle G_2 \uplus I_2 \,;\, P_2\rangle \xrightarrow{\varepsilon} \langle G_2' \uplus I_2 \,;\, P_2'\rangle \;\wedge\; \langle G_2' \uplus I_2 \,;\, Q_2\rangle \xrightarrow{\langle \tau \circledcirc \tau'\rangle @ a} \langle G_2' \uplus I_2 \,;\, Q_2'\rangle$$

Now, we use the rule `par` to obtain

$$\langle G_2 \uplus I_2 \,;\, P_2 \parallel Q_2\rangle \xrightarrow{\varepsilon} \langle G_2' \uplus I_2 \,;\, P_2'' \parallel Q_2\rangle$$

Since the policy Q_2 perform a signal notification, the Lemma 1 ensures

$$\langle G_2' \uplus I_2 \,;\, P_2'' \parallel Q_2\rangle \sim \langle G_2' \uplus I_2 \,;\, P_2'' \parallel \langle \tau_{\circledcirc}\tau'\rangle @ a \parallel Q_2'\rangle$$

Finally, we use the Theorem 2 to verify that the obtained policies are in the relation $\mathcal{B}$

$$\big(\langle G_1 \uplus I_1 \,;\, P_1' \parallel Q_1'\rangle, \langle G_2 \uplus I_2 \,;\, P_2' \parallel \langle \tau_{\circledcirc}\tau'\rangle @ a \parallel Q_2'\rangle\big) \in \mathcal{B}$$

Appendix B

Proof of theorems in Section 4.2

B.1 Lemma 2

Lemma 2 *Let* $N = a[B]^R_F$ *be an* SC *component and* $[\![N]\!]$ *its* NCP *translation. If* $[\![N]\!] \xrightarrow{\varepsilon} \langle G_1 \; ; \; P_1 \rangle$ *it holds that:*

1. $\langle G_1 \; ; \; P_1 \rangle = \langle G_1 \; ; \; P_1' \parallel [\![R]\!]_a \rangle$
2. $N \rightarrow N'$
3. $[\![N']\!] = \langle G_1 \; ; \; P_1 \rangle$

First we highlight that the *NCP* state $\langle a \boxtimes F \; ; \; [\![B]\!]_a \rangle$ can perform only silent actions, and that the *NCP*state $\langle a \boxtimes F \; ; \; [\![R]\!]_a \rangle$ can perform only input actions. These statements can be trivially verified by the transformation rules for behaviors and reactions, respectively.

The hipsters of the lemma require that the transformation of the *SC* network N performs a silent action, more formally:

$$[\![N]\!] = \langle a \boxtimes F \; ; \; [\![B]\!]_a \parallel [\![R]\!]_a \rangle \xrightarrow{\varepsilon} \langle G_1 \; ; \; P_1 \rangle$$

The parallel composition of two policies can perform this action exploiting the transition rules `close`, `com` or `par`. Since $\langle a \boxtimes F \; ; \; [\![B]\!]_a \rangle$ cannot perform output actions, only the rule `par` is suitable. Moreover we remarked that $\langle a \boxtimes F \; ; \; [\![R]\!]_a \rangle$ cannot perform an internal action, then

- $\langle a \boxtimes F \ ; \ [\![B]\!]_a \rangle \xrightarrow{\varepsilon} \langle G_1 \ ; \ P_1' \rangle$
- $\langle G_1 \ ; \ P_1 \rangle = \langle G_1 \ ; \ P_1' \parallel [\![R]\!]_a \rangle$

We prove the theorem by induction over the structure of *SC* behavior.

Empty

If the behavior B is empty ($B = 0$) then

$$\langle a \boxtimes F \ ; \ [\![B]\!]_a \rangle = \langle a \boxtimes F \ ; \ \mathbf{0} \rangle$$

Nevertheless, the translation of the behavior does not perform an internal action, making this case not possible.

Skip

If the behavior B is an internal action ($B = \varepsilon; B_1$) then

$$[\![N]\!] = \langle a \boxtimes F \ ; \ \iota . [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle$$

The *NCP* state can perform a silent action using the rule `skip`, ensuring that

$$[\![N]\!] \xrightarrow{\varepsilon} \langle a \boxtimes F \ ; \ [\![B_1]\!]_a \parallel [\![T]\!]_a \rangle \ \wedge \ G_1 = a \boxtimes F \ \wedge \ P_1' = [\![B_1]\!]_a$$

Using the rule ($SC - SKIP$), the network N can be reduced as follows:

$$N \rightarrow a[B_1]_F^R = N'$$

Finally, the translation of the resultig network is

$$\left[\left[N'\right]\right] = \langle a \boxtimes F \ ; \ [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle = \left\langle G_1 \ ; \ P_1' \parallel [\![R]\!]_a \right\rangle$$

New

If the behavior B is the restriction of a topic name τ ($B = (\nu\tau)B_1$), since $\tau \notin fn(F) \cup fn(R)$, then

$$[\![N]\!] = \langle a \boxtimes F \ ; \ (\nu \ \tau : \emptyset) \left([\![B_1]\!]_a \parallel [\![R]\!]_a \right) \rangle$$

The translation of the network N perform the internal action via the rule `new`, which ensures that

$$\langle G \uplus \tau \boxdot \emptyset \ ; \ [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle \xrightarrow{\varepsilon} \langle G_2 \ ; \ P_2 \rangle$$

$$\wedge\ T_2 = G_2(\tau)$$

$$\wedge\ [\![N]\!] \xrightarrow{\varepsilon} \langle G_2 \setminus \tau \boxdot T_2\ ;\ (\nu\ \tau : T_2)\,(P_2 \parallel [\![R]\!]_a)\rangle$$

Let $N_0 = a\,[B_1]_F^R$ such that $[\![N_0]\!] = \langle G\ ;\ [\![B_1]\!]_a \parallel [\![R]\!]_a\rangle \xrightarrow{\varepsilon} \langle G_2\ ;\ P_2\rangle$, for the induction hypothesis

$$N_0 \to N_0'\ \wedge\ \left[\left[N_0'\right]\right] = \langle G_2\ ;\ P_2\rangle$$

Since $\tau \notin fn(F) \cup fn(R)$ then

$$a\,[(\nu\tau)B_1]_F^R \equiv (\nu\tau)a\,[B_1]_F^R$$

The reduction rule (*NEW*) can be applied

$$N \to (\nu\tau)N_0'\ \ \wedge\ \ \left[\left[(\nu\tau)N_0'\right]\right] = \langle G_2 \setminus \tau \boxdot T_2\ ;\ (\tau : T_2, P_2)\rangle$$

Signal emission

If the behavior B is the raising of the event $\tau_{\circledcirc}\tau'$ ($B = \mathtt{out}\langle\tau_{\circledcirc}\tau'\rangle; B_1$) then

$$[\![N]\!] = \left\langle a \boxtimes F\ ;\ \overline{\tau}\ \tau'@a.\,[\![B_1]\!]_a \parallel [\![R]\!]_a \right\rangle$$

The translation of the network N performs the internal action via the rule `emit`, which ensures that

$$[\![N]\!] \xrightarrow{\varepsilon} \left\langle a \boxtimes F\ ;\ [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel \prod_{b \in (a \boxtimes F)(\tau, a) = F\downarrow_\tau} \langle\tau_{\circledcirc}\tau'\rangle@b \right\rangle = \left\langle G_1\ ;\ P_1' \parallel [\![R]\!]_a \right\rangle$$

The reduction rule *EMIT* can be applied to the network N, obtaining

$$N \to a\,[B_1]_F^R \parallel \prod_{b \in F\downarrow_\tau} \langle\tau\rangle@\tau' b = N'$$

Finally, by the transformation rules,

$$\left[\left[N'\right]\right] = \left\langle a \boxtimes F\ ;\ [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel \prod_{b \in (a \boxtimes F)(\tau, a) = F\downarrow_\tau} \langle\tau_{\circledcirc}\tau'\rangle@b \right\rangle = \left\langle G_1\ ;\ P_1' \parallel [\![R]\!]_a \right\rangle$$

Reaction update

If the behavior B is a reaction update ($B = \mathtt{rupd}\,(R_1)\,; B_1$) then

$$[\![N]\!] = \langle a \boxtimes F\ ;\ (\iota.\,[\![R_1]\!]_a \parallel [\![B_1]\!]_a) \parallel [\![R]\!]_a\rangle$$

The translation of the network N performs the internal action via the rule `skip`, which ensures that

$$[\![N]\!] \xrightarrow{\varepsilon} \langle a \boxtimes F \;;\; [\![R_1]\!]_a \parallel [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle = \langle G_1 \;;\; P_1' \parallel [\![R]\!]_a \rangle$$

