UNIVERSITÀ DEGLI STUDI DI BERGAMO
SCHOOL OF ENGINEERING
DEPARTMENT OF ENGINEERING AND APPLIED SCIENCES
DOCTORAL PROGRAMME IN ENGINEERING AND APPLIED SCIENCES

MODEL-BASED TECHNIQUES FOR THE DESIGN OF MODERN SOFTWARE SYSTEMS

Doctoral Dissertation of:
Steven Capelli

Supervisor:
Prof. Stefano Paraboschi

The Chair of the Doctoral Program:
Prof. Valerio Re

2016 – XXIX

Abstract

THE Thesis reports on the work done for the definition of several techniques to support the design of modern systems. *SCA-PatternBox* is a framework for modeling service-oriented applications with design patterns that also permits the validation and verification of their behavior. *Security Enhanced Docker* study improves the use of SELinux in Docker. Two techniques are presented: *Docker Policy Module* (DPM) improves the use of the SELinux *Type Enforcement* (TE) model and *Category minimization problem* improves the use of the SELinux *Multi Category Security* (MCS) model. *J-CO query language* devised to query heterogeneous collections of JSON objects stored in a NoSQL document DBMS such as MongoDB.

Introduction

THE great spread of systems and services (i.e., Cloud Services, Geo-localization services) which use a large amount of often sensitive data requires the study of techniques and the realization of tools that validate and verify the behavior of the systems and services and support the management of the policy rules to ensure data security. Moreover, the dissemination of large amounts of data, also heterogeneous, requires the study of techniques to support and facilitate their querying.

In line with such vision, this Thesis presents several model-based techniques in order to design and improve modern software system. *SCA-PatternBox* is a framework which uses techniques to validate and verify the behavior of the system in order to support the design of the systems and services. It provides to engineers a tool to find the best design solution for the system. *Security Enhanced Docker* presents two techniques to improve Docker security. These techniques are also useful approaches to follow for security systems design. *J-CO query language* is a language for querying of heterogeneous data. This query language technique can be seen as a guideline for declarative query language design. *J-CO query language* has been designed thinking about the operators useful to query data and those useful to enrich the lenguage. Moreover, J-CO is not complete. It has been thought as a continuously growing language where the developers can add new operators according to their needs (at the moment of the language design, it is not possible to think of all operators useful for the users. The needs can change with time).

In the following, these three research lines will be presented: *SCA-PatternBox*, *Security Enhanced Docker* and *J-CO query language*.

SCA-PatternBox is a framework for modeling service-oriented applications with design patterns that also permits the validation and verification of their

behavior. *SCA-PatternBox*, through XML template-based mechanism, allows the definition of design patterns and their instantiation from a scratch SCA design or from an SCA component assembly of an existing service-oriented application. It also permits to generate automatically the corresponding compound SCA component assembly and the SCA-Java skeleton code (indeed, Java classes and interfaces with appropriate annotations for SCA). Since patterns have two complementary aspects (structural and behavioral), for patterns that have a significant behavioral aspect, it is necessary to understand how service components collaborate to achieve the expected behavior. To this purpose, SCA-PatternBox allows also the formal specification and analysis of the functional behavioral aspect of design patterns using a formal service specification language called SCA-ASM [32, 33] which is based on the formal method Abstract State Machine [29]. In this way, the framework provides formally verified design patterns by producing precise and unambiguous functional descriptions of design patterns in SCA-ASM and then validates and verifies these formal specifications (e.g., through simulation and model checking). The purpose of this work is to facilitate the construction of a service- oriented application and to help engineers understand alternative means for achieving alternative architectures in the application's design and code. Moreover, early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort. The mathematical foundation of the method also facilitates reasoning about component behavior in order to guarantee their correctness.

Docker security study presents two techniques in order to improve Docker security. Docker is a tool to perform container virtualization using Linux containers where a container is a virtual machine used to perform a specific service. Docker provides lightweight virtualization which permits to virtualize many containers on the same host. The containers do not have a dedicated operating system and use the host operating system to perform their services. This behavior creates attack risk and needs isolation. We considered Docker in Fedora Linux distribution that use SELinux [49] in order to ensure containers isolation. We present two techniques in order to improve the use of SELinux in Docker: *Docker Policy Module* (DPM) and *Category minimization problem*. The first one, improves the isolation between host and containers. It permits to define specific security rules for the processes inside the containers, through the definition of SELinux modules. The second one, improves the isolation between containers. It permits to assign the SELinux categories in a specific way and allows the system to use only the categories needed to satisfy the policy. These improvements ensure greater system security giving to the containers only the rules that are needed in order to perform their functionality.

J-CO query language is a query language for heterogeneous collections of possibly geo-referenced JSON objects. The dissemination of many and het-

erogeneous data creates the needed to have easy ways to querying these data. These data are often released as JSON collections. JSON collections is a format for data storage and is supported by several *NoSQL* document DBMSs such as MongoDB (relational/SQL databases are unable to flexibly integrate heterogeneous collections). However, the query language provided by NoSQL DBMS is not easy to use for non programmers. In line with such vision, J-CO query language provides non-programmer users with a declarative query language able to handle JSON collections in such a way objects can be filtered, recombined and aggregated in a flexible way. Ambitiously, this research wants to repeat what was attempted with SQL in the 70s, i.e., enabling database technology for non-programmer users with a declarative query language.

The techniques described in this Thesis exhibit some level of heterogeneity and focus on different research topics. However, they all have in common the goal of using adhoc models to identify techniques that improve the design of modern systems.

The Thesis is organized as follows: Chapter 1 shows the structure and the funzionality of SCA-PatternBox, Chapter 2 presents two techniques to improve Docker security, Chapter 3 shows a query language for heterogeneous collections, named J-CO and then the Conclusions are drawn.

Contents

CHAPTER 1

SCA-PatternBox

1.1 Introduction

Today the biggest shift in mainstream programming and design is toward service-oriented applications. Service-oriented applications are playing so far an important role in several domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few). Cloud service providers, in particular, are expanding their offerings to include the entire traditional IT stack, ranging from foundational hardware and platforms to application components, software services, and whole software applications.

Service-oriented Computing (SoC) is a paradigm for developing loosely-coupled, interoperable, dynamic systems relying on the basic unification principle that "*everything is a service*". *Services* are intended as loosely-coupled autonomous and heterogeneous[1] computational components that are offered by service providers in a distributed environment via publish/discovery protocols. The architectural foundation for SOC is provided by the *Service-Oriented Architecture* (SOA), which states that applications expose their functionality as services in a uniform and technology-independent way such that they can be discovered and invoked over a network and clouds.

Such a paradigm shift relies on interface-based design, composition, and reuse; but, differently from traditional component-based design where abstrac-

[1] Services are in general, heterogeneous, i.e. they differ in their implementation/middleware technology.

tion, encapsulation, and modularity were the only main concerns, the service paradigm raises a bundle of problems which did not exist previously. Early designing, prototyping, and testing of the functionality of such assembled service-oriented applications is hardly feasible since services are discoverable, loosely-coupled, and heterogeneous components that can only interact with others on compatible interfaces. Moreover, this paradigm shift requires us to discover, formally define, document, and share new *design patterns*. A SOA design pattern provides a solution in support of successfully applying service orientation and establishing a quality service-oriented architecture. SOA patterns describe common architectures, implementations, and their areas of application to help in the planning, implementation, deployment, management, and maintenance of complex systems. According to the SOA principles [34], design patterns play a fundamental role to support the engineering of service-oriented applications and the attainment of the strategic goals of SoC. In particular, a special emphasis has been put so far on the development of a catalog of *SOA design patterns*[2].

In line to such a vision, this research proposes a framework, SCA-PatternBox, for modeling service-oriented applications with design patterns. The framework relies on the OASIS open standard *Service Component Architecture* (SCA) [3] as modeling language for heterogeneous service assembly in a technology agnostic way, and on the Java implementation for SCA. A supporting prototype tool based on the Eclipse environment is also available at [5]. It was developed by extending the Eclipse plug-in *PatternBox* [4], an existing design pattern editor for Java code, and by integrating it with the Eclipse-based SCA Composite Designer and the SCA runtime platform Tuscany [1]. Through an XML template-based mechanism, SCA-PatternBox allows the definition of design patterns and their instantiation from a scratch SCA design to be further customized depending on the application needs, or from an SCA component assembly of an existing service-oriented application to generate automatically the corresponding compound SCA component assembly and the SCA-Java skeleton code (indeed, Java classes and interfaces with appropriate annotations for SCA). The template-based approach for the pattern instantiation and code generation makes SCA-Patterbox higher usable than wizard-based approaches where you have to complete the whole design pattern instance at once. Moreover, SCA-PatternBox can be easily extended. New design patterns and code generators can be introduced by defining new XML templates depending on the target implementation platform and application domain.

Since patterns have two complementary aspects (structural and behavioral), for patterns that have a significant behavioral aspect, it is necessary to understand how service components collaborate to achieve the expected behavior. To this purpose, in addition to a Java-like implementation of design patterns,

[2] http://www.soapatterns.org/

SCA-PatternBox allows also the formal specification and analysis of the functional behavioral aspect of design patterns using a formal service specification language called SCA-ASM [32,33]. SCA-ASM is based on the formal method *Abstract State Machine* [29] that allows the definition of executable and state-based specifications of systems behavior. The main goal is to provide formally verified design patterns by producing precise and unambiguous functional descriptions of design patterns in SCA-ASM and then to validate and verify these formal specifications (e.g. through simulation and model checking).

The proposed framework offers several advantages with respect to the current state of art (see Sect. 1.6 for a detailed comparison with related works). In the literature, design patterns are typically described using a combination of natural language, UML class and sequence diagrams, and program code (see related work in Sect. 1.6.1). Such descriptions lack design pattern-specific visual formalisms, leading to pattern descriptions that are hard to understand, hard to incorporate into tool support, and therefore hard to be machine-processable in order to automate their instantiation and application in the current design and then in the software code. The proposed framework was conceived instead with automation in mind and to this purpose a template-based approach was adopted. The purpose of this work is to facilitate the construction of a service-oriented application and to help engineers understand alternative means for achieving alternative architectures in the application's design and code. Moreover, existing approaches are specific to object-oriented system design and do not address SOA design patterns. Instead, our approach addresses also design patterns related to the SOA domain. Finally, some existing approaches that use a mathematical formalism for defining design patterns formally require strong mathematical background to the user and lack of good tool support. Based on the practical and scientifically well-founded ASM formal method, SCA-ASM models are instead executable and without mathematical overkill. SCA-ASM allows modeling both *structure* and *behavior* of service components in a unique framework integrating architectural and behavioral views. By exploiting the prototyping/validation environment for SCA-ASM [33], patterns and components can be executed already at high level of formalization, without caring about implementation details. Early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort. The mathematical foundation of the method also facilitates reasoning about component behavior in order to guarantee their correctness.

A first prototype of the SCA-PatternBox environment was presented as a tool demo at the Eclipse Italian workshop [66], while a preliminary overview of the framework was presented in [58]. This research extends these preliminary works in several aspects. First, this chapter provides a more accurate description of the pattern definition language adopted by the SCAPatternBox

framework and its use through concrete patterns examples. Formal analysis techniques supported by SCAPatternBox for validating and verifying the functional behavioral aspects of patterns and applications are presented. The chapter introduces also a supporting methodology for a general and agile prototyping of a service-oriented application with the SCA-PatternBox framework, and illustrates the methodology through two case studies – the Order system and the Stock Trading System (STS). Lessons learned that we gained through our experience in developing the case studies and that should be retained for future use are also reported. Moreover, a comparison of existing design pattern languages is also presented according to some specific criteria.

The remainder of this chapter is organized as follows. Section 1.2 provides background concepts on SCA, SCA-Java and SCA-ASM. Section 1.3 introduces the SCA-PatternBox language and alternative notations for editing design patterns and instantiating them automatically. Section 1.4 describes the SCA-PatternBox framework and a supporting design/development methodology for prototyping service-oriented applications with design patterns. Section 1.5 illustrates the methodology with two case studies and provides some lessons learned during the development of the framework and its use for the case studies. Section 1.6 surveys the main existing design pattern languages and provides a comparison of them and of our proposed SCA-PatternBox language. Finally, Section 1.7 concludes the chapter and sketches some future directions of our work.

1.2 Background concepts

Service Component Architecture (SCA) [3] is an XML-based component model used to develop service-oriented applications independently from SOA platforms and middleware programming APIs. SCA is also endowed with a visual notation and supported by an Eclipse-based design tool and runtime platforms (like Apache Tuscany, FRAscaTI, IBM WebSphere Application Server V7, etc.) for the development and deployment of service-oriented applications.

Fig. 1.1 shows an *SCA composite* (or *assembly*) as a composition of SCA components. An *SCA component* is a piece of configured software that provides business functions (operations) for interaction with the outside world. This interaction is accomplished through: *services* that are externally visible functions provided by the component; *references* (functions required by the component) wired to services provided by other components; *properties* allowing for the configuration of a component implementation and *bindings* that specify access mechanisms used by services and references according to some technology/protocol (e.g., WSDL binding to consume/expose web services, JMS binding to receive/send Java Message Service, etc.). Services and references are typed by *interfaces*. An interface describes a set of related opera-

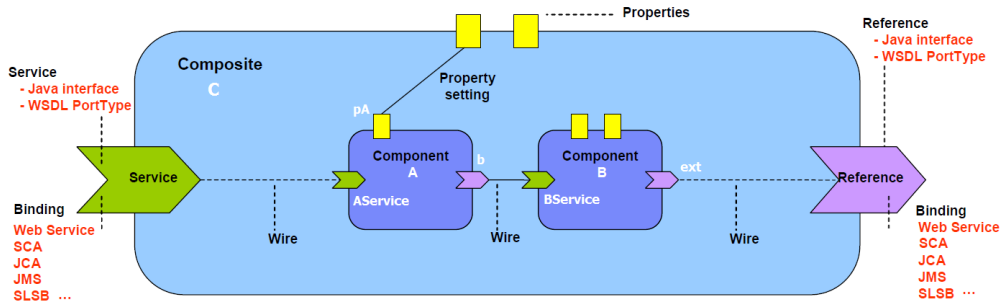


Figure 1.1: *An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)*

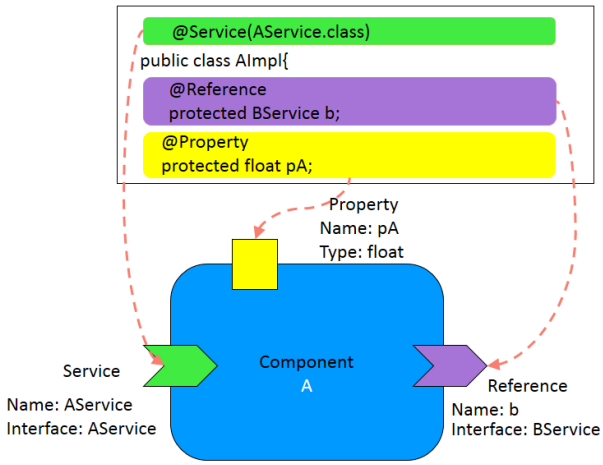


Figure 1.2: *SCA-Java component shape*

tions (or business functions) which as a whole make up the service offered or required by a component. The provider may respond to the requester of an operation with zero or more messages. Message exchange may be synchronous or asynchronous.

The *SCA-Java Component Implementation* [3] defines how to implement an SCA component using Java. Fig. 1.2 shows the SCA component `A` of Fig. 1.1 and its Java implementation class `AImpl`. Java annotations (`@Property`, `@service`,etc.) are used to augment Java classes with SCA concepts.

The *SCA-ASM specification type* [32, 33] complements the SCA component model with the ASM model of computation to provide ASM-based formal and executable description of services *internal behavior*, *orchestration* and *interactions*. An open framework, the ASM toolset ASMETA (ASM mETA-modeling) [2, 56], based on the Eclipse/EMF platform and integrated with the SCA runtime Tuscany, is also available for editing, simulating, validating, and potentially model checking SCA-ASM models [32, 33].

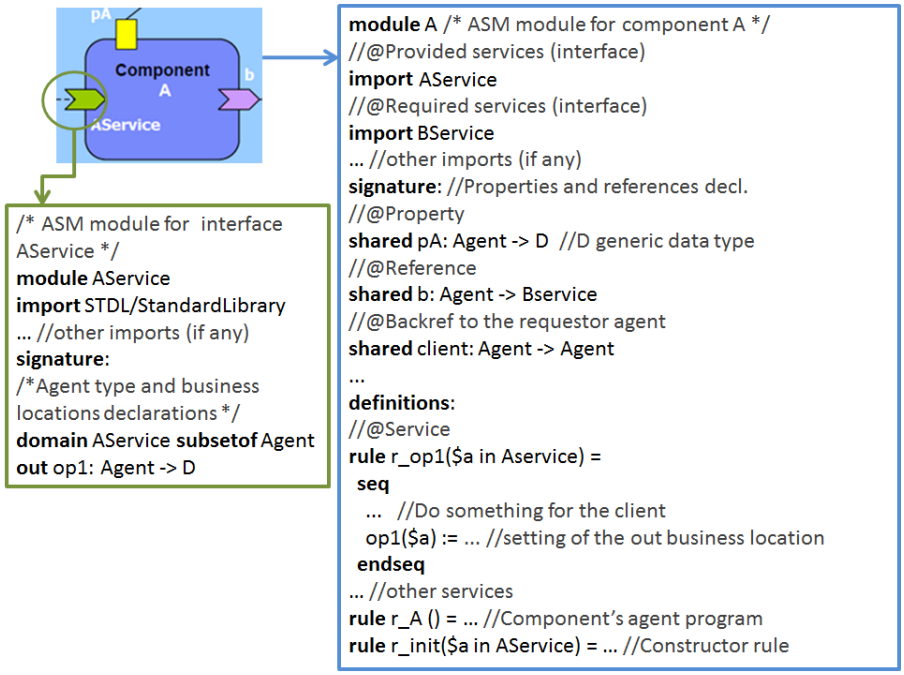


Figure 1.3: *SCA-ASM component shape*

ASMs [29] are an extension of FSMs where states are arbitrary complex data (multi-sorted first-order structures) and the transition relation is specified by rules describing how functions change from one state to the next. The basic rule has the form of *guarded update* "**if** *Cond* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Cond* is true. Rule constructors express parallel actions (*par*), sequential actions (*seq*), iterations (*iterate*, *while*, *recwhile*), nondeterminism (existential quantification *choose*) and unrestricted synchronous parallelism (universal quantification *forall*). Distributed computation is modeled by means of *multi-agent ASMs*: multiple agents interact in a synchronous and asynchronous way, each executing a program specified by an ASM rule.