The rule $(SC-RUPD)$ can be user to reduce the network N, obtaining

$$N \to a\,[B]_F^{R|R_1} = N'$$

Finally, by the translation rules,

$$[[N']] = \langle G_1 \;;\; P_1' \parallel [\![R]\!]_a \rangle$$

Flow update

If the behavior B is a flow update ($B = \texttt{fupd}\,(F_1)\,;B_1$) then

$$[\![N]\!] = \langle a \boxtimes F \;;\; \texttt{fupd}\,(F_1)\,@a.\,[\![B_1]\!]_a \parallel [\![R]\!]_a \rangle$$

The translation of the network N performs the internal action via the rule `fupd`, which ensures that

$$[\![N]\!] \xrightarrow{\varepsilon} \langle a \boxtimes F \uplus a \boxtimes F_1 = a \boxtimes F|F_1 \;;\; [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle = \langle G_1 \;;\; P_1' \parallel [\![R]\!]_a \rangle$$

The rule $(SC-FUPD)$ can be user to reduce the network N, obtaining

$$N \to a\,[B]_{F|F_1}^{R} = N'$$

Finally, by the translation rules,

$$[[N']] = \langle G_1 \;;\; P_1' \parallel [\![R]\!]_a \rangle$$

Parallel composition

If the behavior B is a parallel composition ($B = B_1 \mid B_2$) then

$$[\![N]\!] = \langle a \boxtimes F \;;\; [\![B_1]\!]_a \parallel [\![B_2]\!]_a \parallel [\![R]\!]_a \rangle$$

Since the translation of the behaviors B_1 and B_2 cannot perform input or output actions, their composition cannot communicate internally. The translation of the network N can performs the internal action only via the rule `par`, which ensures that

$$[\![N]\!] \xrightarrow{\varepsilon} \langle G_1 \;;\; P_2 \parallel [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle = \langle G_1 \;;\; P_1' \parallel [\![R]\!]_a \rangle$$

$$\wedge\ \langle a \boxtimes F\ ;\ [\![B_1]\!]_a \rangle \xrightarrow{\varepsilon} \langle G_1\ ;\ P_2 \rangle$$

Let $\left[\left[a\,[B_1]_F^R\right]\right] = \langle a \boxtimes F\ ;\ [\![B_1]\!]_a \parallel [\![R]\!]_a \rangle \xrightarrow{\varepsilon} \langle G_1\ ;\ P_2 \parallel [\![R]\!]_a \rangle$ the parallel branch that perform the internal action, then by induction hypothesis

$$a\,[B_2]_F^R \rightarrow N_2' \ \wedge\ \left[\left[N_2'\right]\right] = \langle G_1\ ;\ P_2 \parallel [\![R]\!]_a \rangle$$

We highlight that N_1' is of the form

$$N_2' = (\nu\tau)\left(a\left[B_2'\right]_{F|F'}^{R|R'} \parallel N_\pi\right)$$

Where F' and R' are the flows and reactions, possibly empty, added by the behavior B_2 and N_π is the set, possibly empty, of envelope spawned. Translating to *NCP* state this kind of processes we obtain

$$\left[\left[N_2'\right]\right] = \left\langle G_1\ ;\ (\nu\,\tau : T)\left(\left[\left[B_2'\right]\right]_a \parallel [\![R]\!]_a \parallel \left[\left[R'\right]\right]_a\right)\right\rangle$$

Now, we use the *SC* reduction rule (*PAR*) to verify that the network N can perform an action

$$N = a\,[B_1 \mid B_2]_F^R \rightarrow (\nu\tau)\left(a\left[B_1 \mid B_2'\right]_{F|F'}^{R|R'} \parallel N_\pi\right) = N'$$

We translate the network N' to *NCP* state exploiting the information known about the translation of the network N_2'

$$\left[\left[N'\right]\right] = \left\langle G_1\ ;\ (\nu\,\tau : T)\left([\![B_1]\!]_a \parallel \left[\left[B_2'\right]\right]_a \parallel [\![R]\!]_a \parallel \left[\left[R'\right]\right]_a\right)\right\rangle$$

We can move the translations of the reaction R and of the behavior B_1 out of the scope of the restriction, since we know that $\tau \notin fn(R) \cup fn(B_1)$

$$\left[\left[N'\right]\right] = \langle G_1\ ;\ [\![B_1]\!]_a \parallel P_2 \parallel [\![R]\!]_a \rangle = \left\langle G_1\ ;\ P_1' \parallel [\![R]\!]_a \right\rangle$$

B.2 Lemma 3

Lemma 3 *Let N and N' be* SC *networks. It holds that if $N \rightarrow N'$ then $[\![N]\!] \xrightarrow{\varepsilon} (G,P)$ and $(G,P) \sim [\![N']\!]$*

We prove the theorem by induction over reduction rules of *SC* networks

SKIP

If the rule *SKIP* has been applied, then

$$N = a\left[\varepsilon;B \mid B'\right]_F^R \rightarrow a\left[B \mid B'\right]_F^R = N'$$

The translation of the network $[\![N]\!]$ can perform the empty action by using the rule (`skip`):

$$[\![N]\!] = \left\langle a \boxtimes F \;;\; \imath.\, [\![B]\!]_a \parallel \left[\left[B'\right]\right]_a \parallel [\![R]\!]_a \right\rangle \xrightarrow{\varepsilon} \left\langle a \boxtimes F \;;\; [\![B]\!]_a \parallel \left[\left[B'\right]\right]_a \parallel [\![R]\!]_a \right\rangle = \left[\left[N'\right]\right]$$

RUPD

If the rule $SC-RUPD$ has been applied, then

$$N = a\left[\mathtt{rupd}\,(R_1);B_1 \mid B\right]_F^R \rightarrow a\left[B_1 \mid B\right]_F^{R|R_1} = N'$$

The translation of the network $[\![N]\!]$ can perform an internal action by applying the rule `skip`:

$$[\![N]\!] = \langle a \boxtimes F \;;\; \imath.([\![R_1]\!]_a \parallel [\![B_1]\!]_a) \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle$$

$$\langle a \boxtimes F \;;\; \imath.([\![R_1]\!]_a \parallel [\![B_1]\!]_a) \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle \xrightarrow{\varepsilon} \langle a \boxtimes F \;;\; [\![R_1]\!]_a \parallel [\![B_1]\!]_a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle$$

$$\langle a \boxtimes F \;;\; [\![R_1]\!]_a \parallel [\![B_1]\!]_a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle = \left[\left[N'\right]\right]$$

Fupd

If the rule *FUPD* has been applied, then

$$N = a\left[\mathtt{fupd}\,(F_1);B_1 \mid B\right]_F^R \rightarrow a\left[B_1 \mid B\right]_{F|F_1}^R$$

The translation of the network $[\![N]\!]$ can perform an internal action by using the rule `fupd`:

$$[\![N]\!] = \langle a \boxtimes F \;;\; \mathtt{fupd}\,(F_1)\,@a.\, [\![B_1]\!]_a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle$$

$$\xrightarrow{\varepsilon}$$

$$\langle a \boxtimes F \uplus a \boxtimes F_1 \;;\; [\![B_1]\!]_a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle = \langle a \boxtimes F|F_1 \;;\; [\![B_1]\!]_a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \rangle = \left[\left[N'\right]\right]$$

Check

If the rule (*Check*) has been applied, then

$$N = \langle \tau \circledcirc \tau' \rangle @ a \parallel a\,[B_1]_F^{R|\tau \circledcirc \tau' \gg B_2} \rightarrow a\,[B_1 \mid B_2]_F^R = N'$$

The translation of the network can perform an internal action by using the rule (`com`):

$$[\![N]\!] = \left\langle a \boxtimes F \;;\; \langle \tau \circledcirc \tau' \rangle @ a \parallel [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel \tau\,\tau' @ a.\,[\![B_2]\!]_a \right\rangle$$

$$\left\langle a \boxtimes F \;;\; \langle \tau \circledcirc \tau' \rangle @ a \parallel [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel \tau\,\tau' @ a.\,[\![B_2]\!]_a \right\rangle \xrightarrow{\varepsilon} \langle a \boxtimes F \;;\; [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel [\![B_2]\!]_a \rangle$$
$$\langle a \boxtimes F \;;\; [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel [\![B_2]\!]_a \rangle = \left[\!\left[N'\right]\!\right]$$

Lambda

If the rule (*LAMBDA*) has been applied, then

$$N = \langle \tau \circledcirc \tau' \rangle @ a \parallel a\,[B_1]_F^{R|\tau\,\lambda\,\tau'' \gg B_2} \rightarrow a\left[B_1 \mid \{\tau'/\tau''\} B_2\right]_F^{R|\tau\,\lambda\,\tau'' \gg B_2} = N'$$

The translation of the network can perform an internal action by using the rule `com`:

$$[\![N]\!] = \left\langle a \boxtimes F \;;\; \langle \tau \circledcirc \tau' \rangle @ a \parallel [\![B_1]\!]_a \parallel [\![R]\!]_a \parallel \tau\left(\tau''\right) @ a.\,[\![B_2]\!]_a \right\rangle$$

$$\xrightarrow{\varepsilon}$$

$$a \boxtimes F\,[\![B_1]\!]_a \parallel \{\tau'/\tau''\}\,[\![B_2]\!]_a \parallel [\![R]\!]_a \parallel \tau\left(\tau''\right) @ a.\,[\![B_2]\!]_a = \left[\!\left[N'\right]\!\right]$$

NEW

If the rule (*NEW*) has been applied then:

$$N = (\nu n) N_1 \rightarrow (\nu n) N_2 = N' \;\wedge\; N_1 \rightarrow N_2$$

By exploiting the induction hypothesis, we ensure that:

$$[\![N_1]\!] \xrightarrow{\varepsilon} \langle G_1 \;;\; P_1 \rangle \sim [\![N_2]\!]$$

If $n \in \mathcal{T}$, let $\langle G_0 \;;\; P_0 \rangle = [\![N_1]\!]$ and $T_0 = G_0(n)$ then $[\![N]\!] = \langle G_0 \setminus n \boxdot T_0 \;;\; (\nu\, n : T_0)\, P_0 \rangle$. The state $[\![N]\!]$ can perform a silent action by using the transition rule `new`:

$$[\![N]\!] \xrightarrow{\varepsilon} \langle G_1 \setminus n \boxdot T_1 \;;\; (\nu\, n : T_1)\, P_1 \rangle$$

Where $T_1 = G_1(n)$. The proof is completed using the Theorem 4. The same proof strategy can be used if $n \in \mathcal{A}$.

B.3 Lemma 4

Lemma 4 *Let $N_1 \parallel N_2$ a well formed network, $[\![N_1]\!] = \langle G_1 \; ; \; N_1 \rangle$ and $[\![N_2]\!] \langle G_2 \; ; \; P_2 \rangle$ the translation of the two sub-networks. If $\langle G_1 \; ; \; P_1 \rangle \xrightarrow{\varepsilon} \langle G_1' \; ; \; P_1' \rangle$ then $\langle G_1 \uplus G_2 \; ; \; P_1 \rangle \xrightarrow{\varepsilon} \langle G_1' \uplus G_2 \; ; \; P_1' \rangle$*

The lemma can be proved exploiting the Theorem 1 , showing that the subjects of the translations of the two sub-networks are always disjoint. This can be done trivially, by using the notion of well formed *SC* network, which requires that the component names of N_1 and N_2 are disjoint.

B.4 Lemma 5

Lemma 5 *Let N and N' be* SC *networks. It holds that if $[\![N]\!] \xrightarrow{\varepsilon} (G,P)$ then $N \rightarrow N'$ and $(G,P) \sim [\![N']\!]$*

We prove the lemma by induction over the structure of the *SC* network N. If the N is the empty network ($\emptyset$) or an envelope ($\langle \tau_{@}\tau' \rangle @a$), its translation $[\![N]\!]$ does not perform an empty action, then we check only the other primitives.

Topic Restriction

If the network is a topic restriction ($N = (\nu\tau)N_1$), let $[\![N_1]\!] = \langle G_1 \; ; \; P_1 \rangle$ be the translation of the network N_1 and $T = G_1(\tau)$ the projection of the topology G_1 respect to the topic τ, then the translation of the network N is

$$[\![N]\!] = \langle G_1 \setminus \tau \boxdot T \; ; \; (\nu \, \tau : T) P_1 \rangle$$

The state $[\![N]\!]$ can perform a silent action only by using the rule $(NCP - \texttt{new}$. The rule ensures that:

$$[\![N]\!] \xrightarrow{\varepsilon} \langle G \; ; \; P \rangle \; \wedge \; \langle G_1 \; ; \; P_1 \rangle \xrightarrow{\varepsilon} \langle G_1' \; ; \; P_1' \rangle \; \wedge \; T' = G_1'(\tau)$$

then

$$\langle G \; ; \; P \rangle = \langle G_1' \setminus \tau \boxdot T' \; ; \; (\nu \, \tau : T') P_1' \rangle$$

The induction hypothesis ensures that if

$$\langle G_1 \; ; \; P_1 \rangle = [\![N_1]\!] \xrightarrow{\varepsilon} \langle G_1' \; ; \; P_1' \rangle$$

then

$$N_1 \rightarrow N_1' \; \wedge \; [[N_1']] = \langle G_2 \; ; \; P_2 \rangle \sim \langle G_1' \; ; \; P_1' \rangle$$

The network N can be then reduced by using the rule $SC-NEW$, which ensures that:

$$(\nu\tau)N_1 \rightarrow (\nu\tau)N_1'$$

Let be $T_2 = G_2(\tau)$, the translation of the resulting network is

$$[\![(\nu\tau)N_1']\!] = \langle G_2 \setminus \tau \boxdot T_2 \; ; \; (\nu\, \tau : T_2)\, P_2 \rangle$$

which is bisimilar to $\langle G \; ; \; P \rangle$ by exploiting the Theorem 4.

The same proof strategy can be exploited also for the component name restriction.

Component

If the network is simply a component $a\,[B]_F^R$ then the Theorem 2 can be directly exploited.

Parallel composition, rule PAR

If the network N is the parallel composition of two sub-networks N_1 and N_2 and their translation are $[\![N_1]\!] = \langle G_1 \; ; \; P_1 \rangle$ and $\langle N_2 \; ; \; P_2 \rangle$, then the translation of N is

$$\langle G_1 \uplus G_2 \; ; \; P_1 \parallel P_2 \rangle$$

We check one of the three possible *NCP* transition rules ($\texttt{par}, \texttt{com}, \texttt{close}$) that permit to the state to perform a silent action. If the rule `par` has been applied, then

$$\langle G_1 \uplus G_2 \; ; \; P_1 \parallel P_2 \rangle \xrightarrow{\varepsilon} \langle G \; ; \; P_1' \parallel P_2 \rangle = \langle G \; ; \; P \rangle \; \wedge \; \langle G_1 \uplus G_2 \; ; \; P_1 \rangle \xrightarrow{\varepsilon} \langle G \; ; \; P_1' \rangle$$

Since $\langle G_1 \; ; \; P_1 \rangle \perp G_2$ then the Theorem 1 can be applied, ensuring that

$$[\![N_1]\!] = \langle G_1 \; ; \; P_1 \rangle \xrightarrow{\varepsilon} \langle G_1' \; ; \; P_1' \rangle \; \wedge \; G = G_1' \uplus G_2$$

The induction hypothesis ensure that

$$N_1 \rightarrow N_1' \; \wedge \; [\![N_1']\!] \sim \langle G_1' \; ; \; P_1' \rangle$$

Then, the starting parallel composition of the two sub-networks can be reduced using the rule $SP-PAR$:

$$N_1 \parallel N_2 \rightarrow N_1' \parallel N_2$$

Finally we can use the Theorem 5 to guarantee that

$$[\![N_1' \parallel N_2]\!] \sim \langle G_1' \uplus G_2 \; ; \; P_1' \parallel P_2 \rangle = \langle G \; ; \; P \rangle$$

Parallel composition, rule COM

If the network N is the parallel composition of two sub-networks N_1 and N_2 and their translation are $[\![N_1]\!] = \langle G_1 \; ; \; P_1 \rangle$ and $\langle N_2 \; ; \; P_2 \rangle$, then the translation of N is

$$\langle G_1 \uplus G_2 \; ; \; P_1 \parallel P_2 \rangle$$

We check one of the three possible *NCP* transition rules (`par`, `com`, `close`) that permit to the state to perform an internal action. If the rule `com` has been applied and by Theorem 4, then $\exists a, \tau, \tau'$ such that

$$[\![N_1]\!] \xrightarrow{\langle \tau \circledcirc \tau' \rangle @ a} \langle G_1' \; ; \; P_1' \rangle \wedge [\![N_2]\!] \xrightarrow{\tau \; \tau' @ a} \langle G_2' \; ; \; P_2' \rangle \wedge \langle G \; ; \; P \rangle = \langle G_1' \uplus G_2' \; ; \; P_1' \parallel P_2' \rangle$$

It is trivial to prove that one of the two sub-networks must contains the envelope $\langle \tau \circledcirc \tau' \rangle @ a$ and the other one the component a able to consume the envelope. The component a must have a lambda reaction for topic τ or a check reaction for signals $\tau \circledcirc \tau'$. We prove the theorem only for the first case, since the second one can be proved involves the same strategy. This statement is formalized by

$$N_1 = \langle \tau \circledcirc \tau' \rangle @ a \parallel N_a \wedge N_2 = a \, [B]_F^{R | \tau \, \lambda \, \tau'' \gg B_1} \parallel N_b$$