In SCA-ASM, a service-oriented component is an ASM endowed with (at least) one agent (a business partner or role). Components' agents interact with other agents by providing and requiring services. The service behaviors encapsulated in an SCA-ASM component are captured by ASM transition rules. Fig. 1.3 shows the shape of the SCA-ASM component `A` of Fig. 1.1 and the corresponding ASM modules for the provided interface `AService` (on the left) and the skeleton of the component itself (on the right) using the textual notation ASMETA/AsmetaL and the `@annotations` to denote SCA concepts. ASM rule constructors and predefined ASM rules (i.e., named ASM

Table 1.1: *SCA-ASM rule constructors for computation, coordination, communication*

COMPUTATION AND COORDINATION		
Skip rule	**skip**	do nothing
Update rule	$f(t_1, \ldots, t_n) := t$	update the value of f at $t_1, \ldots, t_n$ to t
Call rule	$R[x_1, \ldots, x_n]$	call rule R with parameters $x_1, \ldots, x_n$
Let rule	**let** $x = t$ **in** R	assign the value of t to x and then execute R
Conditional rule	**it** ϕ **then** R_1 **else** R_2 **endif**	if ϕ is true, then execute rule R_1, otherwise R_2
Iterate rule	**while** ϕ **do** R	execute rule R until ϕ is true
Seq rule	**seq** $R_1 \ldots R_n$ **endseq**	rules $R_1 \ldots R_n$ are executed in sequence without exposing intermediate updates
Parallel rule	**par** $R_1 \ldots R_n$ **endpar**	rules $R_1 \ldots R_n$ are executed in parallel
Forall rule	**forall** x **with** ϕ **do** $R(x)$	forall x satisfying ϕ execute R
Choose rule	**choose** x **with** ϕ **do** $R(x)$	choose an x satisfying ϕ and then execute R
Split rule	**forall** $n \in N$ **do** $R(n)$	split N times the execution of R
Spawn rule	**spawn** child **with** R	create a child agent with program R
COMMUNICATION		
Send rule	**wsend**[*lnk,R,snd*]	send data *snd* to *lnk* in reference to rule *R* (no blocking, no acknowledgment)
Receive rule	**wreceive**[*lnk,R,rcv*]	receive data *rcv* from *lnk* in reference to *R* (blocks until data are received, no ack)
SendReceive rule	**wsendreceive** [*lnk,R,snd,rcv*]	send data *snd* to *lnk* in reference to *R* waits for data *rcv* to be sent back (no ack)
Reply rule	**wreply**[*lnk,R,snd*]	returns data *snd* to *lnk*, as response of *R* request received from *lnk* (no ack)

rules in a model library) are used as SCA-ASM behavioral primitives. These rules are recalled in Table 1.1 by separating them according to the separation of concerns *computation*, *communication* and *coordination*. In particular, communication primitives provide both synchronous and asynchronous interaction styles (corresponding, respectively, to the *request-response* and *one-way* interaction patterns of the SCA standard). Communication relies on a dynamic domain *Message* that represents message instances managed by an *abstract message-passing* mechanism: components communicate over wires according to the semantics of the communication commands reported above and a message encapsulates information about the partner link and the referenced service name and data transferred. We abstract, therefore, from the SCA notion of *binding*[3]. Rules for fault/compensation handling are also supported [33].

1.3 The SCA-PatternBox pattern language

This section describes the SCA-PatternBox language supporting the specification of architectural design patterns and their instantiation into SCA assembly models of service-oriented applications. The section also provides concrete examples of design patterns.

[3]Indeed, we adopt the default SCA binding (`binding.sca`) for message delivering, i.e. the SOAP/HTTP or the Java method invocations (via a Java proxy) depending if the invoked services are remote or local, respectively.

The SCA-PatternBox language consists of a metamodel (the *abstract syntax*) providing a set of modeling constructs to define and reuse design patterns, and of XML-based notations (the *concrete syntax*) for modeling design pattern solutions and solution instances within SCA models. The XML-based concrete notations are both human- and machine- comprehensible.

The SCA-PatternBox language can be used as a stand-alone modeling notation for design patterns or in conjunction with SCA to model design pattern instances within SCA assembly models of service-oriented applications. A design pattern instance describes the relationships between the design pattern elements modeled in the SCA-PatternBox language and the design elements (service-oriented components and interfaces) in the SCA assembly of a specific application.

The proposed language is intended to be used only to model the generalized solutions proposed by design patterns and facilitate their application and reuse. Further details, such as when the solution should be applied and consequences of using the pattern, are not included. The SCA-PatternBox language has been designed with automated tool support in mind. It is relatively easy to learn, particularly in conjunction with SCA.

1.3.1 Design pattern specification

Typically, an architectural design pattern describes components and details their roles and interactions. All components together solve the problem that the pattern addresses.

Fig. 1.4 shows the SCA-PatternBox language's metamodel capturing the very essence of a design pattern independently of its domain. A design pattern in SCA-PatternBox is to be intended as an instance of this metamodel. Essentially, a design pattern (the class `Pattern`) defines roles (the class `Role`), role dependencies (the class `Dependence`), and an informal description (the attribute `comment`) in natural language. Roles are played by the participants of a pattern solution. A role has, among other things, an attribute `type` to tailor the type of architectural design elements (a component or interface) that can play the role, a multiplicity (the attributes `min` and `max`) that constrains the number of elements that can play the role, an attribute `property` to denote a set of mandatory properties, and an attribute `operation` to denote a set of mandatory service operations. The default multiplicity interval `(min,max)` in the class `Role` is 1..*, specifying that there must be at least one element playing the role. Dependencies are typically between a component role and a provided/required interface role, but not necessarily. There can be also dependencies between components themselves to denote, for example, wires connecting a composite component with a sub-component. In this last case the attribute type of the dependence is not present. This is enforced by the follow-

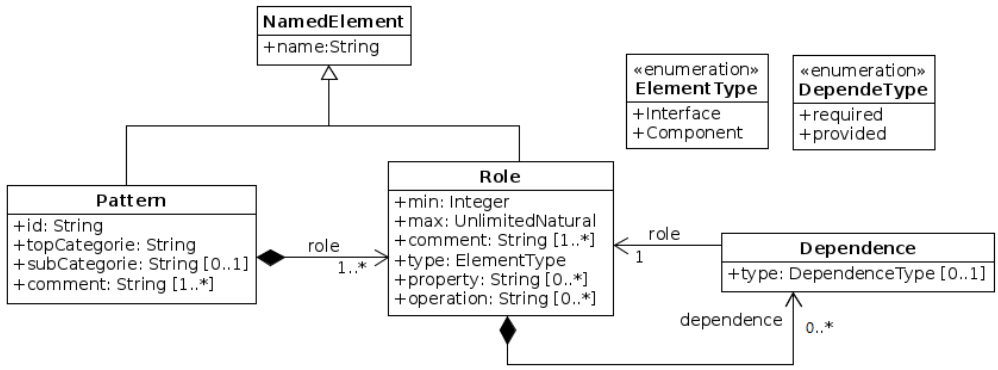


Figure 1.4: *SCA-PatternBox design pattern metamodel*

ing OCL invariant (i.e., a constraint that must always be met by all instances of a class) introduced within the context of the class `Dependence`:

context Dependence

inv targetRoleType: **not** self.type.isNull **implies** self.role.type=interface

Some other OCL constraints not reported here have been defined to constrain the number of valid instances of this metamodel. These constraints are embedded in the SCAPatternBox editor and checked during pattern instantiation.

As concrete textual syntax associated to the metamodel, a pattern is defined in an XML file conforming to a DTD `manifest.dtd`. Code 1.1 reports the XML file for the definition of a service interaction micro-pattern[4] called `Request-Response`. In this interaction schema, a component `ClientRequestResponse` invokes a service of the component `Server` and waits for the result to be returned before continuing with its processing. Request-response is the default mode of invoking a service in a synchronous way.

As complementary concrete syntax, a pattern can be also defined in terms of an SCA assembly model thus exploiting the SCA graphical notation. This SCA assembly consists of service components and reference-to-service wires corresponding, respectively, to roles and role dependencies. Exploiting the SCA (graphical) notation provides a much higher expressive power, but it is tied to a specific design language. Fig. 1.5 shows the SCA assembly for the Request-Response micro-pattern.

[4]Micro-patterns are prime candidates of "units of design" to look for and are the "basis" of more complex SOA patterns.

Listing 1.1: *Request-Response micro-pattern*

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="requestresponse" name="RequestResponse"
topCategorie="SCA" subCategorie="Micro_Pattern">
<role name="RequestResponseService" min="1" max="1" type="Interface" operation="request">
<comment>
<li>defines a service interface.</li>
</comment></role>
<role name="Server" min="1" max="1" type="Component">
<comment>
<li>implements the RequestResponseService interface.</li></comment>
<dependence role="RequestResponseService"/ type="provided"></role>
<role name="ClientRequestResponse" min="1" max="1" type="Component">
<comment>
<li> maintains a reference to a RequestResponseService;</li>
<li> makes a request to a RequestResponseService and waits for the result. </li></comment>
<dependence role="RequestResponseService" type="required"/></role>
</pattern>
```

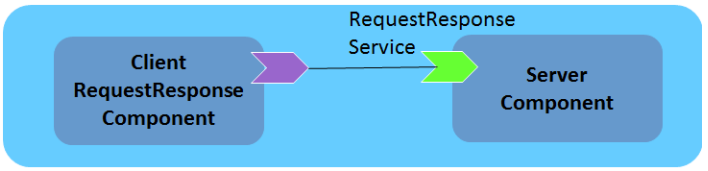


Figure 1.5: *Request-Response micro-pattern in SCA*

1.3.2 Design pattern instantiation

Instantiating a design pattern means creating a new piece of design called "design pattern instance" by mapping the design elements and relationships of the design pattern with elements and relationships of the domain (application) knowledge. After applying (instantiating) a pattern into an existing design, the resulting software architecture should include a particular structure that provides for the roles specified by the pattern, but adjusted and tailored to the specific needs of the problem at hand.

SCA-PatternBox instantiates a pattern by generating from scratch an SCA-assembly and a skeleton SCA-Java code to be further re-factored and refined to the specific application needs. The generation process is *template-based*: there must exist a template for each design pattern supported. Such a template is an XML file conforming to a DTD called `templates.dtd` in SCA-PatternBox. Code 1.2 shows an example of SCA-Java template for the Request-Response micro-pattern.

Similarly, to support the generation of SCA-ASM formal specifications from patterns, we defined a grammar module `ASM templates.dtd` for the definition of SCA-ASM pattern templates. The goal is to precisely define the intended execution semantics of a pattern (which is typically expressed in a quite informal way by graphical notations like SCA or UML) using the ASM

Listing 1.2: *SCA-Java template for RequestResponse*

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE templates SYSTEM "templates.dtd">
<templates id="Request-Response" version="1.0">
<role name="RequestResponseService" type="interface" modifiers="public">
<import type="org.osoa.sca.annotations.*"/>
<method modifiers="public" return="java.lang.Object" name="request">
<comment> The request service operation </comment>
<param type="java.lang.Object" name="item"/>
</method> </role>
<role name="Server" type="class" modifiers="public" >
<import type="org.osoa.sca.annotations.*"/>
<annotation>@Service(RequestResponseService.class)</annotation>
<interface type="$RequestResponseService$"/>
<constructor modifiers="public">
<comment>Default constructor</comment>
<code> super();</code> </constructor>
<method modifiers="public" return="java.lang.Object" name="request">
<comment>Method implementing the service operation
of the RequestResponseService interface. </comment>
<param type="java.lang.Object" name="item"/>
<code> // TODO Write your code here ...
return item; </code> </method> </role>
<role name="ClientRequestResponse" type="class" modifiers="public" >
<import type="org.osoa.sca.annotations.*"/>
<annotation>@Service(Runnable.class)</annotation>
<interface type="java.lang.Runnable"/>
<field modifiers="protected" type="$RequestResponseService$"
name="fRequestResponseService">
<comment>reference to a RequestResponseService</comment>
<annotation>@Reference</annotation>
</field>
<constructor modifiers="public">
<comment>Constructor</comment>
<param type="$ClientRequestResponse$" name="requestresponse" />
<code> super(); </code>
</constructor>
<method modifiers="public" return="void" name="run">
<code> Object item = fRequestResponse.request(item);
// TODO Write your code here ... </code> </method> </role>
</templates>
```

formalism. SCA-ASM offers a more accurate description of the principles involved in a design pattern and offers a more sophisticated insight into the pattern behavior for those familiar with formal notations like ASM.

Code 1.3 shows the SCA-ASM template for the request-response micro pattern, namely the SCA-ASM definition of the interface `RequestResponse` and of the `ClientRequestResponse` and `Server` component roles.

Code 1.4 shows the resulting ASM specification after instantiating the micro pattern from a scratch design. The ASM module `RequestResponseService` corresponds to the `RequestResponseService` interface. It contains only declarations of the business agent type `RequestResponseService` and of the business function (ASM out function) `request`. The ASM module `ClientRequestResponseComponent` imports the ASM module of the required service interface `RequestResponseService` of the component, annotated with `@Required`. The signature of the component contains declarations for a reference (shared function annotated with @Reference) as abstract access endpoint to the `RequestResponse` service, and declarations of ASM functions used by the component for internal computation only. In particular, the function `items` represents data associated to the request made to the server. The ASM module `ServerComponent` imports the ASM module of the provided service interface `RequestResponseService` of the component, annotated with `@Provided`. The annotation `@MainService` on the import clause for the `RequestResponseService` interface denotes the main service (read: main component's agent) that is responsible for initializing the component's state (in the predefined `r_init` rule). The signature of the component contains declarations for: a back reference to requester agent (the shared function `client` annotated with @Backref), and declarations of ASM functions used by the component for internal computation only. This resulting SCA-ASM specification is directly executable within the SCA-ASM execution environment (see Sect. 1.4) for formal validation.

1.3.3 Design patterns examples

We defined and collected different types of design patterns. First, we considered *micro-patterns* for service interaction. Table 1.2 summarizes the SCA micro-patterns for service interactions. Further semantics details can be found in the SCA assembly specification model [3].We inspired to the SCA standard and defined all these service oriented micro-patterns in SCA-PatternBox.

We then started to work with conventional architectural patterns and with SOA patterns (like FIFO, Replicator, Ping-Echo, Publish-Subscribe, Router-Filtering, etc.) related to the service level of abstraction (rather than the business process or orchestration level). Some of them are to be considered as

Listing 1.3: *SCA-ASM template for RequestResponse*

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE templates SYSTEM "templates.dtd">
<templates id="Request-Response" version="1.0">
<role name="RequestResponseService" type="module"/>
<import type="STDL/StandardLibrary"/>
<delimiter type="signature"/>
<domain name="RequestResponseService"/> <subsetof name="Agent"/>
<function modifier="out" name="response">
<domain>Agent</domain> <codomain>D</codomain>
</function>
</role>
<role name="ClientRequestResponseComponent" type="module"/>
<comment> <li>//@Required interface</li> </comment>
<import type="RequestResponseService"/>
<comment> <li>//@Reference to the agent RequestResponseService</li></comment>
<function modifier="shared" name="fRequestResponse">
<domain>Agent</domain> <codomain>RequestResponseService</codomain>
</function>
<function modifier="controlled" name="items">
<domain>Agent</domain> <codomain>D</codomain>
</function>
<delimiter type="definitions"/>
<rule name="r_ClientRequestResponseComponent">
<body> //Make the request in a synchronous manner by send-receive
r_wsendreceive[fRequestResponse(self),"r_request(Agent,D)",items(self),result(self)]</body>
</rule>
<rule name="r_init">
<param type="ClientRequestResponseComponent" name="$a" />
<body> //Complete this rule body for the startup of the component
status($a) := READY </body>
</rule>
</role>
<role name="ServerComponent" type="module"/>
<comment> <li>//@Provided interface</li> </comment>
<import type="RequestResponseService"/>
<comment> <li>//@Backref to the client agent</li> </comment>
<function modifier="shared" name="client">
<domain>Agent</domain> <codomain>Agent</codomain>
</function>
<function modifier="controlled" name="params">
<domain>Agent</domain> <codomain>D</codomain>
</function>
<delimiter type="definitions"/>
<comment> <li>//@Service</li> </comment>
<rule name="r_request">
<param type="Agent" name="$a" /> <param type="D" name="$params" />
<body> skip //Replace this rule body with your ASM rule scheme </body>
</rule>
<rule name="r_ServerComponent">
<body>
let($r = nextRequest(self)) in//Select the next request (if any)
if isDef($r) then seq //Handle the request $r
r_wreceive[client(self),"r_request(Agent,D)",params(self)]
if (isDef(params(self))) then r_request[self,params(self)] endif
r_wreply(client(self),"r_request(Agent,D)",response(self))
endseq endif endlet
</body>
</rule>
<rule name="r_init">
<param type="RequestResponseService" name="$a" />
<body> //Complete this rule body for the startup of the component
status($a) := READY </body>
</rule>
</role>
</templates>
```

Listing 1.4: *SCA-ASM specification of the request-response micropattern*

```
1  //@Remotable
2  module RequestResponseService
3  import STDL/StandardLibrary
4  import STDL/CommonBehavior
5  export ∗
6  signature:
7  domain RequestResponseService subsetof Agent
8  out request: Prod(Agent,D) −> Rule
9
10 module ClientRequestResponseComponent
11 import STDL/StandardLibrary
12 import STDL/CommonBehavior
13 //@Required service
14 import RequestResponseService
15 export ∗
16 signature:
17 //@Reference
18 shared fRequestResponse : Agent −> RequestResponseService
19 controlled items: Agent −> D
20 controlled result: Agent −> D
21 definitions:
22 rule r_ClientRequestResponseComponent =
23 //Make the request in a synchronous manner by send−receive
24 r_wsendreceive[fRequestResponse(self),"r_request(Agent,D)",items(self),result(self)]
25 rule r_init($a in ClientRequestResponseComponent) =
26 //Complete this rule body for the startup of the component
27 status($a) := READY
28
29 module ServerComponent
30 import STDL/StandardLibrary
31 import STDL/CommonBehavior
32 //@MainService
33 import RequestResponseService
34 export ∗
35 signature:
36 //@Backref to the client agent
37 shared client : Agent −> Agent
38 controlled params: Agent −> D
39 definitions:
40 //@Service
41 rule r_request($a in Agent,$params in D)=
42 skip //Replace this rule body with your ASM rule scheme
43 rule r_ServerComponent =
44 let($r = nextRequest(self)) in//Select the next request (if any)
45 if isDef($r) then seq //Handle the request $r
46 r_wreceive[client(self),"r_request(Agent,D)",params(self)]
47 if (isDef(params(self))) then r_request[self,params(self)] endif
48 r_wreply(client(self),"r_request(Agent,D)",response(self))
49 endseq endif endlet
50 rule r_init($a in RequestResponseService) =
51 //Complete this rule body for the startup of the component
52 status($a) := READY
```

tactics, i.e., architectural patterns providing a generic solution to issues pertaining to extra-functional quality attributes (such as performance, availability, etc.). For example, a design concern for availability is "Fault Detection". Since services and their service providers can be discovered at run time, new service providers may be brought into existence at any time and existing service providers may fail or stop operating entirely. To make the application more robust to these kind of faults, two well-known tactics for fault detection are Ping/echo and Heartbeat.

As an example of design pattern, Fig. 1.6 shows the SCA diagram for the Ping/Echo tactic. Ping-Echo is a tactic for monitoring and checking the availability of a component by sending ping messages to the component (the receiver) regularly every `timeInterval` units. If the receiver component does not send back an echo to the sender component within the maximum waiting time (property `maxWaitingTime`), the sender considers the receiver component failed. Code 1.5 shows the XML file defining the Ping/Echo tactic in more general terms for people not familiar with SCA. The Ping/Echo tactic can be intended as a refinement of the call-back micro-pattern.

Fig. 1.7 shows the *Heartbeat* design pattern in SCA as another example of tactic for availability. It detects a fault by listening to heartbeat messages from monitored components periodically. A sender sends a heartbeat message to a receiver (operation `imAlive`) every specified time interval (property `timeInterval`). The receiver updates the current time when it receives the heartbeat message. The aliveness of the sender is checked by the receiver regularly every specified time interval (property `timeInterval`) by comparing the latency time between the current time and the last time of a heartbeat message is received from the server. If the heartbeat message is not received within a certain time (property `maxWaitingTime`) the sender is considered to be unavailable. Code 1.6 shows the XML file defining the Heartbeat tactic.

The Ping/Echo tactic and the Heartbeat tactics can be combined to make more efficient and bidirectional the fault detection mechanism. The result of this tactic composition is shown in Fig. 1.8 using SCA. Technically it has been obtained by applying the Heartbeat tactic to the SCA asssembly of the Ping-Echo tactic. Property re-naming is necessary to avoid ambiguities. For exam-

Table 1.2: *Service Interaction Micro-patterns*

Micro-Pattern	**Communication**	**Annotation**
Local	*pass-by-reference*	–
Remote	*pass-by-value*	@Remotable
Request-response	*Synchronous*	–
Oneway	*Asynchronous*	@oneway
CallBack	*Asynchronous Bidirectional*	@Callback

Listing 1.5: *Ping-echo tactic*

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="pingecho" name="PingEcho" topCategorie="SCA">
<comment>
One component issues a ping and expects to receive back an
echo, within a predefined time, from the component under
scrutiny.
</comment>
<role name="PingService" min="1" max="1" type="Interface" operation="ping">
<comment>
<li>Defines an interface for the Ping sending service.</li>
</comment></role>
<role name="EchoService" min="1" max="1" type="Interface" operation="echo">
<comment>
<li>Defines a callback interface for the Echo sending service.</li>
</comment></role>
<role name="PingSender" min="1" max="1" type="Component"
property= "timeInterval maxWaitingTime ">
<comment>
<li>Maintains a reference to one or more Receiver services.</li>
<li>Sends a Ping to a Receiver component. </li>
</comment>
<dependence role="PingService" type="required"/></role>
<dependence role="EchoService" type="provided"/></role>
<role name="PingReceiver" min="1" max="1" type="Component">
<comment>
<li>Maintains a reference to a Sender service.</li>
<li>Receives a Ping from a sender and reply to it with an Echo.</li>
</comment>
<dependence role="PingService" type="provided"/></role>
<dependence role="EchoService" type="required"/></role>
</role>
</pattern>
```

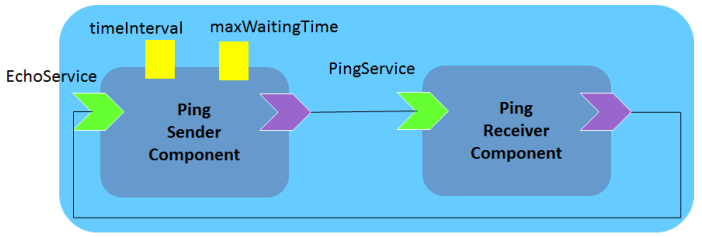


Figure 1.6: *Ping/Echo tactic in SCA*

Listing 1.6: *Heartbeat tactic*

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="heartbeat" name="Heartbeat" topCategorie="SCA">
<comment> One sender component sends a heart beat message to a receiver periodically.
The aliveness of the sender is checked by the receiver regularly every specified time interval.
If the heartbeat message is not received within a certain maximum waiting, the sender is
considered to be unavailable.
</comment>
<role name="HeartbeatService" min="1" max="1" type="Interface" operation="imAlive">
<comment>
<li>Defines an interface for the heart beat sending service.</li>
<role name="HeartbeatSender" min="1" max="1" type="Component" property= "timeInterval">
<comment> <li>Maintains a reference to one or more Receiver services.</li>
<li>Sends an heartbeat message to a Receiver component. </li> </comment>
<dependence role="HeartBeatService" type="required"/></role>
<role name="HeartbeatReceiver" min="1" max="1" type="Component"
property= "timeInterval_maxWaitingTime">
<comment> <li>Maintains a reference to a Sender service.</li>
<li>Receives an heart beat from a sender periodically.</li></comment>
<dependence role="HeartbeatService" type="provided"/></role>
</role>
</pattern>
```

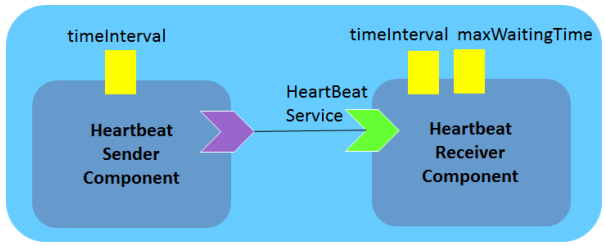


Figure 1.7: *Heartbeat tactic in SCA*

ple, property `timeInterval` of the role *HeartBeatSender* of the Heartbeat tactic is renamed in `beatTimeInterval`. The result of this pattern composition can be saved and reused as new compound design pattern.

1.4 SCA-PatternBox framework

SCA is supported by an Eclipse-based design tool and runtime platforms for the development of service-based applications. Based on the ideas of model-based development, the proposed framework SCA-PatternBox is Eclipse-based and complements the SCA design environment by the use of a collection of models to specify and configure architectural design patterns and the automated application of them in the development of SCA models of service-oriented applications. SCA aims to encompass a wide range of target implementation technologies for SCA service components and for the access methods which are used to connect them. In our work, we focused on the SCA-Java implementation type and on the SCA-ASM formal specification type.

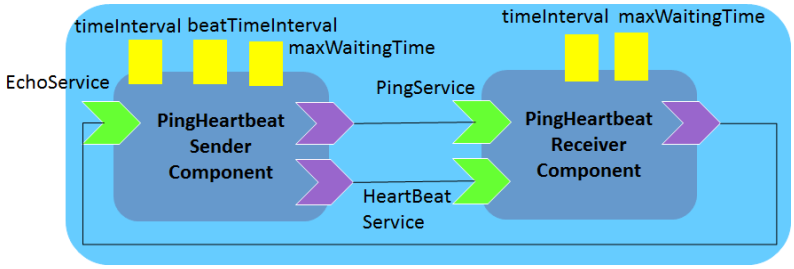


Figure 1.8: *SCA composition of the Ping/Echo and HeartBeat tactics*

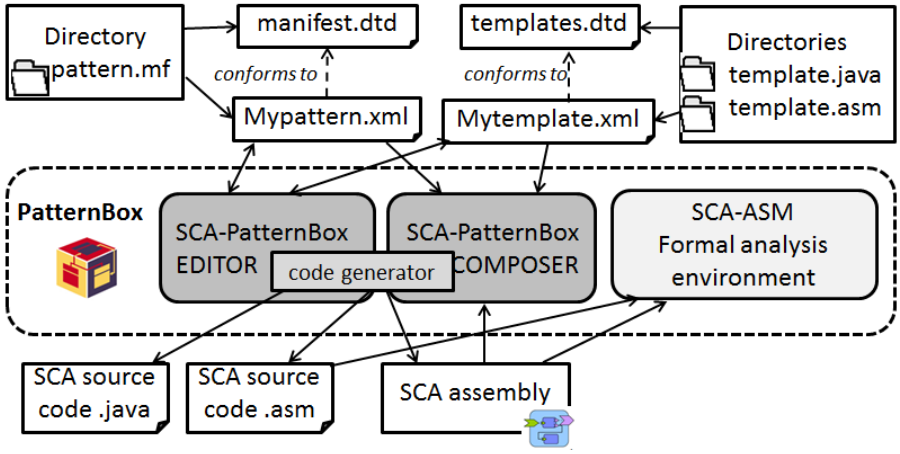


Figure 1.9: *SCA-PatternBox architecture*

In the following, we provide an overview of the SCA-PatternBox framework, a general methodology for application designers to develop service-oriented component architectures using SCA-PatternBox and design patterns, and a description of the supported formal analysis techniques.

1.4.1 Framework architecture

Fig. 1.9 shows the SCA-PatternBox architecture using a free-style notation, while Fig. 1.10 shows a screenshot of the Eclipse view of SCA-PatternBox. SCA-PatternBox includes a design *pattern editor*, *code generators* for producing SCA-Java code and SCA-ASM specifications, and a *pattern composer*. It also exploits an external *formal analysis environment* for SCA-ASM specifications.

Pattern editor The pattern editor adopts a template-based approach that allows you to insert new pattern definitions (file `Mypattern.xml` in Fig. 1.9, and

appropriate templates (file `Mytemplate.xml` in Fig. 1.9) for the generation of SCA-Java code and of SCA-ASM formal specifications. A design pattern is defined in an XML file within a predefined directory `pattern.mf` and must be conforming to the DTD `manifest.dtd` within the same directory.

Code generator There exists an XML-based template files within the predefined directory `template.java` and within `template.asm` of SCA-PatternBox for each design pattern to support the generation of the corresponding SCA-Java code and SCA-ASM specification, respectively. For the SCA-Java code generation, we extended the existing PatternBox XML grammar `templates.dtd` to allow the use of Java annotations for SCA into the code templates, and the source code of PatternBox (`CodeTemplateXmlHandler.java` and `MemberCodeGenerator.java`) to generate Java code from the pattern code templates with the appropriate SCA annotations. The code generator makes intensive use of Eclipse's Java development tooling (JDT) and the Plug-in Development Environment (PDE). It creates Java classes and interfaces according to the code templates and generates also an SCA XML assembly file corresponding to it. Fragments of SCA assembly files and the associated Java skeleton code can be therefore produced from scratch from a pattern definition.

Pattern composer The composer supports the composition and application of design patterns on existing SCA assemblies (and on the corresponding Java or ASM implementation). Currently, the composition is carried out in an interactive manner, indeed the designer has to specify the names of the elements (components or interfaces) from the SCA assembly that play specific roles of the design pattern. The composition strategy is *incremental*, i.e., the application of design patterns to an existing SCA assembly is made through a sequential chain of adaptation actions of the SCA assembly and of the associated components implementation according to the pattern definitions.

Formal specification and analysis environment The formal analysis environment allows for an early and formal validation of the design of a service-oriented application. It consists of the Eclipse-based SCA-ASM design and execution framework for the simulation of SCA-ASM components within an SCA assembly (exploiting the SCA Tuscany runtime platform), and of the ASM analysis toolset ASMETA [2, 56]. An SCA-ASM specification of a service-oriented component (or of a component assembly), possibly not yet implemented in code or available as off-the-shelf, can be: (i) simulated and possibly formally verified (by model checking techniques) offline, i.e., in isolation from the other components, by the use of the ASM toolset ASMETA; (ii) configured in place within the SCA Tuscany runtime platform as abstract implementation (or spec-

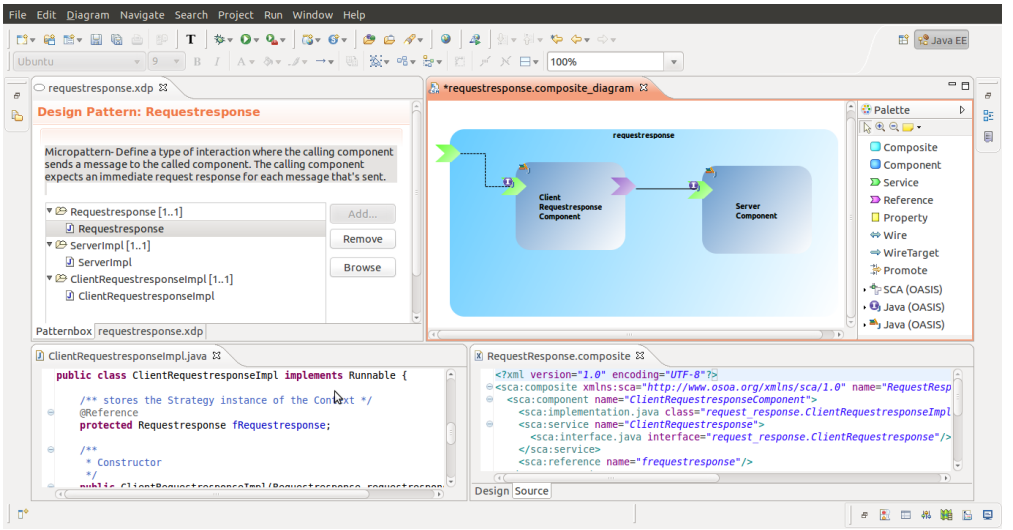


Figure 1.10: *A screenshot of the SCA-PatternBox framework*

ification) of a "mock" component and then executed with the other components implementations according to the chosen SCA assembly.

This toolkit allows to study the behavior of some "critical" components in a formal way before the final transformation into a specific implementation code. The same consideration also apply to component assemblies representing design patterns that can be therefore specified formally in SCA-ASM as abstract and partially complete specifications. Moreover, once these patterns are instantiated into the current SCA design and their specification in SCA-ASM is completed, the structural and behavioral conformance of the component assemblies to design patterns can be (potentially) checked formally.

1.4.2 Framework methodology

A general and agile development of an SCA service-oriented component architecture (or SCA assembly) with SCA-PatternBox can be organized in the following steps (see also Figure 1.11).

Step 1: Sketch an outline of the application's architecture. Define an SCA assembly of the service-oriented application with the SCA Composite Designer by determining the ground services and components required according to the main requirements and use cases from the user perspective.

Step 2: Refine the architecture. Identify the main ways in which the components will interact and the interfaces between them. Decide how each piece of data and functionality will be distributed among the various components. Choose among the various service-oriented design patterns. Instantiate and compose them within the SCA assembly.

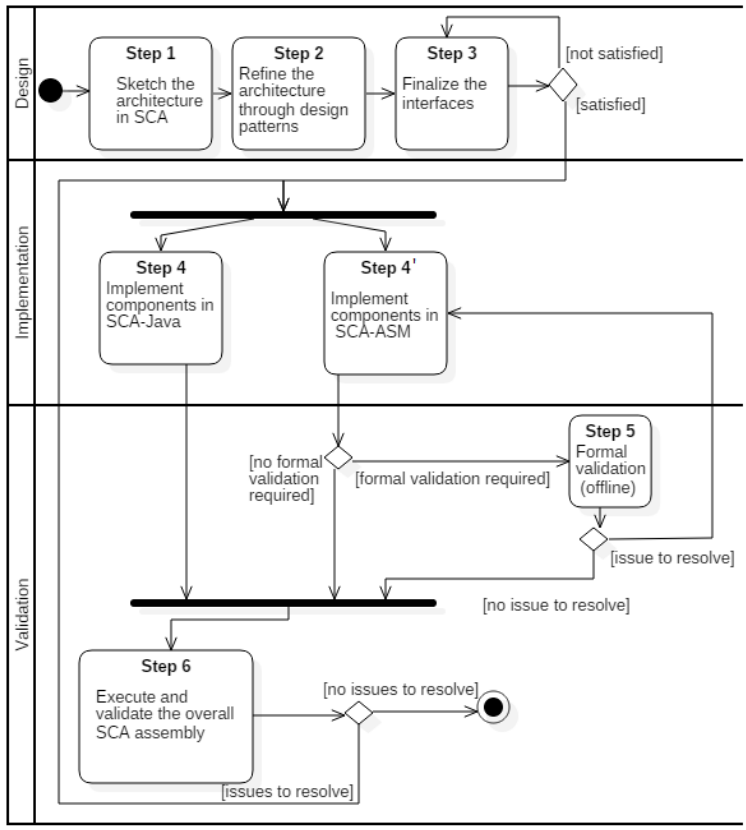


Figure 1.11: *Model development steps with SCA-PatternBox*

Step 3: Finalize the interfaces. Consider each use case and adjust the architecture to make it realizable trying to finalize the interface of each component. Refine components properties in the SCA assembly.

Step 4: Map design to implementation. Finalize the architecture as you define the final implementation classes for components and interaction protocols according to the (possibly many and different) target implementation technologies. Determine if you can re-use existing components implementations before implementing them from scratch. You may implement, for example, some SCA components in SCA-Java and adopt Java-based standard communication bindings – such the Java API for RESTful Web Services (JAX-RS) or the Java Message Service (JMS) – for specifying how SCA services and references enable a component to communicate with other components/applications.

Step 4ı: Map design to formal specification. Optionally, you can specify formally the behavior of some "critical" components using the SCA-ASM im-

plementation type.

Step 5: Formal validation. Optionally, once specified formally the behavior of some components or of a component assembly in SCA-ASM (Step 4′), you may validate such components or assembly separately (in an offline manner) using the ASM analysis toolset ASMETA.

Step 6: Overall design validation. Execute and validate the overall SCA assembly of the application within an SCA runtime platform (like Tuscany).

More sophisticated development processes can be adopted as well. SCA-PatternBox has been used, for example, to support a design exploration process [60] involving more automation and combining meta-heuristic search techniques with design patterns to produce and evaluate different design alternatives.

1.4.3 Formal analysis techniques

We here describe the SCA-ASM model analysis activities we can perform offline for validating and verifying the functional behavior of design patterns, and getting early feedback (already at system design time) of their functional correctness before applying them in concrete systems implementations.

Validation is the process of investigating a model with respect to the user perceptions in order to ensure that the model really reflects the user needs and statements about the application. On the SCA-ASM models of design patterns and their concrete instantiations, formal validation can be carried out in terms of *model simulation* and construction of execution scenarios – *scenario-based validation* – through the execution platform for SCA-ASM [33] and the ASM simulator `AsmetaS` [15] and validator `AsmetaV` [13] provided by the ASMETA framework [2, 56]. Fig. 1.12 shows the validation process: the user can directly simulate an ASM-based specification in an interactive way or write a scenario that automatizes the simulation and the checking of the produced output. Early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort, and usually, though not in an exhaustive manner, it permits to detect faults in the specification with limited effort w.r.t. more sophisticated analysis techniques such as property verification through model checking.

Model simulation and Scenario-based validation `AsmetaS` permits to perform either *interactive simulation*, where required inputs are provided interactively by the user during simulation, and *random simulation*, where inputs values are chosen randomly by the simulator itself. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to

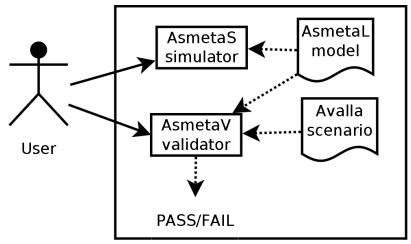


Figure 1.12: *Validation in the ASMETA framework*

two different values at the same time [29]. In preliminary versions of our patterns specifications, by simulation we found some consistency violations due to a wrong order scheduling of the send-receive operations of the participants agents. Moreover, the `AsmetaS` simulator also permits to check if some invariants are satisfied during simulation. Obviously, by simulation we can verify only the states covered by the executed runs, whereas model checking (see next paragraph) gives the assurance that the invariants hold in each model state.

A more advanced way to simulate and inspect ASMs is by specifying a scenario representing a description of the actions of an external actor and the corresponding reactions of the system. There are two kinds of external actors:

- a *user* interacts with the system in a black box manner, by setting the values of the external environment (e.g., asking for a particular service), waiting for a step of the machine as reaction to his/her request, and checking the output values;
- an *observer*, instead, can also inspect the internal state of the system (i.e., values of machine functions) and check the validity of possible invariants of a certain scenario.

Scenarios are described in an algorithmic way using the textual language `Avalla` [13]. A scenario is as interaction sequence consisting of actions of the external actor (user or observer) and activities of the machine as reaction to the actor actions. The `Avalla` language provides constructs to **set** the environment (i.e., the values of input/shared functions), to **check** the machine state, to ask for the **exec**ution of certain transition rules, and to force the machine itself to make one **step** (or a sequence of steps by **step until**).

The tool `AsmetaV` reads scenarios written in `Avalla` and executes them using the simulator `AsmetaS`; during simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict (see Fig. 1.12).

As an example, the excerpt of the scenario reported in Code 1.7 describes the interactions among the client c and server agents s in the pattern request-response from the client side. Appropriate assertions control that the result message is sent. This scenario has to be used as a template and therefore it has