Let $[\![N_a]\!] = \langle G_a \; ; \; P_a \rangle$ and $[\![N_b]\!] = \langle G_b \; ; \; P_b \rangle$ the translations of the two networks N_a and N_b, the translation of the starting network N can perform an internal action using the rule (*COM*) as follows:

$$[\![N]\!] = \left\langle G_a \uplus G_b \uplus a \boxtimes F \; ; \; P_a \parallel P_b \parallel \langle \tau \circledcirc \tau' \rangle @ a \parallel [\![B]\!]_a \parallel [\![R]\!]_a \parallel \tau \left(\tau'' \right) @ a . \, [\![B_1]\!]_a \right\rangle$$

$$\xrightarrow{\varepsilon}$$

$$\left\langle G_a \uplus G_b \uplus a \boxtimes F \; ; \; P_a \parallel P_b \parallel [\![B]\!]_a \parallel [\![R]\!]_a \parallel \tau \left(\tau'' \right) @ a . \, [\![B_1]\!]_a \parallel \{ \tau' / \tau'' \} \, [\![B_1]\!]_a \right\rangle = \langle G \; ; \; P \rangle$$

The starting *SC* network can be reduced by the rule $SC - LAMBDA$ obtaining:

$$N \rightarrow N_a \parallel N_b \parallel a \left[B \mid \{ \tau' / \tau'' \} B_1 \right]_F^{R | \tau \, \lambda \, \tau'' \gg B_1} = N'$$

Finally, it is trivial to verify that $N' = \langle G \; ; \; P \rangle$.

The same proof strategy can be exploited for the transitions involving the rule `close`.

B.5 Proof of Theorem 6

Let N and N' be *SC* networks. It holds that $N \rightarrow N'$ if and only if $[\![N]\!] \xrightarrow{\varepsilon} (G, P)$ and $(G, P) \sim [\![N']\!]$

The theorem is directly implied by the two Lemmas 3 and 5.

Appendix C

Proof of theorems in Section 4.4

C.1 Proof of Theorem 8

Let be *TC* and *DelegatedTC* such that

$$TC = (\nu ok, ex)\left(\begin{array}{l} f\ \lambda\ s \gg \left(\begin{array}{l} \mathtt{rupd}\left(ok \circledcirc s \gg \begin{array}{l}\mathtt{rupd}\,(r \circledcirc s \gg [\![CompA]\!]_c) \\ \mid \mathtt{out}\langle f \circledcirc s\rangle\end{array}\right) \\ \mid\ \mathtt{rupd}\,(ex \circledcirc s \gg \mathtt{out}\langle r \circledcirc s\rangle) \\ \mid\ [\![A]\!] \end{array}\right) \\ a\,[0]_{\{ok \rightsquigarrow a, ex \rightsquigarrow a, f \rightsquigarrow \vec{c}_1, r \rightsquigarrow \vec{c}_2\}} \end{array}\right)$$

$$DelegatedTC = (\nu b, ok, ex)\left(\begin{array}{l} f\ \lambda\ s \gg \left(\begin{array}{l} \mathtt{rupd}\left(ok \circledcirc s \gg \begin{array}{l}\mathtt{rupd}\,(r \circledcirc s \gg \mathtt{out}\langle r \circledcirc s\rangle) \\ \mid \mathtt{out}\langle f \circledcirc s\rangle\end{array}\right) \\ \mid\ [\![A]\!] \end{array}\right) \\ a\,[0]_{\{ok \rightsquigarrow a, ex \rightsquigarrow b, f \rightsquigarrow \vec{c}_1, r \rightsquigarrow b\}} \\ \parallel\ b\,[0]^{R_b}_{\{r \rightsquigarrow \vec{c}_2\}} \end{array}\right)$$

$$where\ R_b = ex\ \lambda\ s \gg \mathtt{out}\langle r \circledcirc s\rangle \mid\ r \lambda s \gg [\![CompA]\!]_c$$

then $[\![TC]\!] \approx [\![DelegatedTC]\!]$

The proof is constructive. We construct a relation $\mathcal{B}$ including the pair

$$([\![TC]\!], [\![DelegatedTC]\!])$$

and, then, we prove that $\mathcal{B}$ is a *NCP* weak-bisimulation. The crucial part of the proof consists of the construction of the relation $\mathcal{B}$. Notice that our notion

of bisimilarity takes into account asynchrony, hence, $\mathcal{B}$ will contains bags of envelopes because of the transition rule `async`.

Before constructing the relation $\mathcal{B}$, we state again which are the hypothesis we did on $[\![A]\!]$ and $[\![CompA]\!]$ (see Section 4.3). Without loss of generality, we can assume that for any component name n and linkage E:

- if $A \mapsto \boxdot$ then $[\langle(\emptyset,E)\ ;\ [\![[\![A]\!]]\!]_n\rangle \approx \langle(\emptyset,E)\ ;\ [\![\mathtt{out}\langle ok \circledcirc s\rangle]\!]_n\rangle$
- if $A \mapsto \boxtimes$ then $\langle(\emptyset,E)\ ;\ [\![[\![A]\!]]\!]_n\rangle \approx \langle(\emptyset,E)\ ;\ [\![\mathtt{out}\langle ex \circledcirc s\rangle]\!]_n\rangle$
- $\left\langle(\emptyset,E)\ ;\ [\![[\![CompA]\!]_c]\!]_n\right\rangle \approx \langle(\emptyset,E)\ ;\ [\![\mathtt{out}\langle r \circledcirc s\rangle]\!]_n\rangle$

Hence, hereafter we work up-to this assumptions. We start translating the two *SC* networks into *NCP* policies. By definition (Section 4.2), the mapping of the transactional component becomes:

$$
\begin{array}{ll}
[\![TC]\!] & = \langle a \boxtimes F\ ;\ (\nu\ (n,))P_1\rangle \\
where\ F & = f \rightsquigarrow \vec{c_1}, r \rightsquigarrow \vec{c_2} \\
and\ n = & ok : (a,a) ex : (a,a) \\
and\ P_1 & = f\,(s)\ @a.(P_2(s) \parallel P_3(s) \parallel [\![[\![A]\!]]\!]_a) \\
and\ P_2(s) & = ok\ s@a.(P_4 \parallel \overline{f}\ s@a) \\
and\ P_3(s) & = ex\ s@a.\overline{r}\ s@a \\
and\ P_4(s) & = r\ s@a.\ [\![[\![CompA]\!]_c]\!]_a
\end{array}
$$

Similarly, we have:

$$
\begin{array}{ll}
[\![DelegatedTC]\!] & = \langle a \boxtimes F'\ ;\ (\nu\ n_1)\,(Q_1 \parallel Q_2)\rangle \\
where\ F' & = f \rightsquigarrow \vec{c_1} \\
and\ n_1 = & b : (b,r,\vec{c_2}), (a,r,b), (ok : (a,a), ex : (a,b)) \\
and\ Q_1 & = f\,(s)\ @a.Q_3(s) \parallel [\![[\![A(s)]\!]]\!]_a \\
and\ Q_2 & = Q_4 \parallel Q_5 \\
and\ Q_3(s) & = ok\ s@a.(Q_6 \parallel \overline{f}\ s@a) \\
and\ Q_4 & = ex\,(s)\ @b.\overline{r}\ s@b \\
and\ Q_5 & = r\,(s)\ @b.\ [\![[\![CompA(s)]\!]_c]\!]_a \\
and\ Q_6(s) & = r\ s@a.\overline{r}\ s@a
\end{array}
$$

Notice that both $[\![TC]\!]$ and $[\![DelegatedTC]\!]$ can perform a transition only reacting to events for the topic f. Intuitively, this corresponds to saying that the forwarding event f is the only mean to activate their behavior. Technically, this constraint results in expressing the other topics under the scope of a restriction. Also the subject of policies Q_4 and Q_5 (i.e. the component b) is restricted.

However, because of asynchrony, $[\![TC]\!]$ and $[\![DelegatedTC]\!]$ can perform an `async` transition, thus buffering a new envelope. We have several cases.

Assume that the buffered envelope is of the form:

$$\langle f \circledcirc s' \rangle @ a$$

In this case, it is easy to see that both policies can start their behavior.