```
1  scenario request−response
2  load mainRequestResponse.asm
3  ...
4  //for the startup of the client and server agents
5  set status(c) := READY;
6  set fRequestResponse(c) := s;
7  set items(c):= ...;
8  set ...
9  exec r_wsendreceive[fRequestResponse(c),"r_request(Agent,D)",items(c),result(c)];
10 step
11 check isDef(response(fRequestResponse(c))) and isDef(result(c));
12 ...
```

Listing 1.7: *Request-response validation scenario in Avalla*

then to be instantiated according to the real services and components involved in the pattern instance. An example of its instantiation will be given for the case studies presented in Sect. 1.5.

Model verification Model checking is an automated formal verification technique based on state exploration of the system to be checked. `AsmetaSMV` [55] is a tool of the ASMETA framework that translates ASM specifications into models of the NuSMV model checker. It allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae expressing desired behavioral properties of the system under verification.

Behavioral properties related to interactions among the participants (agents) can be obtained from a pattern definition and then expressed and configured in terms of CTL formulae within the SCA-ASM model itself of the pattern instance. This last is then validated against the behavioral properties by using the `AsmetaSMV` model checker. Thereafter the state space of the SCA-ASM model to be checked is searched by the model checker to verify or falsify (by generating counterexamples) the CTL properties.

For our purposes, CTL can be used to express a temporal ordering between send and receive messages in a sequence of interactions among agents or also general properties such as *reactivity* and *liveness*. For example, the following CTL property can be checked to validate the *reactivity* of the behavior of an instance of the Request-Response micro-pattern:

```
1 ag(isDef(nextRequest(server)) implies af(isDef(result(client(server)))))
```

It specifies that whenever there is a request, eventually the server (agent *server* of type *RequestResponseService*) will reply producing a result for the client. The formula has then to be instantiated according to the real services of the components involved in the pattern instance.

Liveness properties (informally, liveliness means that some actions will be executed infinitely) can be expressed and verified by giving explicit fairness

requirements. For example, for the Ping-echo tactic, used to test the reachability of a host, the following property can be checked for a timed confirmed sender. Property: A ping message is always followed by an echo confirmation message or timeout:

```
1 ag( ping(sender,receiver) implies af( (maxWaitingTime(sender) and not(echo(
      receiver,sender)))
2 or (not(maxWaitingTime(sender)) and echo(receiver,sender))) )
```

where we assumed the following atomic propositions: *ping(sender,receiver)* $\equiv$ a ping request has been sent from the sender to the receiver, *echo(receiver,sender)* $\equiv$ an echo request has been sent back from the receiver to the sender, and *maxWaitingTime(sender)* $\equiv$ the ping request is not answered with the echo reply within the given time.

Invariants, i.e. properties that must hold in all the states, can be also checked. For example, for a Request-Response micro-pattern we can verify that, after making the request in a synchronous manner by a send-receive action, the client effectively remains blocked until it receives back the result:

```
1 ag( isDef(awaitingRespMsg(client)) implies status(client) = BLOCKED)
```

where *client* is the client component's agent and the controlled function *awaitingRespMsg*, that is set within the predefined rule send-receive [33], stores the message for which the client agent is waiting to receive the corresponding response message. Invariants are useful, for example, for guaranteeing certain *safety* properties (informally, that nothing will go wrong with the system) by verifying that invariants' formulae are effectively true at all states of the system.

Currently, the derivation of CTL formulae from a pattern definition is carried out by hand. We postpone as future work the automatic generation of CTL formulae and their configuration with information from concrete pattern instances. Moreover, verification of properties on large SCA-ASM models is possible, but it would require the use of some *model slicing* and *model abstraction* techniques in order to avoid the well-known problem of state explosion of the model checking, and make the verification feasible.

1.5 Illustrative case studies and lessons learned

This section presents two case studies as illustrative examples of the proposed framework. In the first example, we designed and validated a service-oriented architecture model for the Order system case study [46]. As second example, we considered the Stock Trading system originally presented in [67]. Finally, the section reports our lessons learned as gained by our experience in developing these case studies.

1.5.1 The Order System

The Order system is essentially an exercise of requirements capture, notoriously a difficult and error prone activity that requires a formalization task. To this purpose, we show how the SCA-ASM method allows one to capture informal functional requirements of a system architecture, including both the structural and behavioral aspects of services.

The main service of the system is that of invoicing orders. Every order refers to a product for a certain quantity (greater than zero). The same product can be referenced by several different orders. Every product is in the stock in different quantity. Invoicing requires to check if the order can be satisfied, i.e. if the ordered quantity of products is less than or equal to the quantity in stock. If so, the stock is updated and the order state changes from the state pending to the state invoiced. If the order cannot be satisfied, it is left pending.

Two additional services offered by the system are: cancel orders and add new quantities of products in the stock. A customer can only cancel his or her personal orders. The order is cancelled if it is still pending. When the order is already invoiced, the conflict must be resolved manually by the user. For the third functionality of adding a quantity of product in the stock, the product must be already registered in the system because the system does not consider entries of unreferenced products. So for each product entry, the supplier must specify the product provided and its quantity.

Step 1 – Sketch an outline of the application's architecture

According to Step 1 of the design methodology presented in Sect. 1.4.2, we modeled the initial SCA architecture of the system based (see Fig. 1.13) on the main service of invoicing orders, and the additional services for order cancellation and supply new products quantity. Essentially, we consider the following application scenarios.

Order management (order entry/cancellation) For an order entry scenario, the user requirements specify that an order is made by sending (the service operation `sendOrder(ref, qty, customerID)`) a reference to the desired product, a quantity, and a customer identifier to the system. So the methodology leads us to define a service-oriented component namely `OrderManagement`, which takes into account this entry and is in charge of saving internal orders. For an order cancellation, we assume that a customer must identify the order he wants to cancel and invokes the service operation `cancelOrder(orderID, customerID)` also offered by the component `OrderManagement`. This last checks that the customer can cancel the order (a customer can cancel only one of his or her orders, not the order of another customer) and if it is the case, cancels the order.

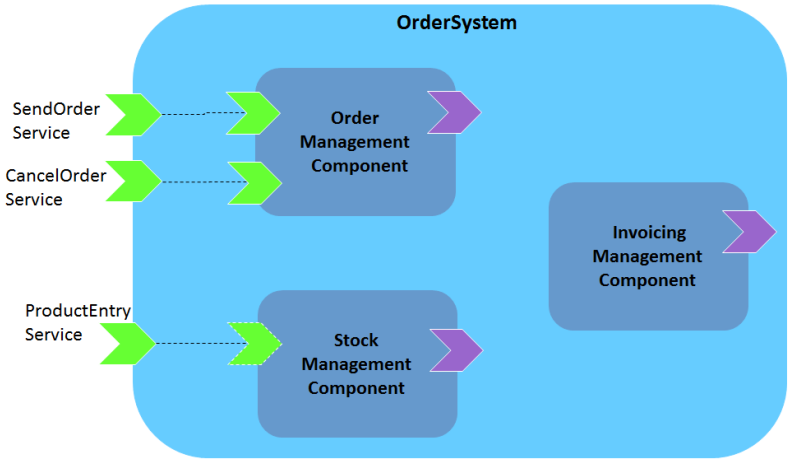


Figure 1.13: *SCA assembly of the Order System*

Stock Management We assume that all products are already referenced in the system. So, when a supplier sends a new quantity of product to the `StockManagement` component (by invoking the service operation `productEntry (productID, qty)`), the product quantity is updated.

Invoicing Management (order invoicing) The `OrderManagement` component only registers in the database that there are new orders to invoice (i.e., orders initially pending), while the component `InvoicingManagement` is in charge of effectively invoicing orders. This last component does not expone any puplic service. It executes the order invoicing functionality in background. Essentially, it selects a set of orders which are invoicable, i.e. they are pending and refer to a product in the stock in enough quantity, it simultaneously changes the state of each order in this set from pending to invoiced, and updates the stock by subtracting the total product quantity in orders to invoice. The system keeps to invoice orders as long as there are orders which can be invoiced. The system guarantees that the state of an order is always defined and the stock quantity is always greater or equal to zero.

Steps 2 and 3 – Refine the application's architecture and finalize the interfaces

The second step of the methodology suggests to refine the architecture by introducing new components and using design patterns.

First, we decided to organize the overall system architecture according to the *three-layer architectural pattern*[5] by introducing the components `GUI`, `Application`, and `Data`. The component `Data` represents the data layer of a classical three-layer-architecture so it hides details of the database and

[5]The templates definition and SCA assembly of such patterns are available online at [5].

provides data access to the application layer represented by the component `Application`. The component `Application` contains the application logic. It uses the services `QueryService` and `PersistenceService` defined by the component `Data` in order to send queries or changes to the database, and provides the interface `StoreService` to deliver results of database queries to the component `GUI`. This last acts like an interface for the user and the `Application` component.

We identify the `OrderSystem` composite component introduced in the previous step as the component `Application`. So according to the three-layer architectural pattern, we have to refine the `OrderSystem` component to allow the interaction with two other external components: the `OrdersDBComponent` (the `Data` component) and the `GUIComponent` (the `GUI`). These last are left in abstract. The result of such refinement is shown in Fig. 1.14.

According to the user requirements, the `InvoicingManagment` component can adopt different selection strategies of orders to invoice: *single-order*, *all-or-none*, *max-orders*, and *default*. Single-order strategy means that per step at most one order is invoiced, with an unspecified schedule (thus also not taking into account any arrival time of orders). In case all orders for one product are simultaneously invoiced or none if the stock cannot satisfy the request, a all-or-none strategy can be expressed. To further maximize a product quantity invoiced at the time, a new strategy (strategy max-orders) consists in choosing a maximal invoicable subset of simultaneously invoiced pending orders for the same product. If the user requests a selection strategy which is not driven by a first choice of a product, another possible strategy consists in choosing a set of pending orders, with enough referenced products in the stock, to be simultaneously invoiced. This last strategy matches the intended behavior of the system better than the previous ones, so it is the default strategy. For supporting one of this mode of operation, we added to the `InvocingManagement` component the property `OrderPolicy` (see Fig. 1.14) whose value range in the set {single-order, all-or-none, max-orders, or default}. By default, this property is initialized to the value *default*.

As further refinement, we applied the *router pattern*[6] to enable different invoicing strategies as implemented by different business components: `SingleOrderBComponent`, `AllOrNoneBComponent`, `MaxOrdersBComponent`, and `DefaultBComponent` (see Fig. 1.15). The router component, i.e. the `InvoicingManagment` component, is responsible for sending the pending orders to a certain business component for invoicing them according to a specific strategy (the routing criteria) as reflected by the current value of the property `OrderPolicy`.

The third step of the methodology is to finalize the interfaces and properties

[6]The templates definition and SCA assembly of such patterns are available online at [5]

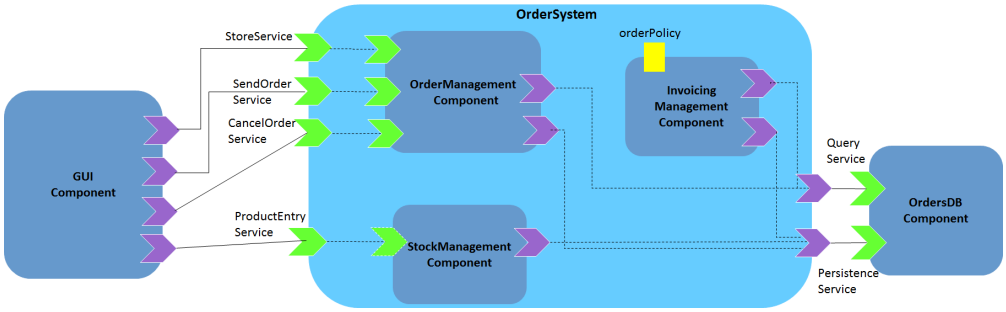


Figure 1.14: *SCA Order system refined (Step 2 — `GUIApplicationData` design pattern)*

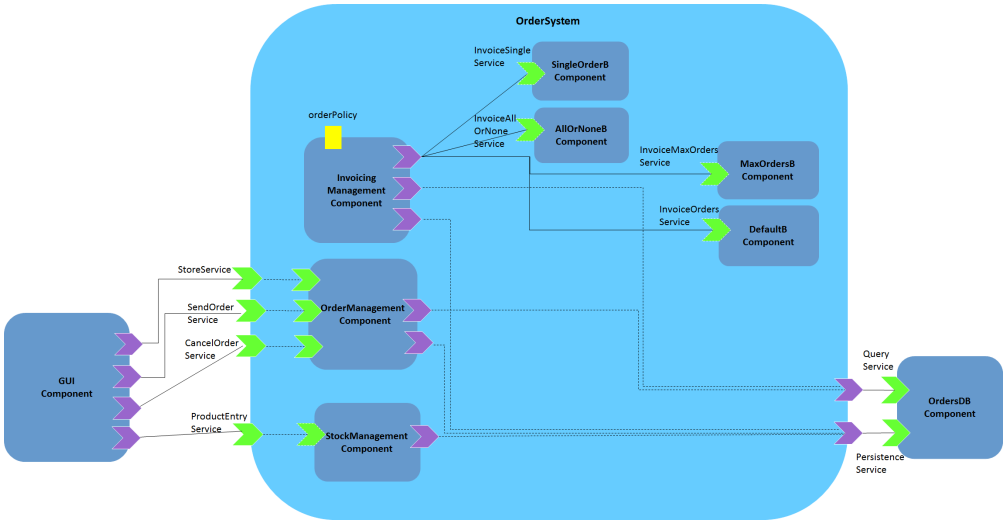


Figure 1.15: *SCA Order system refined (Step 2 — `Router` design pattern)*

Listing 1.8: *ASM module of the* `InvoiceOrdersService` *interface*

```
1  module InvoiceOrdersService
2  import STDL/StandardLibrary
3  import Order //Interface of the Order data type
4  signature:
5  // the domain defines the type of the provider component's agent
6  domain InvoiceOrdersService subsetof Agent
7  out invoiceOrders: Prod(Agent, Powerset(Order)) -> Powerset(Order)
```

of each component in the SCA assembly. So we added the service operations to the interfaces and properties types.

Steps 4/ and 5 – Map design to formal specification and formal validation

Being a design from scratch, before implementing components in Java we preferred specifying the behavior of the business components for invoicing orders formally using SCA-ASM. The complete SCA-ASM implementation of such components is also available at [5].

The default invoice strategy (the `InvoiceOrdersService` service) is provided by the business component `DefaultBComponent`. The ASM definition for the interface `InvoiceOrdersService` is reported in Code 1.8. It is an ASM module containing only declarations of a business agent type (the subdomain `InvoiceOrdersService` of the predefined ASM `Agent` domain) and of a business function used as temporary location to store service computation results (i.e., the orders to invoice) to return back to the service caller.

The behavior of the `DefaultBComponent` is formalized in SCA-ASM as reported in Code 1.9. The service rule `invoiceOrders` uses a predicate `invoicable` that is true on a set of pending orders with enough quantity of requested products in the stack, and a function `refProducts` which yields the set of all products referenced in a set of orders. Note that the non-deterministic selection of the orders to invoice could be performed by a input function which would formalize the user selection of a set of orders or the results of a particular scheduling.

A single-order strategy is realized by the service `invoiceSingleOrder` as provided by the `SingleOrderBComponent`. This service's behavior is formalized in SCA-ASM as reported in Code 1.10. Per step at most one order is invoiced, with an unspecified schedule (by the `choose` rule contructor).

The `InvoiceAllOrNone` service of the `AllOrNoneBComponent` for the all-or-none strategy can be specified in SCA-ASM as reported in code 1.11. The service rule makes use of a function `pendingOrders` yielding the set of pending orders for a certain product, and of a (static) function `totalQuantity` returning the total quantity of a set of orders.

Finally, the service `InvoiceMaxOrders` can be formalized in SCA-

Listing 1.9: *The behavior of the* `DefaultBComponent`

```
asm DefaultBComponent
import STDL/StandardLibrary
import STDL/CommonBehavior
import InvoiceOrdersService
signature:
//@Backref
shared client: Agent -> Agent
// orders to invoice
controlled orders: Agent -> Powerset(Order)
definitions:

rule r_DeleteStock($p in Product ,$q in Natural) = stockQuantity($p):= stockQuantity($p) − $q

//@Service Choose subset of orders
rule r_invoiceOrders($a in Agent, $orders in Powerset(Order)) =
choose $orderSet in Powerset($orders) with invoicable($orderSet) do
par
forall $order in $orderSet with true do orderState($order) := INVOICED
forall $product in referencedProducts($orderSet) with true do
r_DeleteStock[$product, totalQuantity($orderSet,$product)]
invoiceOrders(self,orders(self)) := $orderSet //setting of the out business function
endpar

rule r_DefaultBComponent =
if nextRequest(self)="r_invoiceOrders(Agent,␣Powerset(Order))" then
seq
r_wreceive[client(self),"r_invoiceOrders(Agent,␣Powerset(Order))",orders(self)]
if (isDef(orders(self))) then r_invoiceOrders[self,orders(self)] endif
r_wreply[client(self),"r_invoiceOrders(Agent,␣Powerset(Order))",invoiceOrders(self,orders(self))]
endseq
endif

rule r_init($a in InvoiceOrdersService) = status($a):=READY
```

Listing 1.10: *The behavior of the* `invoiceSingleOrder` *service*

```
//@Service Invoice an order at a time.
rule r_invoiceSingleOrder($a in Agent, $orders in Powerset(Order)) =
choose $order in orders with orderState($order) = PENDING do
if(orderQuantity($order) <= stockQuantity(referencedProduct($order))) then
par
orderState($order) := INVOICED
r_DeleteStock[referencedProduct($order),orderQuantity($order)]
invoiceSingleOrder(self,orders(self)) := $order //setting of the out business function
endpar
endif
```

Listing 1.11: *The behavior of the* `InvoiceAllOrNone` *service*

```
//@Service All orders for one product are simultaneously invoiced or none.
rule r_invoiceAllOrNone($a in Agent, $product in Product) =
let ( $pending = pendingOrders($product) ) in
let ( $total = totalQuantity($pending) ) in
seq
if $total > 0 and $total <= stockQuantity($product) then
par
forall $order in $pending do orderState($order) := INVOICED
r_DeleteStock[$product, $total]
endpar
endif
invoiceAllOrNone(self,product(self)) := $pending //setting of the out business function
endseq
endlet
endlet
```

Listing 1.12: *The behavior of the* `InvoiceMaxOrders` *service*

```
1  //@Service Invoice maximum orders for one product.
2  rule r_InvoiceMaxOrders($a in Agent, $product in Product) =
3  let ($pending = pendingOrders($product)) in
4  let ($invoicablePending = {$o in $pending | totalQuantity($o) <= stockQuantity($product) : $o}) in
5  choose $orderSet in maxQuantitySubsets($invoicablePending) do
6  par
7  forall $order in $orderSet do orderState($order) := INVOICED
8  r_DeleteStock[$product, totalQuantity($orderSet)]
9  invoiceMaxOrders(self,product(self)) := $orderSet //setting of the out business function
10 endpar
11 endlet
12 endlet
```

```
1  scenario DefaultInvoicingManagement
2  load main.asm
3  ...
4  //for the startup of the client and server agents
5  set status(c) := READY;
6  set fRequestResponse(c) := s;
7  set orders(c):= ...;
8  set ...
9  exec r_wsendreceive[fRequestResponse(c),"r_invoiceOrders(Agent,Powerset(Order))",orders(c),result(c)];
10 step
11 check isDef(invoiceOrders(fRequestResponse(c),orders(c))) and isDef(result(c));
```

Listing 1.13: *Validation scenario in Avalla for the* `InvoicingManagementComponent` *– default strategy*

ASM as shown in code 1.12. For this rule we need to define a static function `maxQuantitySubsets` which, given a set of set of orders, returns the set of all the sets having a maximum quantity.

The Order system case study was essentially a requirements formalization task and we have shown how the SCA-ASM method allowed us to capture informal functional requirements of systems services by constructing a consistent and unambiguous, simple and concise, abstract and complete models of service-oriented components, including behavioral aspects of services. These models can be understood and checked (for correctness and completeness) by both domain experts and system architects/designers. For example, Code 1.13 reports a simulation scenario in Avalla (instantiated from the one reported in Code 1.7) for checking the orders invoicing. It checks the interactions among the `OrderDeliveryComponent` (the client c) and the default business component (the server agent s) in the pattern request-response from the client side. Appropriate assertions control that the result message (the set of orders to invoice as chosen by the default strategy) is sent.

Through the requirements capture we have introduced several assumptions to fill missing information. We introduced some assumptions directly in the specification by means of invariants and used the `AsmetaS` simulator to check them during simulation. For example, the assumption that the quantity in every order must be greater than 0 is formalized as:

```
1 invariant over orderQuantity:
2 forall $o in Order with orderQuantity($o) > 0
```

We have also stated the following desired properties which express state invariants and correctness conditions. The first one states that the stock quantity is always greater than 0.

```
1 invariant over stockQuantity:
2 forall $p in Product with stockQuantity($p)>=0
```

The second property is that the state of every order is either pending or invoiced, but never undefined.

```
1 invariant over orderState:
2 forall $o in Order with orderState($o) != undef
```

For other more complex properties, which are not state invariants but they refer to execution paths, the model checker can be used, although assumptions about the finiteness of the domains are necessary and uninterpreted domains are not allowed. For example, one may want to express that an order o is eventually invoiced if it refers to a product available in the stock in enough quantity. In CTL, this can be expressed as:

```
1 AF( AG( orderState(o) = INVOICED or
2 orderQuantity(o) > stockQuantity(referencedProduct(o)))
```

Steps 5 and 6 – Mapping design to implementation and overall design validation

In a further iteration of the methodology, we implemented in Java (Step 4) the components `OrderManagement`, `InvoicingManagement`, and `StockManagement` as mock components, and then execute and validate the overall SCA assembly of the application within the SCA runtime platform (Step 6). We did not effectively implemented all the system in Java since, being an example of requirements elicitation, our main goal was to develop a ground model showing how the SCA-ASM formal method allows one to capture informal behavioral requirements of components' services. So, for validation we used orders collections and customers identifiers with fixed values to avoid at this design phase the implementation of a real database component.

1.5.2 The Stock Trading System

The goal of such a case study is to embody non-functional requirements (NFRs) using architectural tactics. We chose such a case study as major evaluation of the SCA-PatternBox language and tool and as a comparative benchmark because our work is on the spirit of the approach in [67] for design pattern specification and application (see related work in Sect. 1.6.1), but instead of extending UML diagrams like the approach in [67] does, we preferred just to

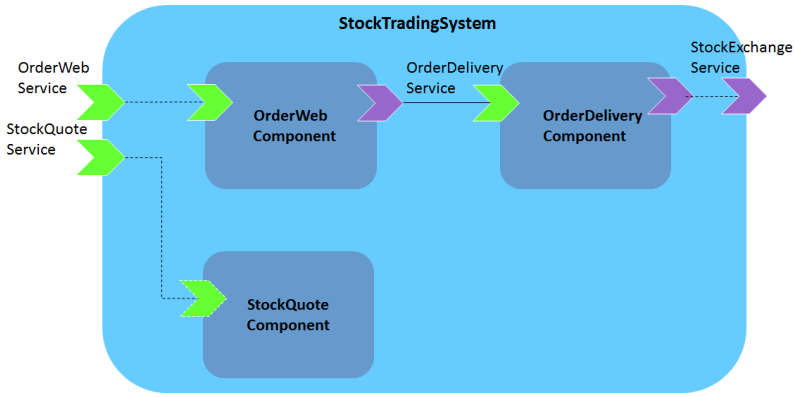


Figure 1.16: *SCA assembly of the Stock Trading System*

specify what is really needed to express a design pattern for facilitating pattern instantiation and code generation.

Our case study was adopted in the adaptation exploration process presented in [60]. It consists of an optimization process with respect to different architectural configurations of the system for multiple adaptation scenarios (e.g., a user claims a new level of reliability and response time, or the monitor raises the violation of the minimum level of required availability).

Figure 1.16 shows the initial SCA assembly of the STS obtained by carrying out Step 1 of the design methodology presented in Sect. 1.4.2. Briefly, an STS user, through the `OrderWebComponent` interacting with the `OrderDeliveryComponent`, can check the current price of stocks, placing buy or sell orders and reviewing traded stock volume. Moreover, he/she can know stock quote information through the `StockQuoteComponent`. STS interacts also with an external Stock Exchange system.

By executing steps 2 and 3 of the methodology, architectural tactics are selected, composed and instantiated based on a given set of NFRs to refine the initial architecture of the application into one that meets the desired NFRs. We here show, in particular, how availability and performance tactics can be used to embody NFRs into the SCA architecture of the STS application. Let us assume the following NFRs (as taken from [67]):
– NFR1. *The STS should be available during the trading time (7:30 AM – 6:00 PM) from Monday through Friday. If there is no response from the system for 30 s, the STS should notify the administrator.*
– NFR2. *The system should be able to process 300 transactions per second, 400,000 transactions per day. A client may place multiple orders of different kinds (e.g., stocks, options, futures), and the orders should be sent to the system within 1 s in the order they were placed.*

We support NFR1 by applying the Fault Detection tactics *Ping/E-*

cho and *Heartbeat* through a monitoring component playing the role of `PingHeartbeatReceiver`. NFR2 is supported by combining the tactics *FIFO* and *Introduce Concurrency*. The *FIFO* tactic allows clients to place each type of orders (e.g., stocks, options, futures) to a dedicated queue for immediate processing. The *Introduce Concurrency* tactic allows the concurrent dispatching of the same kind of orders thus reducing the blocking time of transactions on I/O. Figure 1.17 shows the new SCA assembly obtained by composing these tactics: the assembly is extended to add the new component `Monitoring` (for the Fault Detection tactics) and to refine the existing components `OrderWebComponent` and `OrderDeliveryComponent`. These last two components are to be intended as subsystems, indeed they are composite components with a hierarchical design not further reported here. In particular, the `OrderDeliveryComponent` is refined for adding a queue sub-component for the FIFO tactic, a sub-component for the functionality of a `PingHeartbeatReceiver` role and for the concurrent consuming of different kinds of orders placed into the queue sub-component. Similarly, the `OrderWebComponent` is refined by adding a sub-component for concurrently producing orders to place into the queue of `OrderDelivery-Component`. Of course, this refinement implies a change of the components shape (i.e., in the required/provided interfaces) and of their behavior.

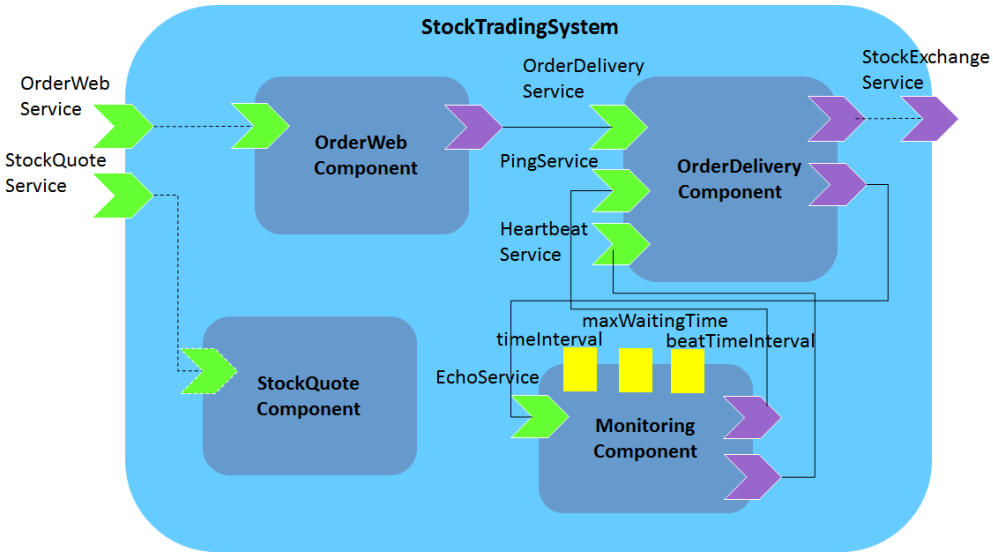


Figure 1.17: *SCA assembly of the STS after applying tactics for NFR1 and NFR2*

Some components were implemented in SCA-ASM (Step 4*ı*). As an example, Code 1.14 shows a fragment of an SCA-ASM specification of the `OrderDeliveryComponent`. The main service of this component (the rule `r_place` annotated with `@service`) is to place buy or sell orders when

Listing 1.14: *The behavior of the* `OrderDeliveryComponent`

```
1  module OrderDeliveryComponent
2  ...
3  @Service
4  rule r_place($client in Agent,$o in Order)= ... //to place buy or sell orders
5  ...
6  rule r_OrderDeliveryComponent=
7  seq
8  r_wreceive(client(self),"place",order(self))
9  r_place(client(self),order(self))
10 r_wreply(client(self),"place",place(self,order(self)))
11 endseq
12 ...
```

Listing 1.15: *ASM modules of the* `OrderDeliveryComponent` *interfaces*

```
1  module OrderDeliveryService
2  import ... //Other module imports
3  signature:
4  // the domain defines the type of the provider component's agent
5  domain OrderDelivery subsetof Agent
6  // business function value
7  out place: Prod(Agent,Order) -> Order
8
9  //@Remotable
10 module StockExchangeService
11 import ... //Other module imports
12 signature:
13 domain StockExchange subsetof Agent
14 out sendOrder: Prod(Agent,Order) -> Rule
```

requested (see the blocking receive action and the reply action preceding and following, respectively, the service invocation within the component's main rule `r_OrderDeliveryComponent`). The ASM definition for the provided and required interfaces of the `OrderDeliveryComponent` are reported in Code 1.15. They are ASM modules containing only declarations of business agent types (the subdomains `OrderDelivery` and `StockExchange` of the predefined ASM `Agent` domain) and of business functions (parameterized ASM out functions) used as temporary locations to store service computation results.

Code 1.16 reports a simplified fragment of a simulation scenario (instantiated from the one reported in Code 1.7) for checking the order placement executed by the `OrderDeliveryComponent`.

The SCA-ASM specification of some components behavior were further refined to reflect at behavioral level the Introduce Concurrency tactic. The behavior, for example, of the `OrderDeliveryComponent` was further refined in ASM as shown in the fragment reported in Code 1.17: the consuming and sending of different kind of orders (stock, option, or future) are executed concurrently by the `par` rule. The availability of such pattern specified in abstract terms by ASM allowed us to formally validate early in the design the benefits provided by such a tactic (Steps 5 and 6).

```
1  scenario OrderPlacement
2  load main.asm
3  ...
4  //for the startup of the client and server agents
5  set status(c) := READY;
6  set fRequestResponse(c) := s;
7  set order(c):= ...;
8  set ...
9  exec r_wsendreceive[fRequestResponse(c),"r_place(Agent,Order)",order(c),result(c)];
10 step
11 check isDef(place(fRequestResponse(c),order(c))) and isDef(result(c));
```

Listing 1.16: *Validation scenario in Avalla for* `OrderDeliveryComponent`

Listing 1.17: *The refined behavior of the* `OrderDeliveryComponent`