Notice that one can repeatedly apply the `async` rule, yielding to a multiset of envelopes. To deal with this possibility, we introduce some notational machinery. In particular, we equip the multiset of envelopes with a total order. To this purpose we exploit the index-set $\mathcal{N}_f$ of integers (possibly empty) associating an integer to each envelope f consumed be the transactional component. Intuitively, the cardinality of $\mathcal{N}_f$ identifies the number of execution of the activity A by the transactional component. Also:

- If $i \in \mathcal{N}_f$ then $\mathcal{F}(i)$ is the session of the envelope
- If $i \in \mathcal{N}_f$ then $\mathcal{A}(i)$ is the result status of the activity A

Similarly, we introduce a total order $\mathcal{N}_r$, whose cardinality states the number of compensations executed by the transactional component. Without loss of generality we can assume that

$$\mathcal{N}_r \subseteq \{i \mid i \in \mathcal{N}_f \ \wedge \ \mathcal{A}(i) = \boxdot\}$$

because the transactional component reacts to an event r only if the component has previously received:

- the corresponding forward event
- the event *ok* from the main activity A

In the following, we will use the following notations

- $\mathcal{OK} = \{i \mid i \in \mathcal{N}_f \ \wedge \ \mathcal{A}(i) = \boxdot\}$
- $\mathcal{EX} = \{i \mid i \in \mathcal{N}_f \ \wedge \ \mathcal{A}(i) = \boxtimes\}$
- $\mathcal{F}(\mathcal{N}) = \{s \mid \exists i \in N \ \wedge \ \mathcal{F}(i) = s\}$

Our proof strategy consists of characterizing the possible execution of the transactional component.

At first, the transactional component is the *NCP* state:

$$\langle a \boxtimes F \ ; \ (\nu \ n_1) P_1 \rangle$$

Now, if $\mathcal{N}_f$ forward-flow instances have been invoked then the transactional component becomes

$$\left\langle a \boxtimes F \;;\; (\nu\, n) \left(\begin{array}{l} P_1 \parallel P_F(\mathcal{F}(O\mathcal{K})) \parallel P_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K})} P_4(s) \end{array} \right) \right\rangle$$
$$P_F(\vec{s}) = \prod_{s \in \vec{s}} \prod_{c \in \vec{c}_1} \langle f \circledcirc s \rangle @ c \parallel \prod_{s \in \vec{s}} P_3(s)$$
$$P_R(\vec{s}) = \prod_{s \in \vec{s}} \prod_{c \in \vec{c}_2} \langle r \circledcirc s \rangle @ c \parallel \prod_{s \in \vec{s}} P_2(s)$$

For each executed forward request ($i \in \mathcal{N}_f$), the transactional component can produce P_F or P_R, depending on the result of the internal activity A.

- If the activity A successes (all sessions in the multiset $\mathcal{F}(O\mathcal{K})$) the component spawns the envelopes ($\prod_{c \in \vec{c}_1} \langle f \circledcirc s \rangle @ c$) to propagate the forward-flow and installs its check reaction ($P_4(s)$) to activate its compensation handling possible failures. Notice that the local check reactions for the topic *ex* are not consumed ($P_3(s)$), because the main activity A sent an envelope containing the topic *ok*.

- If the main activity A fails (all sessions in the multiset $\mathcal{F}(\mathcal{EX})$), the component spawns the envelopes ($\prod_{c \in \vec{c}_2} \langle r \circledcirc s \rangle @ c$) to start the backward-flow and the check reactions for the compensation are not installed. Notice that in this case, the pending reactions wait for *ok* events ($P_2(s)$), because the main activity A has notified the topic *ex*.

This characterizes the state of the transactional component when $i \in \mathcal{N}_f$ have been consumed. Now, we characterize the state of the transactional component after executing the compensations identified by $\mathcal{N}_r$.

We assumed that $\mathcal{N}_r \subseteq \{i \mid i \in \mathcal{N}_f \wedge \mathcal{A}(i) = \boxdot\}$. The state of the transactional component becomes:

$$\langle a \boxtimes F \;;\; (\nu\, n) P \rangle$$
$$P = \left(\begin{array}{l} P_1 \parallel P_F(\mathcal{F}(O\mathcal{K})) \parallel P_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r)} P_4(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c}_2} \langle r \circledcirc s \rangle @ c \end{array} \right)$$

The number of instances of pending compensations (i.e. $P_4(s)$) has decreased in term of $\mathcal{N}_r$. Since $\mathcal{N}_r \subseteq O\mathcal{K}$, we can always match the right reaction to perform the compensation request. Moreover, the envelopes (required to propagate the backward-flow) are spawned $\prod_{c \in \vec{c}_2} \langle r \circledcirc s \rangle @ c$.

Similarly, we can characterize the states of the delegated transactional com-

ponent. We have:

$$\begin{array}{l}\langle a \boxtimes F_1 \ ; \ (\nu \ n_1)\, Q \rangle \\ Q = \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{E}\mathcal{X})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_f)} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_f)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \end{array} \right) \\ Q_F(\vec{s}) = \prod_{s \in \vec{s}} \prod_{c \in \vec{c_1}} \langle f \circledcirc s \rangle @ c \\ Q_R(\vec{s}) = \prod_{s \in \vec{s}} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel \prod_{s \in \vec{s}} Q_3(s) \end{array}$$

To prove the theorem, we show that the relation $\mathcal{B}$ defined below is a *NCP* weak-bisimulation:

$$\mathcal{B} = \left\{ \left(\left\langle a \boxtimes F \ ; \ (\nu \ n)\, P \parallel \prod_{i \in S} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle, \left\langle a \boxtimes F' \ ; \ (\nu \ n_1)\, Q \parallel \prod_{i \in S} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle \right) \right\}$$

where P and Q are two policies having the structure described above. Notice that we compose the two policies with the same envelopes ($\prod_{i \in S} \langle \tau \circledcirc \tau' \rangle @ c$). It is straightforward to verify that $([\![TC]\!], [\![DelegatedTC]\!]) \in \mathcal{B}$. It is sufficient to fix $\mathcal{N}_f = \emptyset$ and $S = \emptyset$.

We check only that the *NCP* state $\langle a \boxtimes F' \ ; \ (\nu \ n_1)\, Q \parallel \prod_S \langle \tau \circledcirc \tau' \rangle @ c \rangle$ mimics the transitions performed by $\langle a \boxtimes F \ ; \ (\nu \ n)\, P \parallel \prod_S \langle \tau \circledcirc \tau' \rangle @ c \rangle$. In fact, verifying that the *NCP* state $\langle a \boxtimes F \ ; \ (\nu \ n)\, P \parallel \prod_S \langle \tau \circledcirc \tau' \rangle @ c \rangle$ mimics the transitions performed by the *NCP* state $\langle a \boxtimes F' \ ; \ (\nu \ n)\, Q \parallel \prod_S \langle \tau \circledcirc \tau' \rangle @ c \rangle$ can be done by symmetric reasoning.

Notice that we have only four type of transitions, depending on the rule applied: namely `notify` , `async` , `com` and `close` .

In the case of `notify` and `async` the proof is easy. The `notify` transitions performed by the contained envelopes are matched, because the two states contains the same envelopes. All `async` transitions appends the same envelope to the two states.

The other transitions are the synchronizations between one of the pending envelopes ($\prod_S \langle \tau \circledcirc \tau' \rangle @ c$) and P. These communications can occur via the transition rules `com` and `close`. We report the proof only the transition `com`, because the `close` case is similar.

Since the topics *ok* and *ex* are restricted, the communications can occur only for envelopes having topics f or r. We consider these two cases separately.

Case 1) events f

If the envelopes contain an event f, the transition rule `com` can be applied to the envelope and the policy P_1 contained in P. Since P_1 is a lambda reaction, we

obtain the transition:

$$S_1 = \langle a \boxtimes F \;;\; (\nu\, n)\, P \parallel \textstyle\prod_S \langle \tau_{\circledcirc} \tau' \rangle @ c \rangle$$
$$\stackrel{\varepsilon}{\Longrightarrow}$$
$$\langle a \boxtimes F \;;\; (\nu\, n)\, (P \parallel P_2(s) \parallel P_3(s) \parallel [\![[\![A(s)]\!]]\!]_a) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \rangle = S_2$$

Two cases are possible. The activity A either successes of fails. We prove them separately.