```
1  module OrderDeliveryComponent
2  ...
3  rule r_OrderDeliveryComponent=
4  ... seq
5  par //Queue consuming
6  r_wsendreceive[queue(self),"dequeue",("Stock",stockorder(self))]
7  r_wsendreceive[queue(self),"dequeue",("Option",optionorder(self))]
8  r_wsendreceive[queue(self),"dequeue",("Future",futureorder(self))]
9  endpar
10 par //Order sending to the Stock Exchange system
11 r_wsend(stockExchange(self),"sendOrder",(self,stockorder(self)))
12 r_wsend(stockExchange(self),"sendOrder",(self,optionorder(self)))
13 r_wsend(stockExchange(self),"sendOrder",(self,futureorder(self)))
14 endpar
15 endseq ...
```

1.5.3 Lessons Learned

We found the modeling approach to the definition of design patterns and their instantiation in SCAPatternBox particularly useful. First, exploiting the existing PatternBox tool for Java allowed us to develop a large and sophisticated tool for SCA in a very short period of time. The ability to work with design patterns in conjunction with SCA and supporting implementation types was the major benefit. We found it very sympathetic to the implementation of the skeleton code of the application that is obtained automatically from an SCA assembly model and through pattern instantiation.

SCA-ASM allows modeling both structure and behavior of service components in a unique framework integrating architectural and behavioral views. Moreover, SCA-ASM gives us an avenue for the formal validation of the pattern and of the application model to which the pattern is applied. ASM rigorousness, expressiveness, and executability allowed us to reason and validate some services interaction patterns in a formal way but without mathematical overkill. Formal validation can be carried out on SCA-ASM specification fragments resulting from the instantiation from scratch of single design patterns or from the composition of two or more patterns, or on an entire SCA-ASM specification resulting from the application of one or more design patterns on

an existing SCA assembly (partially) implemented in SCA-ASM.

During the development of the case studies, for example, we implemented some components in SCA-ASM in order to have an ASM (abstract) formal specification of their behavior. We then validated the corresponding SCA-ASM specification fragments before and after the application of some formally-validated design patterns, (such as the application of the three-layer and Router patterns to the SCA assembly of the Order system, and of the the Ping-echo and Heartbeat tactics to the STS SCA assembly). We experienced that ASM is a good formalism for prototyping and simulation purposes. We have carried out model validation through the SCA-ASM simulation environment. Model validation is a model analysis activity to be executed from the earlier stages of the model development before other more demanding but more sophisticated and complex analysis techniques such as formal verification by model checking. Formal high-level ASM specifications of SCA components can be assembled together with other SCA components implemented in a different technology (e.g., in Java) and the resulting heterogeneous SCA assembly can be managed and executed within the SCA Tuscany runtime platform as. So SCA components can be early validated at high level of formalization, without caring about implementation details.

During the formalization task of the business components in the Order system case study, we discovered how requirements are often incomplete and assumptions must be stated in order to complete the specification. To this purpose we noted how the SCA architecture (including design patterns) and the SCA-ASM behavioral specification of a service are easy to adapt when different interpretations of same requirements are possible (for example, to support different selection strategies of orders to invoice), and how the rigor of the ASM ground model allows formal validation and verification of properties. Model validation, in general, allowed us to reproduce component configuration scenarios with different pattern instantiations and be confident that the model behaved as expected. We found this early validation is a great means for evaluating architectural choices and alternative designs with limited implementation effort, but it requires to be skilled in the ASM formal method. On this last point, it is widely recognized that ASMs is a lightweight formal method since ASMs are a precise abstract form of pseudo-code, generalizing (the familiar) Finite State Machines to operate over arbitrary data structures [29].

1.6 Evaluation of software design pattern languages

Before starting our work, we evaluated the current state of the art. We, in particular, analyzed existing design pattern languages and their supporting tools. This section reports the outcome of such evaluation. In Subsection 1.6.1 we describe the main approaches existing in the literature, while in Subsection 1.6.2

we briefly compare them (including our proposal, i.e., the SCA-PatternBox pattern language) with respect to some specific criteria.

1.6.1 Related work

In the literature, there are several notations and tools supporting the process of pattern-based system design and development. Some of these languages are based on formal mathematical notations (such as [42, 50, 61, 64, 69]) or ontology (such as [37]). Other languages are based on the OMG standard UML notation [16, 28, 40, 62]. Finally, some other languages are based on XML or general purpose programming languages [14,41,65]. Below, we describe some of these approaches.

The Language for Pattern Uniform Specification (LePUS) [42] is based on mathematics and formal logic. It describes only the structure of design patterns and provides a weak basis for integrated tool support. The language eLeLePUS [61] tries rectify the shortcomings of LePUS.

In [64] another formal language is proposed for the behavioral specification of the GOF design patterns. It is based on the language of temporal ordering specification (LOTOS). This LOTOS adaptation to patterns did not yield simple and clear specifications. Distributed Co-operation (DisCo) [50] is another specification language for design patterns based on an action system to specify the behavioral aspects of a pattern.

The Balanced Pattern Specification Language (BPSL) [69] formally specifies the structural as well as behavioral aspects of patterns. It was derived from LePUS and DisCo, and therefore shares many of the advantages and disadvantages of the two languages.

Conceptual Ontology Design Pattern (CODeP) [37] is based on the Web Ontology Language (OWL) and Resource Description Framework (RDF) for engineering ontology content over the Semantic Web.

Many semi-formal UML-based notations exist. Among the most notable ones are the following. *ArchInst* [68] is a tool developed as a plug-in of the IBM Rational Software Architect (RSA) for UML to support quality-driven development of software architectures. It allows to configure architectural tactics based on quality requirements and compose the configured tactics to produce an initial architecture of the system under development. The tool uses the Role-Based Metamodeling Language (RBML), a UML-based pattern specification notation presented in a previous work [62], to specify tactics. This UML extension defines a design pattern in terms of roles and role dependencies. The concept of *ClassifierRole* that RBML uses has been, however, superseded in UML 2.x. In [27, 67], a prototype tool, called RBML Conformance Checker, has been also presented for verifying the conformance of UML models to design patterns. This work demonstrates how instantiated elements conform to

their corresponding metamodel elements. *PerOpteryx* [16] is an Eclipse-based optimization framework to improve component-based software architectures for performance, reliability, and cost through model-based quality prediction techniques and architectural tactics. It is based on the Palladio Component Model (PCM) and a UML-like modeling notation that uses annotated UML models as software design models. In [28] a prototype tool named *DPTool* is presented. It is based on the DPML (Design Pattern Modeling Language) notation for the specification of design patterns and their instantiation into UML design models. However, the proposed modeling constructs are more complex than other similar UML-based modeling notations. In [40] is presented the problem to prevent defect injection during design-patterns maintenance. A design method called Pattern Instance Changes with UML Profiles (PICUP) has been developed to this purpose as an improved design method for a corrective UML pattern-based design maintenance and assessment after variations.

These are design pattern languages that are based on general purpose programming languages. For example, a Prolog-like language called SPINE [14] allows patterns to be defined in terms of constraints on their implementation in Java. It makes addition of patterns and variants easier for those who have programmed in PROLOG before, rather than creating an entirely new syntax. These patterns can then be processed using a proof engine called Hedgehog. Hedgehog reads the SPINE definitions, along with Java source code, and attempts to automatically prove whether or not the class correctly realizes the design pattern. In [41], AspectJ (aspect-oriented extension to the Java programming language) is used to provide some improvements to the GoF OO design patterns. These improvements allow several advantages like better code locality, reusability, and composability, but are at code level. No abstract modeling notation for pattern specification is supported.

Finally, the Design Pattern Definition Language (DPDL) [65] allows to specify both the structural and behavioral aspects of design patterns. It is a text-based pattern description language purely based on XML, but it is endowed also with a graphical representation through an automated transformation of DPDL descriptions into UML class and sequence diagrams.

1.6.2 A comparison of design pattern languages

To compare existing design pattern languages, we revised and extended the comparison framework originally presented in [65]. The criteria adopted for comparing the languages reflect the main and generally agreed objectives in proposing a language for defining, sharing and applying design patterns into concrete system designs. Essentially, a design pattern language should be:

- Easy to understand and use: The language should be easily understandable by the designers/developers and thus easy to use.

- Unambiguous: The language should have a clear semantics enabling its sharing among design/development teams without any ambiguity.
- Extensible: Because technology is progressing, the language should be capable of being extended.
- Based on existing technology/languages: The language should be based on existing and common technology/languages so that it gains wider and faster acceptance.
- Able to support also a graphical notation: A visual representation of a design pattern and/or a pattern instance provides an intuitive and lightweight overview for the user.

Table 1.3 below summarizes the results of such a comparison by reporting the main features of the design pattern languages described in Subsection 1.6.1 and of our proposed language. The main difference with respect to our proposal, is that all the existing approaches that we considered are specific to object-oriented system design. They do not address design patterns related to SOA. Moreover, most of these approaches are not *implementation-oriented*; indeed, they provide the needed formality in the pattern specification, for example at UML level, but often at the expense of usability and programmaticity of the approach because they do not aim at generating the corresponding implementation code. In particular, languages that purely use a mathematical formalism for modeling design patterns allows to specify the behavioral aspects of a design pattern in a rigorous and unambiguous way, but often they are considered not easy to use because they require strong mathematical background to the user and lack of good tool support. In conclusion, to the best of our knowledge, there is a lack of tools supporting an "implementation-oriented" definition and use of design patterns, especially in the SOA domain. So the experience gained with the development of SCA-PatternBox makes us optimistic.

Table 1.3: *Design pattern languages' features comparison (extended from [65])*

	Basis	Integration in IDEs	Platform independence	Template support	UML support	Learning curve	Graphical	Target	Service oriented
LePUS	Mathematical logic	No	N/A	Yes	No	Required strong mathematical background (high)	No	Verification of design pattern on first order logic basis	No
eLePUS	Mathematical logic	No	N/A	Yes	No	Required strong mathematical background (high)	No	Design pattern designing through mathematical formalism	No
DPML	UML based	No (separate IDE)	Yes	Yes	Yes	UML knowledge required (medium)	UML	Creating design patterns in UML	No
RBML	UML based	No	Yes	No	Yes	UML knowledge required (medium)	UML	Adding support of design pattern in UML	No
DisCo	Temporal logic of action (TLA)	No	N/A	Yes	Yes	Based on rigid formal apparatus (high)	No	Capturing behavioral aspect of design patterns	No
SPINE	Prolog	No	N/A	Yes	No	Required Prolog understanding (hard)	No	Verification of design pattern implementation in application	No
CODeP	OWL (Web Ontology)	No	Yes	No	No	Knowledge of RDF and OWL required (hard)		Creation of knowledge artifacts based on RDF	No
LOTOS	TLA	No	N/A	No	No	Based on rigid formal apparatus (high)	No	Verifying the behavioral aspect of design patterns	No
BPSL	TLA	No	No	Yes	No	Based on rigid formal apparatus (high)	No	Capturing behavioral aspect of design patterns	No
DPDL	XML	Yes (using XML)	Yes	Yes	Yes	XML knowledge is required (Low)	Yes - Optional	Easy initiation and implementation in software development	No
PCM	UML based	Yes	Yes	Yes	Yes	UML knowledge required (medium)	Yes	Prediction of QoS	No
PICUP	UML based	Yes	Yes	Yes	Yes	UML knowledge required (medium)	Yes	Pattern design mantainer	No
SCA-PatternBox Language (our approach)	XML, SCA, ASM	Yes	Yes	Yes	No	Scalable: – Knowledge of XML, SCA (low effort) – Knowledge of ASMs (high effort)	Yes - Optional	Prototyping and formal validation of service-oriented software	Yes

1.7 Evaluation of SCA-PatternBox

We presented a methodology and a supporting framework SCA-PatternBox for the design and prototyping of service-oriented applications with design patterns. The framework allows the definition and the semi-automated application of design patterns into the design of a service-oriented software architecture. We evaluated the usability and usefulness of the framework on the Order system case study and on a quality-driven adaptation scenario of the Stock Trading System [60] where patterns and tactics required would have been difficult to apply and combine manually without the availability of a tool like SCA-PatternBox. We also provided a comparison of the SCA-PatternBox's language with existing design pattern languages.

There are a number of issues that we want to address in the future:

– Automation support for design pattern composition to create complex pattern hierarchies through composition strategies.

– Formal strategies and techniques to check the structural and behavioral conformance of the component assemblies to the applied design patterns.

– Transformation support for compatibility with UML component diagrams, since UML is the standard language for system modeling.

– Definition of more sophisticated SOA patterns.

– Support for more SCA component implementation types.

Moreover, a software engineering area that is gaining importance for the maintenance and evolution of software systems is reverse engineering [30]. A real challenge in this context is to obtain representations of a software system at a higher level of abstraction and to identify the fundamental components, its constituent structures and design patterns. As future work, we want to extend the proposed framework with a methodology for the detection of design patterns and the reconstruction of service architecture models from the source code.

Finally, since Cloud service providers are expanding their offerings to include cloud-native software services (other than foundational hardware and platforms to application components), we would also like to extend our framework to support the new standard TOSCA [6] for modeling cloud-based applications and design patterns related to the Cloud domain.

CHAPTER 2

Security Enhanced Docker

2.1 Introduction

Docker is a tool to perform container virtualization using Linux containers where a container is a virtual machine used to perform a specific service. *Docker* provides lightweight virtualization which permits to virtualize many containers on the same host but provides less security than Hypervisor virtualization. Hypervisor virtualization is a type of virtualization where each virtual machine has a dedicated operating system (OS) separated by host operating system using the Hypervisor. It provides a heavier but more secure virtualization than Linux containers. Linux container virtualization provides less secure than Hypervisor virtualization because the containers (virtual machine) are not completely separated by the host operating system, in fact the containers do not have a dedicated OS but use the host operating system in order to perform their functionality.

Docker uses Linux container virtualization and leverages Linux kernel security features such as *kernel namespaces* to isolate users, processes, networks and devices, and *cgroups* to limit resource consumption. In this chapter we treats Docker used in Fedora Linux distribuction. It uses the SELinux Mandatory Access Control in order to protect the containers between them and the host from the containers. SELinux uses two kinds of policy module: *Type Enforcement* (TE) in order to protect the host from the containers and *Multi Cate-*

gory Security (MCS) in order to protect the containers between them. SELinux assigns a *security context* (or label) to each element of the system (*Subject* or *Object*). The *security contexts* are used by TE to define the rules inside the system policy. Moreover, groups of categories can be assigned to system elements providing a further level of security (MCS).

SELinux is used in *Docker* assigning a *security context* with the *type* equal to $svirt\_lxc\_net\_t$ for each process (*Subject*) inside the containers and a *security context* with the *type* equal to $svirt\_sandbox\_file\_t$ for each resource (*Object*) inside the containers. Moreover, the categories in *Docker* are not used with all containers. For example, the *Data Volume Containers* (containers only used for storage) do not have the categories and are not protected by MCS.

This way of using SELinux permits to have more privileges than necessary, with some attack risk from malicious containers. We present some *Docker* security improvements which permit to improve the use of SELinux in *Docker*: *Docker Policy Module* (DPM) and *Category Minimization problem*.

Docker Policy Module (DPM) adds the SELinux module to each *Docker image* in order to assign a dedicated *security context* to each process and resource inside a specific container (running from the Docker image associated). SELinux module also permits to define specific rules for the *security context*.

Category Minimization problem permits to assign the categories also to *Data Volume Containers* and permits to minimize the categories used by the system. These improvements ensure greater system security giving to the containers only the rules that are needed in order to perform their functionality.

This chapter is structured as follows: Section 2.2 explains Discretionary Access Control (DAC) and Mandatory Access Control (MAC), Section 2.3 presents the SELinux features, Section 2.4 describes the Docker functionality, Sections 2.5 and 2.6 explain our research developed in order to improve Docker security (*Docker Policy Module (DPM)* and *Category Minimization problem*). The last two Sections 2.7 and 2.8 present Related work and Evaluation of Docker security improvements.

2.2 Access Control mechanisms: DAC and MAC

In the system there are many resources and some of them may be critical. Resources in a system have to be protected from unauthorized access. A user (*Subject*) can access only some resources (*Objects*) in the system. In order to do access control in the system, several types of Access Control mechanism have been implemented. The most significant are *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC).

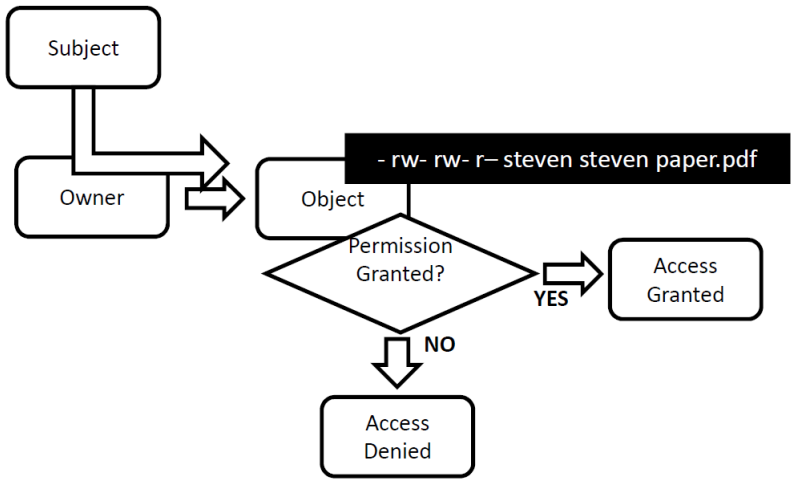


Figure 2.1: *Discretionary Access Control (DAC)*

	O1	O2	O3
S1	r	rw	rwx
S2	rw		rw
S3		rwx	r

Figure 2.2: *Access Matrix*

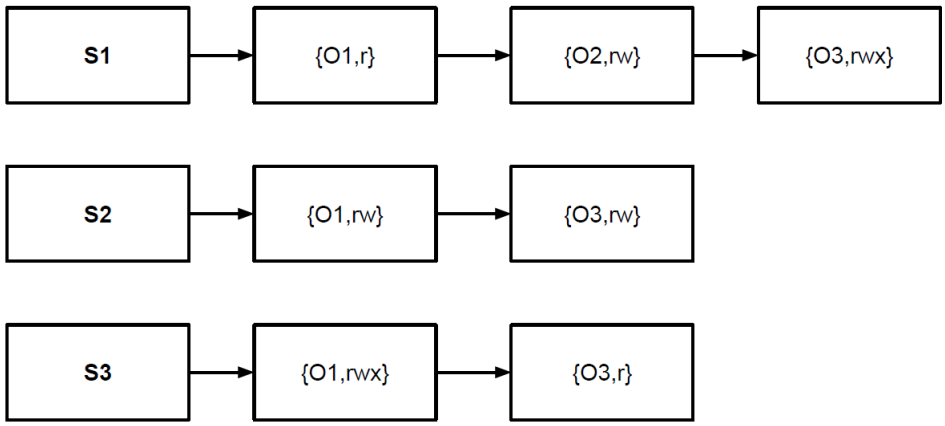


Figure 2.3: *Capabilities*

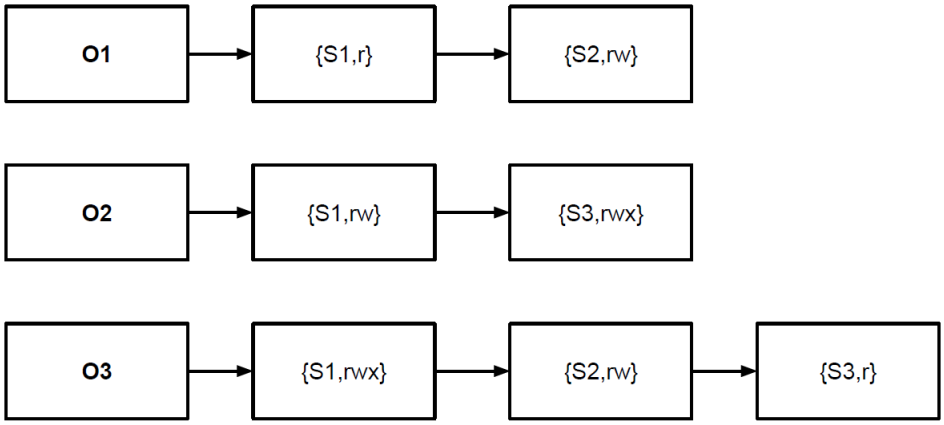


Figure 2.4: *Access Control List (ACL)*

2.2.1 Discretionary Access Control (DAC)

In Discretionary Access Control, Subjects, Objects and Owners come into play. A Subject can be the Owner of an Object and he is able to manage the permissions on that Object.

In DAC the permissions to access an Object can be stored using: *Access Table*, *Access Control List (ACL)* or *Capabilities*.

The *Access Matrix* is a matrix where the rows represent the *Subjects*, the columns represents the *Objects* and each cell represents the actions that a *Subject* can do an *Object*. Figure 2.2, for example, shows a possible *Access Matrix* of a system composed by three *Subjects* and three *Objects*. The cells represent the actions that can be: read (*r*), write (*w*) and execute (*x*).

The actions (considering the *Objects* as files or directories) have the following meaning:

- *read*: it allows to open the file in read mode and, in case of a directory, to view the directory content.
- *write*: it allows to modify the file and, in case of a directory, to create or remove files inside the directory.
- *execute*: it allows to execute a file (if it is executable) and, in case of a directory, to access the directory content.

The *Access Matrix* could have many empty cells (when a *Subject* cannot do any actions on an *Object*) and to solve this problem, it is possible to use the other methods of permissions management, which are used to represent a sparse matrix (*Access Control List* and *Capabilities*).

With *Capabilities*, each *Subject* is associated with a list. The list contains the *Objects* on which the *Subject* can do actions (each element of the list con-

tains an *Object* with associated actions). Figure 2.3, for example, shows a possible *Capabilities* of a system with three *Objects* and three *Subjects*.

With *Access Control Lists (ACL)*, each *Object* is associated with a list. The list contains the *Subjects* that can do actions on that *Object* (each element of the list contains a *Subject* with associated actions). Figure 2.4, for example, shows the possible *ACL* of a system with three *Subjects* and three *Objects*.

Figures 2.2, 2.3 and 2.4 show the permissions rapresentation, of the same system, for the different methods. The *Capabilities* and *ACLs* permit to store the permission information in more efficient way than *Access Matrix*. Figures 2.3 and 2.4 show that *Capabilities* and *ACL* do not waste space and all list elements are filled·

An example of *Access Control List (ACL)*, in Unix system, is shown in the black box of Figure 2.1. It shows the DAC permits and other information about a file named *paper.pdf*. The first character indicates the file type. The character *d* indicates a directory and the absence of *d* character indicates a file. The following nine characters indicate the permissions separated into three groups. The first triple indicates the permissions associated with the owner, the second triple indicates the permissions associated with the owner's group (each user is associated with a group) and the last triple indicates the permits associated with other users(all users that are not the object owner and all users outside the owner group). Inside the triple, there are three types of action: read (*r*), write (*w*) and execute (*x*); with the same meaning shown before.

The following two words indicate: the first one, the owner name (*steven*) and the second one, the owner's group name (*steven*). The last word indicates the file name (*paper.pdf*).

In Discretionary Access Control, the *Owner* of the *Object* can give the permissions on that *Object* also to another *Subject*. The latter can give the received permission on the *Object* to another *Subject* and so on. However, a *Subject* S_1 which is not owner of an *Object* O_1 cannot give his permissions on O_1 to another *Subjects* changing the system permissions. S_1 may provide his permissions on O_1 giving to anothers *Subjects* an application that uses his permissions on O_1. In this way, also other *Subjects* can get the permission on O_1. This behavior reduces the DAC security.

When a *Subject* or an *Owner* wants to do an action on an *Object*, he has to be allowed by the Access Control System. The Access Control System decides according to the system permissions whether to allow or not to allow the *Subject* to access the *Object*. If the *Subject* can do that action on the *Object*, the access is granted, otherwise the access is denied (see Figure 2.1).

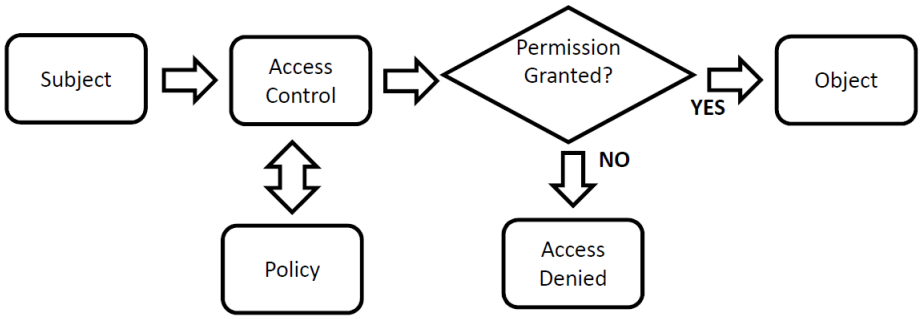


Figure 2.5: *Mandatory Access Control (MAC)*

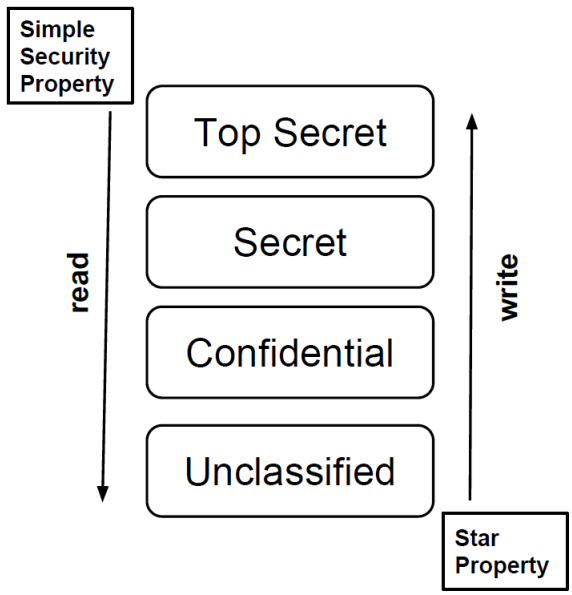


Figure 2.6: *Bell La-Padula (BLP)*

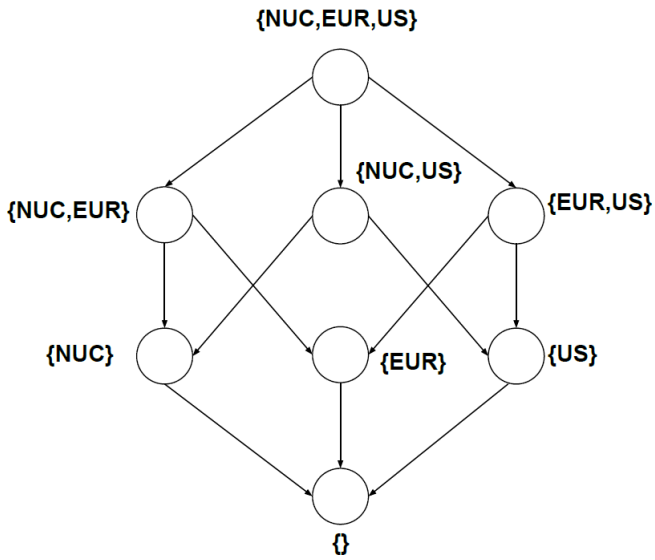


Figure 2.7: *Bell La-Padula (BLP): Lattice Categories*

2.2.2 Mandatory Access Control (MAC)

In Mandatory Access Control, *Subject* and *Object* come into play (in some systems the distinction beetwen *Subject* and *Object* is dropped), the concept of *Owner* is missing. A *Subject* cannot be the owner of an *Object* and cannot change the permits of the *Objects*.

The permissions on the *Objects* can only be changed by the system administrator. The permissions that the *Subjects* have on the *Objects* are stored into a file named *Policy*. Any row of the *Policy* file is named *policy rule* and it has, for example, the following form: **`allow`** `Subject Object {actions}`.

The rule specifies that the *Subject* can do the set of actions (*actions*) on the *Object*. The set of these *policy rules* forms the *Policy* file (the system policy).

When a *Subject* wants to do an action on an *Object*, he has to verify the permission to the Access Control System. The Access Control System decides according to the system policy (*Policy* file) whether to allow or not to allow the *Subject* to access the *Object*. If the *Subject* can do that action on the *Object*, the access is granted, otherwise the access is denied (see Figure 2.5).

One of the most popular security models, based on MAC, is the Bell-La Padula (BLP) model. It follows rules in order to protect data confidentiality (avoid the information read by unauthorized Subject). It is based on state-machines and each state has *Objects* and the current access information. Any *Object* has a classification level ($L(O)$). The classification levels are, for example, *Top Secret*, *Secret*, *Confidential*, *Unclassified* and they have a linear hierarchical structure as shown in Figure 2.6. Moreover, any *Subject* has a maximal

security level ($L_m(S)$) and a current security level ($L_c(S)$). The security levels are equal to the classification levels, for the *Objects*.

According to BLP, a state is secure if it satisfies two properties (see Figure 2.6):

- Simple Security Property (no read up): S can read O iff $L_m(S) >= L(O)$
- Star Property (no write down): S can read O iff $L_c(S) >= L(O)$ and S can write O iff $L_c(S) <= L(O)$

In Bell La-Padula model the *Subjects* and *Objects* can also have some categories, in addition to the security level. Any *Subject* or *Object* can have a *security level* and a *category set* as follows:{*security level*,{*category set*}}. In this way, a *Subject* can read/write an *Object* whether:

- It is allowed by *Simple security property* and *Star property* (comparing *security level* of *Subject* and *Object*)
- The *category set* of *Subject* includes the *category set* of the *Object*

For example, categories (*NUC*, *EUR* and *US*) are added to the system in Figure 2.6. A *Subject* with {*TS*,{*NUC,EUR*}} can read an *Object* with {*S*,{*EUR*}}. Moreover the system structure becomes complex. The categories create the complex lattice structure shown in Figure 2.7. In Figure 2.7, each node can be seen as *Subject* or *Object* with the *category set* indicated. The *security level* is not indicated because it is irrelevant for comparison of categories. However, a *Subject* can read/write an *Object* iff it is allowed by both *security level* and *category set*.

The Bell La-Padula (BLP) model can be implemented by SELinux [49].

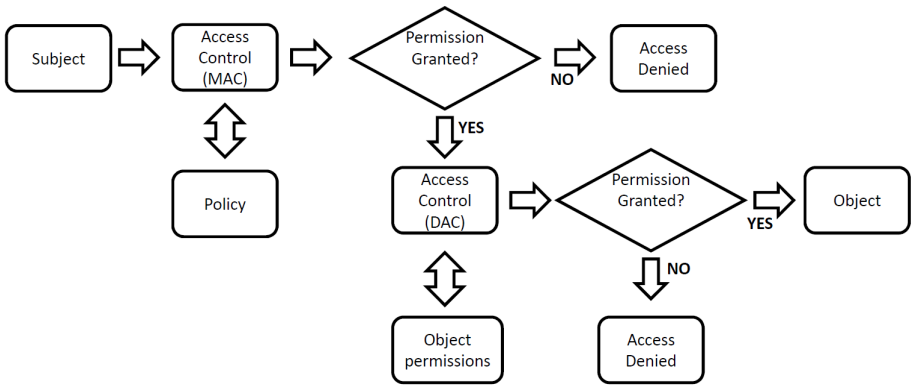
Another popular security model (in contrast with Bell La-Padula) is BIBA [20]. It is based on data integrity.

2.2.3 MAC and DAC

The DAC is the most popular Access Control mechanism and MAC is more secure than DAC but it is less flexible. For example, in MAC when two *Subjects* have to exchange some *Objects*, they have to ask the system administrator (little flexibility but significant security).

The DAC and MAC can coexist in the same system. For example, a system that supports MAC can be extended with DAC. In this case, when a *Subject* wants to access an *Object*, the *Subject* has to verify the permission to the *Mandatory Access Control* system.

The MAC decides according to the system policy (*Policy* file) whether to allow or not the *Subject* to access the *Object*. If the *Subject* can do that action on



*

Figure 2.8: *MAC and DAC*

the *Object* (according to MAC), the access request is given to DAC, otherwise the access is denied.

The DAC decides according to the system permissions whether to allow or not the *Subject* to access the *Object*. If the *Subject* can do that action on the *Object* (according to DAC), the access is granted, otherwise the access is denied (see Figure 2.8).

The use of DAC and MAC together improves the flexibility of the system. The MAC is often too rigid but the DAC is often too insecure. In order to balance these two aspects it is useful to have a system with both MAC and DAC. In this way, the MAC control ensures a solid secure system and DAC permits the correct flexibility where it is needed.

The Bell-La Padula model, for example, is very rigid and in order to improve its flexibility, it can be extended with DAC. In this case an access matrix is used in order to describe the actions that the *Subject* can do on the *Object*.

2.3 SELinux

Security-Enhanced Linux [49] (SELinux) was designed by the Linux community. It has been released by the United States National Security Agency (NSA) on December 22, 2000 under the GNU GPL license. SELinux has been merged in 2.6 series of the Linux kernel and it is used in several Linux distributions: Red Hat, Fedora and CentOS. Moreover, it is used in Android operating system since release 4.3.

SELinux is a Linux kernel module that provides Mandatory Access Control (MAC) support.

It is based on three types of policy model: *Type Enforcement* (TE), *Multi Category Security* (MCS), and *Multi Level Security* (MLS).

In SELinux, each process (*Subject*) and resource (*Object*) in the system is labeled with a *security context* (also known as *security label* or *label*). The

security context (see Listing 2.1) is composed by four parts: **user** indicates the user, **role** indicates the role, **type** is used in Type Enforcement (TE) model in order to define the policy rules and **range** is separated by two parts. The first represents the *security level* range (used by MLS) and the second represents the *category set* (used by MCS).

Listing 2.1: *SELinux security context*