We know that if the activity successes, then it raises the event ok. Then

$$S_2 \stackrel{\varepsilon}{\Longrightarrow} \langle a \boxtimes F \;;\; (\nu\, n)\, (P \parallel P_2(s) \parallel P_3(s) \parallel \langle ok_{\circledcirc} s \rangle @ a) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \rangle = S_3$$

This envelope is not handled by the bisimulation game, since the topic ok is restricted. However, it can react with the policy $P_2(s)$, activating its behavior

$$S_3 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F \;;\; (\nu\, n)\, \left(P \parallel P_4(s) \parallel \overline{f}\; s @ a \parallel P_3(s)\right) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \right\rangle = S_4$$

Now, the transactional component raises the event f to propagate the forward-flow. The resulting envelopes are delivered to $\vec{c_1}$ by exploiting the flow F.

$$S_4 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F \;;\; (\nu\, n) \left(P \parallel P_4(s) \parallel \prod_{c \in \vec{c_1}} \langle f_{\circledcirc} s \rangle @ c \parallel P_3(s) \right) \parallel \prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \right\rangle = S_5$$

We have to verify that the delegated transactional component can perform silent actions up to the emission of the same envelopes for $\vec{c_1}$. Initially, the pending event f can react only to the lambda reaction of a, because all other reactions wait for restricted topics or are performed by the restricted component b.

$$R_1 = \langle a \boxtimes F' \;;\; (\nu\, n_1)\, (Q_1 \parallel Q_2) \parallel \textstyle\prod_S \langle \tau_{\circledcirc} \tau' \rangle @ c \rangle$$
$$\stackrel{\varepsilon}{\Longrightarrow}$$
$$\langle a \boxtimes F' \;;\; (\nu\, n_1)\, (Q_1 \parallel Q_2 \parallel Q_3(s) \parallel [\![[\![A]\!]]\!]_a) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \rangle = R_2$$

Since the main activity terminates, the event ok is raised. Notice that ok is delivered to a itself in accordance with the flow F'.

$$R_2 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F' \;;\; (\nu\, n_1)\, \left(Q_1 \parallel Q_2 \parallel ok\; s @ a.(Q_6(s) \parallel \overline{f}\; s @ a) \parallel \langle f_{\circledcirc} s \rangle @ a\right) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \right\rangle =$$

The resulting envelope can activate the check reaction of a

$$R_3 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F' \;;\; (\nu\, n_1)\, \left(Q_1 \parallel Q_2 \parallel Q_6(s) \parallel \overline{f}\; s @ a\right) \parallel \textstyle\prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @ c \right\rangle \;\; = R_4$$

Now, the delegated transactional component raises the event f to propagate the forward-flow. The resulting envelopes are delivered to $\vec{c_1}$ by exploiting the flow F'.

$$R_4 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F' \,;\, (\nu\, n_1) \left(Q_1 \parallel Q_2 \parallel \prod_{c \in \vec{c_1}} \langle f \circledcirc s \rangle @ c \parallel Q_6(s) \right) \parallel \prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R_5$$

Notice that by construction $(S_5, R_5) \in \mathcal{B}$.

We exploit the same reason in the case of failure of the main activity A. The main activity raises the event ex that is directly delivered to a

$$S_2 \stackrel{\varepsilon}{\Longrightarrow} \langle a \boxtimes F \,;\, (\nu\, n)\,(P \parallel P_2(s) \parallel P_3(s) \parallel \langle ex \circledcirc s \rangle @ a) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \rangle = S_3$$

The envelope can react with the policy $P_3(s)$, activating the corresponding behavior

$$S_3 \stackrel{\varepsilon}{\Longrightarrow} \langle a \boxtimes F \,;\, (\nu\, n)\,(P \parallel \overline{r}\, s @ a \parallel P_2(s)) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \rangle = S_4$$

The transactional component raises the event r to start the backward-flow. In accordance with the flow F, the envelopes are delivered to $\vec{c_2}$

$$S_4 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F \,;\, (\nu\, n) \left(P \parallel \textstyle\prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel P_2(s) \right) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = S_5$$

We verify that the delegated transactional component can perform silent actions up to the emission of the same envelopes for $\vec{c_2}$. Since the main activity A correctly fails, the event ex is issued and, according to the flow F', delivered to b.

$$R_2 \stackrel{\varepsilon}{\Longrightarrow} \langle a \boxtimes F' \,;\, (\nu\, n_1)\,(Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_3(s) \parallel \langle ex \circledcirc s \rangle @ b) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \rangle = R_3$$

Now, the envelope can activate the check reaction declared by b

$$R_3 \stackrel{\varepsilon}{\Longrightarrow} \langle a \boxtimes F' \,;\, (\nu\, n_1)\,(Q_1 \parallel \overline{r}\, s @ b \parallel Q_2) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \rangle = R_4$$

Finally, the component b raises the event r to start the backward-flow. According to the flow F', the resulting envelopes are delivered to $\vec{c_2}$

$$R_4 \stackrel{\varepsilon}{\Longrightarrow} \left\langle a \boxtimes F' \,;\, (\nu\, n_1) \left(Q_1 \parallel \textstyle\prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel Q_2 \parallel Q_3(s) \right) \parallel \textstyle\prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R_5$$

Also in this case $(S_4, R_5) \in \mathcal{B}$ by construction.

Case 2) events r

If the envelopes contain an event of the form $\langle r_{\circledcirc} s' \rangle @a$, the transition rule `com` can be applied to the envelope and the policy $P_4(s')$ contained in P. However, since P_4 is a check reaction for a specific session, to enable the communication the following property must be satisfied:

$$s' \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_t)$$

Intuitively, this corresponds to saying that

- an event f with the same session have been handled by the transactional component
- the main activity A successfully terminates
- the backward-flow (for this instance) was not already invoked.

Now, we have the following transition

$$S_1 = (\nu\, n) \left(\begin{array}{l} P_1 \parallel P_F(\mathcal{F}(O\mathcal{K})) \parallel P_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_t)} P_4(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_t)} \prod_{c \in \vec{c_2}} \langle r_{\circledcirc} s \rangle @c \end{array} \right) \parallel \prod_S \langle \tau_{\circledcirc} \tau' \rangle @c$$
$$\stackrel{\varepsilon}{\Longrightarrow}$$
$$(\nu\, n) \left(\begin{array}{l} P_1 \parallel P_F(\mathcal{F}(O\mathcal{K})) \parallel P_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_t) - \{s'\}} P_4(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_t)} \prod_{c \in \vec{c_2}} \langle r_{\circledcirc} s \rangle @c \parallel [\![[\![CompA(s)]\!]]\!]_a \end{array} \right) \parallel \prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @c = S_2$$

The transactional component activates the compensation and, by construction, the compensation raises the event r to propagate the backward-flow. According to the flows F, this signal is targeted to $\vec{c_2}$. Therefore, we have

$$S_2$$
$$\stackrel{\varepsilon}{\Longrightarrow}$$
$$(\nu\, n) \left(\begin{array}{l} P_1 \parallel P_F(\mathcal{F}(O\mathcal{K})) \parallel P_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_t) - \{s'\}} P_4(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_t)} \prod_{c \in \vec{c_2}} \langle r_{\circledcirc} s \rangle @c \parallel \prod_{c \in \vec{c_2}} \langle r_{\circledcirc} s' \rangle @c \end{array} \right) \parallel \prod_{S-1} \langle \tau_{\circledcirc} \tau' \rangle @c = S_3$$

We verify that the delegated transactional component can perform silent actions up to the emission of the same envelopes for $\vec{c_2}$. Initially, the pending event r can activate only one of the checks reaction of a (Q_6), because all other reactions

wait for restricted topics or are performed by the restricted component b. By construction we have that component a includes a reaction for the received session because $s' \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r)$

$$R_1 = \left\langle a \boxtimes F_1 \;;\; (\nu\, n_1) \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r)} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \end{array} \right) \parallel \prod_S \langle \tau \circledcirc \tau' \rangle @ c \right\rangle$$

$$\stackrel{\varepsilon}{\Longrightarrow}$$

$$\left\langle a \boxtimes F_1 \;;\; (\nu\, n_1) \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r) - s'} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel \bar{r}\, s @ a \end{array} \right) \parallel \prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R$$

The raised event is targeted to b, according to the flows F_1,

$$\stackrel{R_2}{\Longrightarrow}$$

$$\left\langle a \boxtimes F_1 \;;\; (\nu\, n_1) \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r) - s'} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel \langle r \circledcirc s' \rangle @ b \end{array} \right) \parallel \prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R$$

Now, the component b can consume the pending envelope, activating its lambda reaction (Q_5)

$$\stackrel{R_3}{\Longrightarrow}$$

$$\left\langle a \boxtimes F_1 \;;\; (\nu\, n_1) \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r) - s'} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel [\![[\![CompA(s')]\!]]\!]_a \end{array} \right) \parallel \prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R$$

The compensation is activated and, by construction, the event r is raised by the component b. Notice that the spawned envelopes are targeted to c_2.

$$\stackrel{R_4}{\Longrightarrow}$$

$$\left\langle a \boxtimes F_1 \;;\; (\nu\, n_1) \left(\begin{array}{l} Q_1 \parallel Q_4 \parallel Q_5 \parallel Q_F(\mathcal{F}(O\mathcal{K})) \parallel Q_R(\mathcal{F}(\mathcal{EX})) \\ \parallel \prod_{s \in \mathcal{F}(O\mathcal{K}) - \mathcal{F}(\mathcal{N}_r) - s'} Q_6(s) \\ \parallel \prod_{s \in \mathcal{F}(\mathcal{N}_r)} \prod_{c \in \vec{c_2}} \langle r \circledcirc s \rangle @ c \parallel \prod_{c \in \vec{c_2}} \langle r \circledcirc s' \rangle @ c \end{array} \right) \parallel \prod_{S-1} \langle \tau \circledcirc \tau' \rangle @ c \right\rangle = R$$

Finally, by constriction $(S_3, R_4) \in \mathcal{B}$.