```
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
   user     :   role     :    type    :    range
```

2.3.1 Type Enforcement (TE)

The *Type Enforcement* (TE) policy model is based on the *type* part of the *security context*. The TE behavior is based on the *policy rules* stored in the policy file.

The distinction between *Subject* and *Object* is dropped. In fact, each *Subject* and *Object* is associated with a *type* which can be indiscriminately used as *source* or *target* inside a *policy rule*.

All system resources (*Subject* or *Object*) are associated with a *class* which represents the kind of resource (i.e., file or process). Each *class* is associated with a set of *actions* that are applicable on that resources class. The file *class*, for example, is associated with read and write *actions*.

The policy file (used only by TE) is composed by *policy rules*. It permits to declare the *types* (the *type* part of *security context*) that will be used inside the policy file, using the following syntax: **type** $type\_name$.

Moreover, thare are many types of *policy rule* that we can use inside the policy file (We will present only those relevant to understand the next Sections).

The *typebounds* (see Linsting 2.2 for its syntax) defines that some *types* or domains (list of *bounded_domain*) will have equal or less permissions than another *type* or domain (*bounding_domain*). In Listing 2.2, the character * indicates the presence of zero or more of those elements.

Listing 2.2: *typebounds rule*

```
typebounds bounding_domain bounded_domain bounded_domain*;
```

The *type_transition* (see Listing 2.3 for its syntax) defines that a *type* (*source_type*) has to be changed into the *type* (*default_type*), when it does an action on another *type* (*target_type*) with class equal to *class*. The transaction has also to be allowed by the policy (using the *allow* rule, explained below).

Listing 2.3: *type_transition rule*

```
type_transition source_type target_type:class default_type;
```

The *AV rules* define what accesses are allowed for resources (*Subject* and *Object*). There are four types of *AV rule*: **allow** permits a set of *actions* from *source* to a *target*, **neverallow** does not permit the existence of the same *allow*, **dontaudit** stops the audit of that denied message (inside the AVC log), and **auditallow** reports (inside the AVC log) whether that event (access event) has happened.

Each *AV rule* has the structure shown in Listing 2.4:

rule : it represents a type of *AV rule*.

source : it represents the *type* of the *source*. (resource *type* that does a set of actions on resource *target*)

target : it represents the *type* of the *target*. (resource *type* that receives the actions)

class : it represents the *class* of the *target*.

actions : they represent a set of *actions* according to target *class*

Listing 2.4: *AV rule*

```
rule source target:class {actions};
```

In SELinux, an application or software can be asssociated with an SELinux module in order to ensure the system security. The SELinux module defines the requirements that are needed and the rules that have to be installed in the system, in order to use the application or software. When a software or application is downloaded, even the associated SELinux module is downloaded. In this way, during the installation, both software or application and its SELinux module are installed. An example of SELinux module is shown in Listing 2.5.

Listing 2.5: *SELinux module*

```
module MySQL 1.0;

require{
    type tempfs_t;
    type mysqld_t;
};