C.2 Proof of Theorem 9

Let N_{d_1,d_2} be as in 4.1 and N'_{d_1,d_2} as in (4.2) then $[\![N_{d_1,d_2}]\!] \approx \left[\!\left[N'_{d_1,d_2}\right]\!\right]$

First, we describe the structure of the two networks:

$$\begin{aligned} N_{d_1,d_2} &= (\nu d_1)(\nu d_2)(N \parallel D) \\ N'_{d_1,d_2} &= (\nu d_1)\,(\{d_1/d_2\}N \parallel D') \end{aligned}$$

The proof of the theorem follows the same strategy we adopted for the proof of Theorem 8. However, in this case one cannot simply prove that D is weak bisimilar to D'. In fact, we need to consider more refined information on the network context (e.g. N and $\{d_1/d_2\}N$). For instance, N is basically equivalent to $\{d_1/d_2\}N$ just by suitably updating the flows. Pictorially, we can represent the two networks as in Figure C.1.

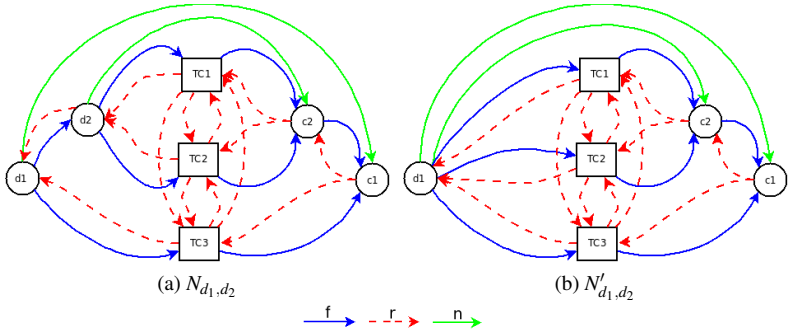


Figure C.1: The example networks

Since N is the translation of a saga process, it is easy to prove, by structural induction, that cannot exist a component that delivers signals to both dispatcher d_1 and d_2. Hence, the network N_{d_1,d_2} can be rewritten as

$$(\nu d_1)(\nu d_2)(N_1 \parallel N_2 \parallel D)$$

where the network N_1 does not contains flows to d_2 and N_2 does not contains flows to d_1.

In the example, N_1 is the network containing TC_3 and c_1, while N_2 is the network containing TC_1, TC_2 and c_2.

Again, by structure induction we obtain that

$$\{d_1/d_2\}N_1 = N_1$$

and by induction over the transition rules it is technically easy (even if elaborated) to prove the following statement:

$$[\![N_2]\!] \xrightarrow{\alpha} [[N_2']] \; \textit{if and only if} \; [\![\{d_1/d_2\}N_2]\!] \xrightarrow{\{d_1/d_2\}\alpha} [[\{d_1/d_2\}N_2']]$$

The key observation is that

- N_2 does not contain any reference to d_1
- the name d_2 can occur only inside envelopes in the network N_2

To guarantee week-bisimilarity we have to check that any component involved into the coordination raises the event r only once for each session. This follows from the observation that

- The network N_1 has one component connected to d_1, then it will deliver up to one envelope $\langle r \circledcirc s \rangle @ d_1$ for each $s \in \mathcal{T}$.
- The network N_2 has two components connected to d_2, then it will deliver up to two envelopes $\langle r \circledcirc s \rangle @ d_2$ for each $s \in \mathcal{T}$.

Indeed, the structural properties detailed above allows one to constraint the multisets of envelopes that characterize successes and failures of behavior.

To better understand this requirement, we report a portion of the *NCP* policy representing a possible state of a dispatcher

$$P_1(s) = r\, s@a.r\, s@a.\overline{f}\, s@a$$

The policy P_1 represents the check reaction installed by the dispatcher to synchronize a backward-flow. A dispatcher can have multiples copies of the policy P_1, depending from the number of forward-flow executed:

$$\prod_{s \in \mathtt{F}(\mathtt{N}_f)} P_1(s)$$

Each copy of P_1 is intended to operate on a distinguished session, otherwise two instances of P_1 compete to handle the same events r. For this reason, it is necessary to constraint the behavior of components in the network context to deliver forward events only once for each session s. This constraint, that can be proved by induction over the structure of *SC* networks implementing sagas, ensures that sessions of envelopes identified by $\mathcal{F}(\mathcal{N}_f)$ and unique.

The statement follows as done for the proof in C.1, substituting the policies P and Q with the translations of the networks D and D'.

Bibliography

[1] "Soa reference model." http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm/. 1, 14

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 2004. 2

[3] W. Vogels, "Web services are not distributed objects," *IEEE Internet Computing*, vol. 7, no. 6, 2003. 2

[4] M. Stal, "Web services: Beyond component-based computing," *Communications of the ACM*, vol. 55, no. 10, 2002. 2

[5] M. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *Proc. Web Information Systems Engineering (WISE)*, 2003. 2

[6] "Soap version 1.2." http://www.w3.org/TR/soap/. 2, 13

[7] "Extensible markup language (xml)." http://www.w3.org/XML/. 2, 12

[8] D. Frankel, *Model Driven Architecture*. New York: Wiley, 2003. 3, 16

[9] "Business process execution language for web services version 1.1." http://www-128.ibm.com/developerworks/library/specification/ws-bpel/. 3, 14, 70

[10] "Web services choreography description language version 1." http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/. 3, 14

[11] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "SOCK: A calculus for service oriented computing.," in *Proc. Service-Oriented Computing (ICSOC)*, vol. 4294 of *Springer LNCS*, 2006. 4

[12] A. Lapadula, R. Pugliese, and F. Tiezzi, "A calculus for orchestration of web services," in *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*, vol. 4421 of *Lecture Notes in Computer Science*, pp. 33–47, Springer, 2007. 4

[13] A. Lapadula, R. Pugliese, and F. Tiezzi, "A formal account of ws-bpel," in *Coordination Models and Languages, 10th International Conference, COORDINATION 200*, vol. 5052 of *Lecture Notes in Computer Science*, pp. 199–215, Springer, 2008. 4

[14] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro, "Scc: A service centered calculus," in Bravetti *et al.* [85], pp. 38–57. 4, 70

[15] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara, "Disciplining orchestration and conversation in service-oriented computing," in *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pp. 305–314, IEEE Computer Society, 2007. 4

[16] M. Boreale, R. Bruni, R. D. Nicola, and M. Loreti, "Sessions and pipelines for structured service programming," in *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008*. 4

[17] H. T. Vieira, L. Caires, and J. C. Seco, "The conversation calculus: A model of service-oriented computation," in *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, vol. 4960 of *Lecture Notes in Computer Science*, pp. 269–283, Springer, 2008. 4

[18] R. Bruni, I. Lanese, H. C. Melgratti, and E. Tuosto, "Multiparty sessions in soc," in *COORDINATION* (D. Lea and G. Zavattaro, eds.), vol. 5052 of *Lecture Notes in Computer Science*, pp. 67–82, Springer, 2008. 4, 70

[19] M. Carbone, K. Honda, and N. Yoshida, "Structured communication-centred programming for web services," in Nicola [83], pp. 2–17. 4, 5, 102

[20] J. Misra and H. M. Vin, "Orchestrating computations on the world-wide web," in *APSEC*, pp. 305–, IEEE Computer Society, 2001. 4

[21] W. V. der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14(1), 2003. 4

[22] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino, "Semantics-based design for secure web services," *IEEE Trans. Software Eng.*, vol. 34, no. 1, pp. 33–49, 2008. 4

[23] W. M. P. van der Aalst and A. H. M. ter Hofstede, "Yawl: yet another workflow language," *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005. 5

[24] R. Bruni, H. C. Melgratti, and U. Montanari, "Theoretical foundations for compensations in flow composition languages," in *POPL* (J. Palsberg and M. Abadi, eds.), pp. 209–220, ACM, 2005. 5, 9, 29, 30

[25] H. Garcia-Molina and K. Salem, "Sagas," in *SIGMOD Conference* (U. Dayal and I. L. Traiger, eds.), pp. 249–259, ACM Press, 1987. 5, 9, 29

[26] J. L. Fiadeiro, A. Lopes, and L. Bocchi, "A formal approach to service component architecture," in Bravetti *et al.* [85], pp. 193–213. 5

[27] D. Strollo, *Designing and Experimenting Coordination Primitives for Service Oriented Computing*. PhD thesis, IMT Lucca, 2009. 5, 112

[28] M. Bravetti and G. Zavattaro, "A theory for strong service compliance," in *COORDINATION* (A. L. Murphy and J. Vitek, eds.), vol. 4467 of *Lecture Notes in Computer Science*, pp. 96–112, Springer, 2007. 5, 102

[29] Y. Huang and D. Gannon, "A comparative study of web services-based event notification specifications," in *ICPP Workshops*, pp. 7–14, IEEE Computer Society, 2006. 6, 7, 38

[30] R. Milner, "The polyadic pi-calculus (abstract)," in *CONCUR* (R. Cleaveland, ed.), vol. 630 of *Lecture Notes in Computer Science*, p. 1, Springer, 1992. 7, 20, 38

[31] "Business process maturity model (bpmm)." http://www.omg.org/spec/BPMM/. 9, 17

[32] M. Wirsing, A. Clark, S. Gilmore, M. M. Hölzl, A. Knapp, N. Koch, and A. Schroeder, "Semantic-based development of service-oriented systems," in Najm *et al.* [84], pp. 24–45. 10, 103

[33] "Software engineering for service-oriented overlay computers." http://www.sensoria-ist.eu/. 10, 103

[34] G. L. Ferrari, R. Guanciale, and D. Strollo, "Jscl: A middleware for service coordination," in Najm *et al.* [84], pp. 46–60. 11

[35] G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto, "Coordination via types in an event-based framework," in *FORTE* (J. Derrick and J. Vain, eds.), vol. 4574 of *Lecture Notes in Computer Science*, pp. 66–80, Springer, 2007. 11

[36] V. Ciancia, G. L. Ferrari, R. Guanciale, and D. Strollo, "Checking correctness of transactional behaviors," in *FORTE* (K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, eds.), vol. 5048 of *Lecture Notes in Computer Science*, pp. 134–148, Springer, 2008. 11

[37] V. Ciancia, G. Ferrari, R. Guanciale, and D. Strollo, "Global coordination policies for services," in *FACS 2008*. 11

[38] G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto, "Refactoring long runing transactions," in *WSFM 2008*. 11

[39] G. L. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto, "Event-based service coordination," in *Concurrency, Graphs and Models* (P. Degano, R. D. Nicola, and J. Meseguer, eds.), vol. 5065 of *Lecture Notes in Computer Science*, pp. 312–329, Springer, 2008. 11

[40] G. Ferrari, R. Guanciale, D. Strollo, and E. Tuosto, "Debugging distributed systems with causal nets," in *PNGT 2008*. 11, 122

[41] G. L. Ferrari, R. Guanciale, and D. Strollo, "Event based service coordination over dynamic and heterogeneous networks," in *ICSOC* (A. Dan and W. Lamersdorf, eds.), vol. 4294 of *Lecture Notes in Computer Science*, pp. 453–458, Springer, 2006. 11

[42] M. Bartoletti, V. Ciancia, G. Ferrari, R. Guanciale, D. Strollo, and R. Zunino, "L'orientamento ai servizi," *Mondo Digitale*, March 2008. 11

[43] "Web services glossary." http://www.w3.org/TR/ws-gloss/. 12

[44] "Web services addressing 1.0 soap binding." http://www.w3.org/TR/2005/CR-ws-addr-soap-20050817/. 13

[45] "Apache axis." http://ws.apache.org/axis/. 13, 14

[46] "Apache tomcat." http://tomcat.apache.org/. 13

[47] "Xml schema." http://www.w3.org/XML/Schema. 13

[48] "Scalable vector graphics." http://www.w3.org/Graphics/SVG/. 13

[49] "Universal business language." http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ubl. 13

[50] D. Kuhlman, "generateds – generate data structures from xml schema." http://www.rexx.com/~dkuhlman/generateDS.html. 13

[51] "The castor project." http://www.castor.org/. 13

[52] "Web services description language (wsdl) 1.1." http://www.w3.org/TR/wsdl. 13

[53] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980. 20

[54] R. M. Amadio, I. Castellani, and D. Sangiorgi, "On bisimulations for the asynchronous pi-calculus," in *CONCUR* (U. Montanari and V. Sassone, eds.), vol. 1119 of *Lecture Notes in Computer Science*, pp. 147–162, Springer, 1996. 26, 28, 84

[55] K. Honda and M. Tokoro, "On asynchronous communication semantics," in *Object-Based Concurrent Computing* (M. Tokoro, O. Nierstrasz, and P. Wegner, eds.), vol. 612 of *Lecture Notes in Computer Science*, pp. 21–51, Springer, 1991. 26, 28, 84

[56] U. Montanari and M. Pistore, "Checking bisimilarity for finitary pi-calculus," in *CONCUR* (I. Lee and S. A. Smolka, eds.), vol. 962 of *Lecture Notes in Computer Science*, pp. 42–56, Springer, 1995. 29

[57] M. Chessell, C. Griffin, D. Vines, M. J. Butler, C. Ferreira, and P. Henderson, "Extending the concept of transaction compensation," *IBM Systems Journal*, vol. 41, no. 4, pp. 743–758, 2002. 33

[58] I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco: Morgan Kaufmann Publishers, 1998. 38

[59] N. Carriero, D. Gelernter, and J. Leichter, "Distributed data structures in linda," in *POPL*, pp. 236–242, 1986. 39, 69

[60] "Uddi specifications tc committee specifications." http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm. 39

[61] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *SIGCOMM* (A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, eds.), pp. 163–174, ACM, 2003. 39

[62] P. T. Eugster and R. Guerraoui, "Distributed programming with typed events," *IEEE Software*, vol. 21, no. 2, pp. 56–64, 2004. 60, 61

[63] S. Bistarelli, U. Montanari, and F. Rossi, "Semiring-based constraint satisfaction and optimization," *J. ACM*, vol. 44, no. 2, pp. 201–236, 1997. 62, 122

[64] R. Milner, "Calculi for synchrony and asynchrony," *Theor. Comput. Sci.*, vol. 25, pp. 267–310, 1983. 68

[65] K. V. S. Prasad, "A calculus of broadcasting systems," *Sci. Comput. Program.*, vol. 25, no. 2-3, pp. 285–327, 1995. 68

[66] L. Cardelli and A. D. Gordon, "Mobile ambients," in *FoSSaCS* (M. Nivat, ed.), vol. 1378 of *Lecture Notes in Computer Science*, pp. 140–155, Springer, 1998. 69

[67] P. Sewell, P. T. Wojciechowski, and B. C. Pierce, "Location-independent communication for mobile agents: A two-level architecture," in *ICCL Workshop: Internet Programming Languages* (H. E. Bal, B. Belkhouche, and L. Cardelli, eds.), vol. 1686 of *Lecture Notes in Computer Science*, pp. 1–31, Springer, 1998. 69

[68] F. Arbab, "Reo: a channel-based coordination model for component composition," *Mathematical Structures in Computer Science*, vol. 14, no. 3, pp. 329–366, 2004. 69

[69] "Javaspace." http://www.jini.org/wiki/JavaSpaces_Specification. 69

[70] C. Priami, ed., *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Rovereto, Italy, February 9-14, 2003, Revised Papers*, vol. 2874 of *Lecture Notes in Computer Science*, Springer, 2003. 69

[71] A. Lapadula, R. Pugliese, and F. Tiezzi, "A calculus for orchestration of web services," in Nicola [83], pp. 33–47. 70

[72] L. Cardelli, G. Ghelli, and A. D. Gordon, "Secrecy and group creation," *Inf. Comput.*, vol. 196, no. 2, pp. 127–155, 2005. 70

[73] G. L. Ferrari, U. Montanari, and E. Tuosto, "Coalgebraic minimization of hd-automata for the pi-calculus using polymorphic types," *Theor. Comput. Sci.*, vol. 331, no. 2-3, pp. 325–365, 2005. 88

[74] "Eclipse." http://www.eclipse.org/, Aug. 2006. 113, 118

[75] "Introducing oracle jms." http://download.oracle.com/docs/cd/B19306_01/server.102/b14257/jm_create.htm. 116

[76] "The eclipse graphical modeling framework (gmf)." http://www.eclipse.org/modeling/gmf/. 118

[77] "Eclipse modeling framework project (emf)." http://www.eclipse.org/modeling/emf/. 118

[78] "The graphical editing framework (gef)." http://www.eclipse.org/gef/. 118

[79] "Service component architecture." http://www.oasis-opencsa.org/sca. 119

[80] "Open composite services architecture." http://www.oasis-opencsa.org/. 119

[81] H. Hosoya, J. Vouillon, and B. C. Pierce, "Regular expression types for xml," in *ICFP*, pp. 11–22, 2000. 122

[82] J. Parrow and B. Victor, "The fusion calculus: Expressiveness and symmetry in mobile processes," in *LICS*, pp. 176–185, 1998. 122

[83] R. D. Nicola, ed., *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, vol. 4421 of *Lecture Notes in Computer Science*, Springer, 2007. 154, 158

[84] E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, eds., *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006*, vol. 4229 of *Lecture Notes in Computer Science*, Springer, 2006. 155, 156

[85] M. Bravetti, M. Núñez, and G. Zavattaro, eds., *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, vol. 4184 of *Lecture Notes in Computer Science*, Springer, 2006. 154, 155