allow mysqld_t tempfs_t:file {open read write lock};
```

The SELinux module is composed by three parts. The **module** command identifies the module *name* and *version*. The module *name* has to be unique and the *version* is used during the version update.

The **require** block identifies which types, classes and roles have to exist in the system before the module can be installed. If one or more of them are not in the system, the module installation fails.

The last part contains the rules that have to be added to the system.

The SELinux module in Listing 2.5 defines a module named *MySQL* at version *1.0*. This module requires that in the system exist the types $tempfs\_t$ and $mysqld\_t$ before the module can be installed.

Moreover, the *AV rule* **allow** $mysqld\_t$ $tempfs\_t$: $file$ $\{openreadwritelock\}$**;** has to be added into the system.

2.3.2 Multi Category Security (MCS)

The *Multi Category Security* (MCS) policy model is based on the *range* part of the *security context*. It uses the part of categories (*category set*) in the *range*. The *range* part of the *security contex* (for the MCS perspective) can be seen as $s0 : c0.c255$, where $s0$ represents the *security level* (used by MLS) and $c0.c255$ represents the categories (used by MCS).

The categories in the *security context* can be specified in two ways: separating categories using comma (i.e $c1, c2$). The categories $c1$ and $c2$ will be assigned to the resource, and using a range (i.e. $c1.c3$). The categories from $c1$ to $c3$ will be assigned to the resource. In the last case, a dot notation is used in order to separate the border categories.

The *security level* sett to $s0$ (the lowest *security level*) is equivalent to not considering the MLS. In this way, it is possible to analyze the categories behavior independently from the MLS.

The *Multi Category Security* MCS permits to have more instances of the same *type*. In fact, the addition of categories to the resources (*Sources* and *Objects*) changes the system behavior. *Subjects* with the same *type* but with different categories might not have permission to access the same *Objects*.

SELinux has a set of 255 categories, from 0 to 254 (from $c0$ to $c254$). The categories can be defined inside the system and then they can be assigned to resources (*Subjects* and *Objects*). It is possible to define a label (i.e, NUC, EUR, US) for a group of categories and assign it to the resources.

Once the categories have been assigned, a *Subject* can access *Object* with some permissions (defined by TE) only if he has all categories of the *Object*. For example, in a system with the following categories: NUC, EUR, US; a *Subject* S_1 with *category set* $\{NUR, EUR\}$ can access an *Object* O_1 with *category set* $\{NUC\}$, but cannot access an *Object* O_2 with *category set* $\{NUR, EUR, US\}$ (see Figure 2.7).

The MCS does not use the policy file (it is used only by TE). The categories and its assignment to the resources are stored in other system files. The *policy rules* defined by TE do not influence the MCS behavior.

The *Multi Level Security* (MLS) is based on the *range* part of the *security context*. It uses the part of *security level* in the range. The *security level* in the *security context* is specified as follows: $s0 - s0$, where the first $s0$ represents the low level of security and the second $s0$ represents the high level of security.

MLS allows to define several types of security levels in the system and assign them to the resources (*Subjects* or *Objects*). They follow a hierarchical structure (see Figure 2.6) and they conform to the Bell-La Padula model (see Section 2.2.2).

In SELinux, when a *Subject* wants to do an action on an *Object*, he has to ask the permission to the Access Control System. The Access Control System decides according to the system policy (TE Policy), *category set* and *security level* of the *Subject* and *Object*, whether to allow or not to allow the Subject to access the Object. If the Subject can do that action on the Object (according to TE, MCS and MLS), the access is granted, otherwise the access is denied (see Figure 2.5).

In SELinux, a *Subject*, in order to access an *Object*, has to pass the check of all security models (TE, MCS and MLS). When an access is denied, a row with reject information will be written in AVC log. The AVC log stores information about denied accesses and other access event information.

2.4 Docker

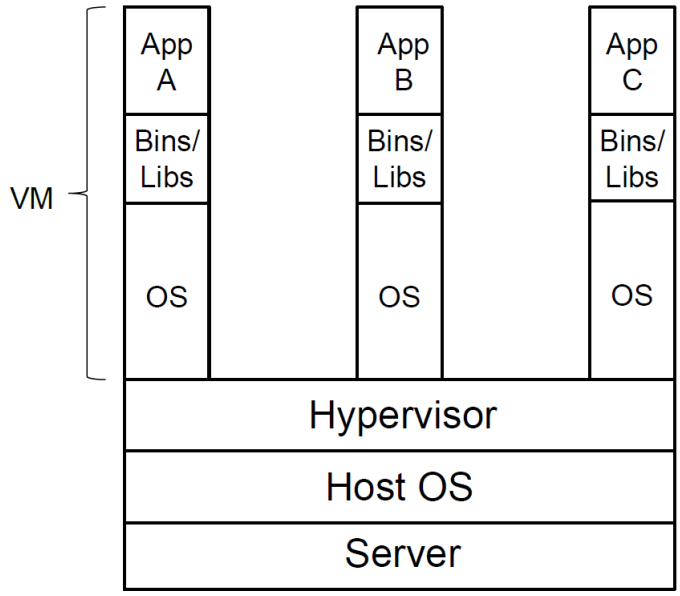


Figure 2.9: *Hypervisor Virtualization*

Docker [11] is an open source software which realizes container virtualization and it has been released as version 0.9, in March 2013. It is based on LCX container enviroment but with some changes. In fact, it uses a libcontainer library written in the Go programming language.

Docker is based on *Container Virtualization* which permits a lightweight but less secure virtualization than *Hypervisor Virtualization*.

Container Virtualization (see Figure 2.10) provides a lightweight virtualization that permits to have many virtual machines (containers) on the same server. The container is an application with its dependencies (Bins/Libs) and it runs on host operating system. In this way, a single host operating system is needed to run several containers. It is not necessary to have an operating system for each container. The container processes look like normal system processes for the host operating system and the containers management is made by *Containers Engine* (*Docker engine* for Docker). Moreover, a container is isolated from each other even though it shares dependencies with them. A container is also isolated from the host.

In *Hypervisor Virtualization* (see Figure 2.9) each virtual machine runs on its operating system (OS) separated by the host operating system. A virtual machine has its operating system (not only an application with its dependencies). Moreover, a virtual machine is isolated from each other (they have different OS) and it is isolated from the host using Hypervisor. The Hypervisor creates a strong isolation between host OS and virtual machine operating systems. In this way, the virtualization becomes heavier (many operating systems) but more secure (presence of Hypervisor) than *Container Virtualization*.

Docker permits to *build*, *ship* and *run* any application. It permits to create an application (*build*), to ship the application from development to testing and distribution enviroment (*ship*) and to run the application on several platforms (*run*). A *Docker application* is built from a *Docker file*. A *Docker file* is composed by some instructions which define how to the *Docker image* of that application has to be created (an image can refer to other images). A *Docker*

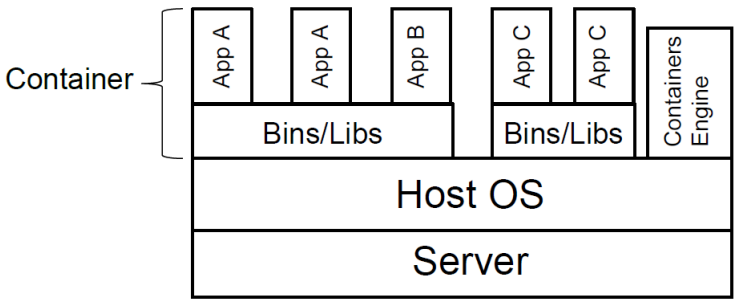


Figure 2.10: *Container Virtualization*

image is a *Docker application* that is not running and it can be run in order to create a *Docker container* of that image. A *Docker container* is a *Docker image* that is running (it can be composed by different images). It is possible to run many *Docker containers* from the same *Docker image* and when a *Docker container* has been run, the associated *Docker application* can be used. A *Docker application* can be seen as a container that can be moved from a platform to another without problems.

Moreover, Docker provides a tool named *docker compose*. The *docker compose* permits to create a virtualization system composed by containers. The containers settings and their links (how the containers are linked) are indicated in a specific file executable by *docker compose* tool. In this way, a system administrator can write all system settings in this file and then this file can be executed by *docker compose* in order to realize the whole system (it is not needed to run the containers one by one).

2.4.1 Docker Architecture

The Docker Architecture is composed by three parts: *Docker Client*, *Docker Engine* and *Docker Hub* (see Figure 2.11).

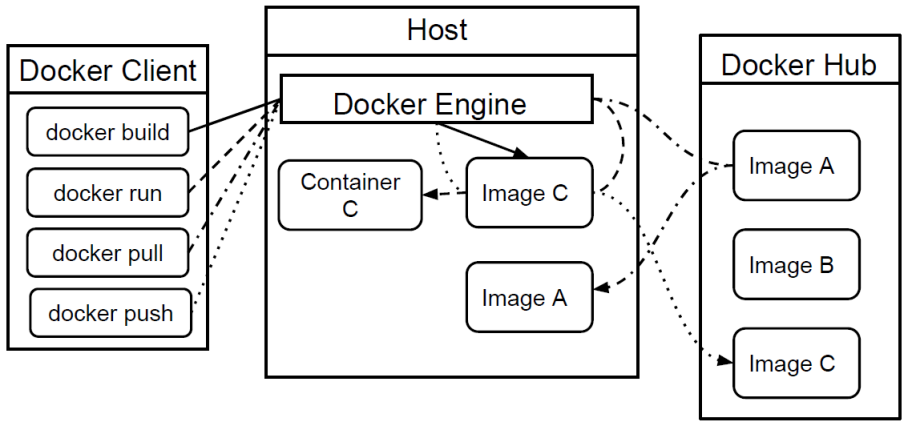


Figure 2.11: *Docker Architecture*

The *Docker Client* provides a user interface for interacting with *Docker Engine* (*Containers Engine* in Figure 2.10). The *Docker Engine* permits to manage the Docker enviroment: it permits to manage containers and images, inside the *Host* (*build* and *run*), and to download and upload (*pull* and *push*) the images from *Docker Hub*. The *Docker Hub* is a docker image repository where the users can share their images. A user can upload his images and can download the images uploaded by other users.

The *Docker Client* communicates with *Docker Engine* (also known as Docker daemon) and provides to users an interface with several commands.

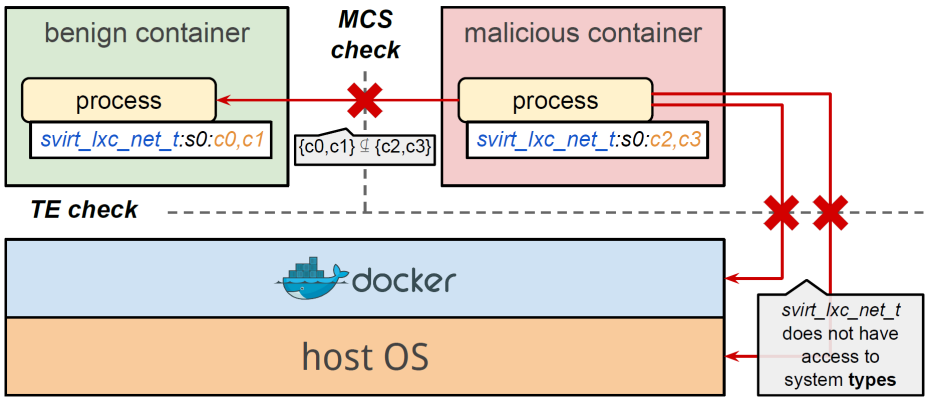


Figure 2.12: *Docker Securty*

Some of these commands are: *docker build*, *docker run*, *docker pull* and *docker push*. An example of their behavior is shown in Figure 2.11. The *docker build* command permits to build a *Docker image* from its *Docker file*, inside the Host. In Figure 2.11, *Image C* has been built (from its *Docker file*) and stored in the Host. The *docker run* command permits to build a *Docker image* inside the host and creates a *Docker container* for that image. In the example, the *Image C* has been run and the *Container C* has been created. The *docker pull* command permits to download a *Docker image* from *Docker Hub* to the Host. The example shows that the *Image A* has been downloaded from *Docker Hub*. The *docker push* command permits to upload a *Docker image* from the Host to *Docker Hub*. Figure 2.11 shows that the *Image C* has been uploaded from the Host. The *Docker Client* takes the input commands from the users and sends the commands to *Docker Engine*. In this way, the *Docker Client* and the *Docker Engine* can be hosted by different servers.

2.4.2 Docker Security

Docker security [22] will be described in this Section only explaining the security aspect useful to understand the next Sections.

Docker uses two kinds of security, offered by SELinux to ensure the isolation of Docker containers: *Type Enforcement* (TE) to protect the host from the containers and *Multi Category Security* (MCS) to protect one container from other containers (see Figure 2.12).

The host system is protected by *Docker containers* using *Type Enforcement* (TE). The TE security model assigns to each process and object inside the containers a *security context* with a specific *type*. The processes have a *type* equal to $svirt\_lxc\_net\_t$ and the objects have a *type* equal to $svirt\_sandbox\_file\_t$. This default behavoir can be modified indicating a specific *type*, which will be

used for running the container.

A process inside a *Docker container* can read, execute and use some files and resources of the host system. The use of these files and resources is permitted in order to allow the *Docker container* to perform the basic operations. Moreover, a process inside a *Docker container* can write (according to TE) all *Objects* (files and resources) with *type* equal to $svirt\_sandbox\_file\_t$ (all files inside the *Docker containers*).

The *Type Enforcement* (TE) protects the host system from the containers, but it does not protect a container from other containers (using only TE, a container could access the files of another container).

The *Multi Categoty Security* (MCS) is used in order to protect a *Docker container* from other *Docker containers*. The processes and *Objects* (files and resources) inside the same *Docker container* have a *security context* with the same *range* and the processes and *Objects* inside different *Docker containers* have instead *security context* with different *range*. For example, in Figure 2.12, the *malicious container* has a *security context* with *range* equal to $s0 : c2, c3$ and the *benign container* has a *security context* with *range* equal to $s0 : c0, c1$. According to MCS, the *malicious container* cannot access *benign container* and the *benign container* cannot access *malicious container*. The containers are isolated one from the other. Each *Subject* (process) and *Object* (file or resource) inside a *Docker container* has a *range* (of the *security context*) with *security level* equal to $s0$ (the MLS is not used) and a *category set* composed by a pair of categories (i.e., $c0, c1$). This pair of categories is generated at random in order to avoid the same assignment of categories to different *Docker containers*. This is the default behavior for Docker, but specifying, it permits to run a container with a different number of categories.

Docker permits to run two types of containers: the containers that can do actions to other containers (we will refer to these containers as *Active Containers*) and *Data volume containers* that are used only for storage and cannot access other containers (we will refer to these containers as *Passive Containers*). The first ones are run with the default settings shown above. The second ones are run by default without using categories. For example, a *Data volume container* is run with a *security context* that has a *range* equal to $s0$: (both MLS and MCS are not used). *Data volume containers* are used for sharing between containers.

Moreover, Docker permits to share directories between host and containers (mount a host directory as a data volume). A host directory can be mounted in two ways: in order to be shared with only one *Docker container* and in order to be shared with many *Docker containers*. In the first case, all *Objects* inside the directory will be mounted with the same *security context* of the *Objects* that are within the *Docker container*, with which the directory will be shared. In the second case, all *Objects* inside the directory will be mounted with a *security*

context that has a *type* equal to $svirt\_sandbox\_file\_t$ and an empty *category set* (the MCS is not used). In the last case, for MCS perspective, the directory can be accessed by all containers and not only by really authorized containers.

The use of a *security context* with the same *type* ($svirt\_lxc\_net\_t$ for *Subjects* and $svirt\_sandbox\_file\_t$ for *Objects*) for all containers and the non-use of the *category set* for *Data volume containers* and some shared directories could created security problems for Docker (the processes inside the *Docker containers* have more privileges than the privileges they need).

In this direction, we studied some improvements for Docker Security with the goal to satisfy the *principle of least privilege*[1]. We considered the Docker used in Fedora Linux distribution. The Docker Security in Fedora Linux is implemented with SELinux Mandatory Access Control (see Section 2.3).

We took into account two SELinux policy modules: *Type Enforcement* (TE) and *Multi Categories Security* (MCS). We did not consider *Multi Level Security* (MLS) because it is not used in Docker and it is rarely used in systems that use SELinux. A study in this direction would have limited impact for Docker Security improvements.

Our first study analyzed the addition of SELinux modules for the Docker image. The work has shown how to better manage TE labels, to provide a better management of privileges (see Section 2.5).

Our second study analyzed a different management of SELinux categories in Docker. It has shown how a better assignment of SELinux categories improves Docker Security (see Section 2.6).

2.5 DPM: Docker Policy Module

Table 2.1: *Validation of the AVC rules in a DPM*

	$\tau \in BASE$	$\tau \in DPM$
$\sigma \in BASE$	(A) **INVALID** *threat for the system*	(B) **OK / INVALID** *based on typebounds*
$\sigma \in DPM$	(C) **OK / INVALID** *based on typebounds*	(D) **OK**

Docker runs all containers with the same SELinux *types* ($svirt\_lxc\_net\_t$ for *Subjects* and $svirt\_sandbox\_file\_t$ for *Objects*) and this is a serious limitation. In fact, we have to grant $svirt\_lxc\_net\_t$ the upper bound of the privileges that a container could ever need. For example, since different applications operate on different network ports, $svirt\_lxc\_net\_t$ is allowed to listen to and communicate over all the network ports [70]. Specializing the *type* per con-

[1] principle of least privilege: it requires that in a computer enviroment, any subjects (e.g. users or process) must be able to access only the information and resources that are necessary for its legitimate purpose.

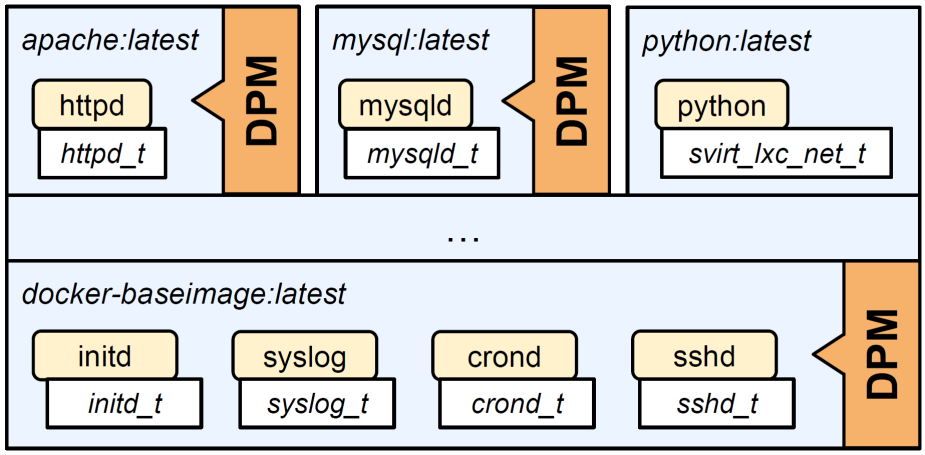


Figure 2.13: *Processes running in three Docker containers (apache, mysql and python), using specific SELinux types defined in the DockerPolicyModules embedded in the images.*

tainer (or even per process) would permit to tighten the security of Docker containers.

Docker already offers the user the ability to start the processes in a container with a different SELinux *type* (i.e., specifying it, in the *run* command). However, in this case the user is in charge of defining a suitable extension to the policy. Recently, an SELinux policy for the *Apache* $httpd$ container has been proposed by Daniel Walsh [70]. When the policy is installed, the container can be run with the specific type.

Although it is possible to start containerized processes with specific SELinux *types*, there are still limits to the applicability of this concept. It is reasonable to expect that many users will either be unfamiliar with the SELinux syntax and semantics, or do not know how to compile and install a policy module.

We propose a solution able to introduce specific SELinux types for different containerized processes in a transparent way for the user. This is based on Docker allowing image maintainers to ship an SELinux policy module together with their images. The module will be installed in the host system and defines the types that will be associated with the processes in the image. These modules are named *DockerPolicyModules* (DPM) and are SELinux modules that must also satisfy the properties defined in the following (shown in Table 2.1 and explained below), in order not to represent a threat for the host system. The DPM for an image will be specified in the *Dockerfile* and embedded in the image metadata at build-time. In order to run containerized processes with specific SELinux *types*, the image maintainer can label the binaries in the image with specific *types*, and write a type transition rule. In this way, when the binary is executed, the process is assigned the SELinux *type* defined in the rule.

Even if we have multiple processes running in the same image (e.g., the widely adopted *docker-baseimage* runs *init*, *syslog*, *cron* and *ssh*), it is possible to execute them with different SELinux labels. When a Docker container consists of different images, all the DPMs for the images that compose the container will be installed. This makes available also the SELinux *types* for processes in the parent images. Figure 2.13 represents different Docker containers with custom *types* provided by their DPM (The *python:latest* container does not use DPM but the other containers use it).

In order to avoid the possible threats that can emerge from letting Docker images install SELinux modules in the host system, we need to analyze the cases that derive from the combination of the system policy and a DPM. Due to the fact that each SELinux rule has a source (σ) and a target (τ) type, and they can be defined either in the system policy or in the DPM, we have four possible scenarios, described in Table 2.1.

A: the DPM must not change the system policy and can only have an impact on processes and resources associated with the DPM itself. Since containers can not be trusted a priori, it is imperative that the provided DPM does not have an impact on privileges where both σ and τ are system types;

B and **C**: new types defined in a DPM must always operate within the boundaries defined by the $svirt\_lxc\_net\_t$ *type*. The SELinux *typebounds* rule is used to confine the *types*, imposing an upper bound to the privileges that a *type* defined in a DPM can request. If a DPM defines a type not typebounded by $svirt\_lxc\_net\_t$, or a privilege not compliant with the *typebounds* rule, it is considered invalid;

D: a DPM provides the flexibility of defining multiple types with different privileges so that the container, according to the functionality in use, may switch to the one that represents the *least privilege* domain needed to accomplish the current task. This permits to limit the abuse that may derive from the exploitation of internal vulnerabilities and to tighten the overall container security.

The *Docker Hub* must ensure that the DPM of any uploaded image satisfies the requirements expressed in Table 2.1. Any image with a DPM not compliant with the above rules will be rejected. To protect the client from a compromised *Docker Hub*, a pre-processing phase will be added to Docker download and update routines in order to verify, before installing it, that the DPM does not represent a threat to the system.

SELinux is a sound security solution and its support for policy modules has already proved to be a significant enhancement in several services. The adaptation to Docker of the support for policy modules will allow the specification of SELinux domains for the different images, leading to an increase of security in Docker. We do not assume that all the image maintainers will include a *DockerPolicyModule* in their images, but the ones who are aware of the benefit

that SELinux provides will certainly appreciate the proposed extension.

2.6 Category Minimization using MCS

The MCS in Docker is used in order to isolate the *Docker containers*. Each container can be run with a set of categories. A container can access another container only if it is allowed by the MCS.

The goal of this study is to improve the isolation of *Docker containers*, improving the use of MCS in Docker. This can be realized running the containers only with the categories that we need according to the system policy and *principle of least privileges*.

In order to improve the MCS management in Docker, we studied a solution to minimize the number of categories used by the system, according to a given system policy. We called this problem *Category Minimization*.

The use of this technique permits to have a secure and scalable system. The *Docker containers* have only the needed categories in order to ensure their operation and the categories assigned with this accuracy improve the system security. The categories assigned do not allow containers to do actions not useful for its operation. Moreover, the system scalability is improved. When the system is modified, the new system policy is calculated and applied to the system. In this way, the system security is ensured. This technique also permits to save system resources (it uses the minimum number of categories that are needed) which will remain available. Moreover, the *Category Minimization* problem that will be described for *Docker containers*, can also be used, in other cases, when the system resources have to be assigned in order to satisfy the policy.

Definition 2.6.1 (Category Minimization)**.** *Let S be a system with its elements $e \in E$, where E is a set of system elements. Find an assignment of categories to system elements, according to the following rules:*

- *The system has to satisfy the* principle of least privilege
- *The categories assigned to system elements have to satisfy the system policy*
- *The system has to use the minimum number of categories in order to satisfy the system policy*

Before the theoretical aspect of the problem, we will present how we solved the problem and describe an implementation that demostrates the feasibility of our solution.

In our analysis, without loss of generality, we consider only *Multi Category Security* (MCS), since TE and MLS do not influence our analysis.

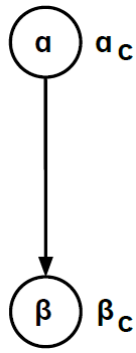


Figure 2.14: *Dominance Relation*

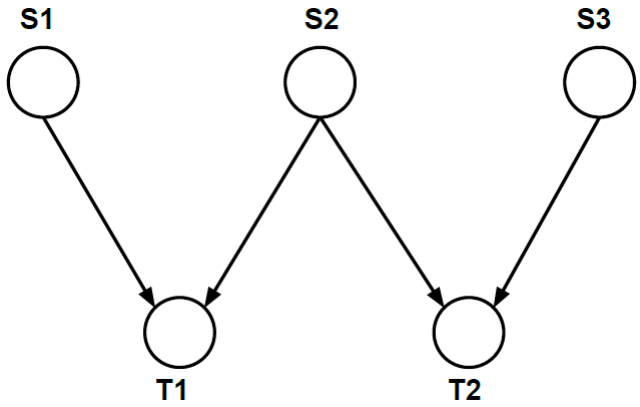


Figure 2.15: *System*

In fact (as explained in Section 2.3), the MCS and TE behaviors are independent. The MCS behavior is based on SELinux categories and TE behavior is based on SELinux types.

A Docker container is run with some categories and all processes (Subjects) and files (Objects) inside it are associated with the same categories. From the MCS perspective only the categories are considered. When we say that a Subject (or *Active Container*) can access an Object/Target (*Data Volume Container* or *Passive Container*), we mean that a process inside *Active Container* can access a file inside *Passive Container*.

2.6.1 Category Minimization problem

The *Category Minimization* problem has been analyzed using *Multi Category Security* (MCS). The MCS allows or does not allow the access to resources according to the *Subject* and *Object* categories. In order to formalize this aspect, we introduced the concept of *Dominance Relation* as follows:

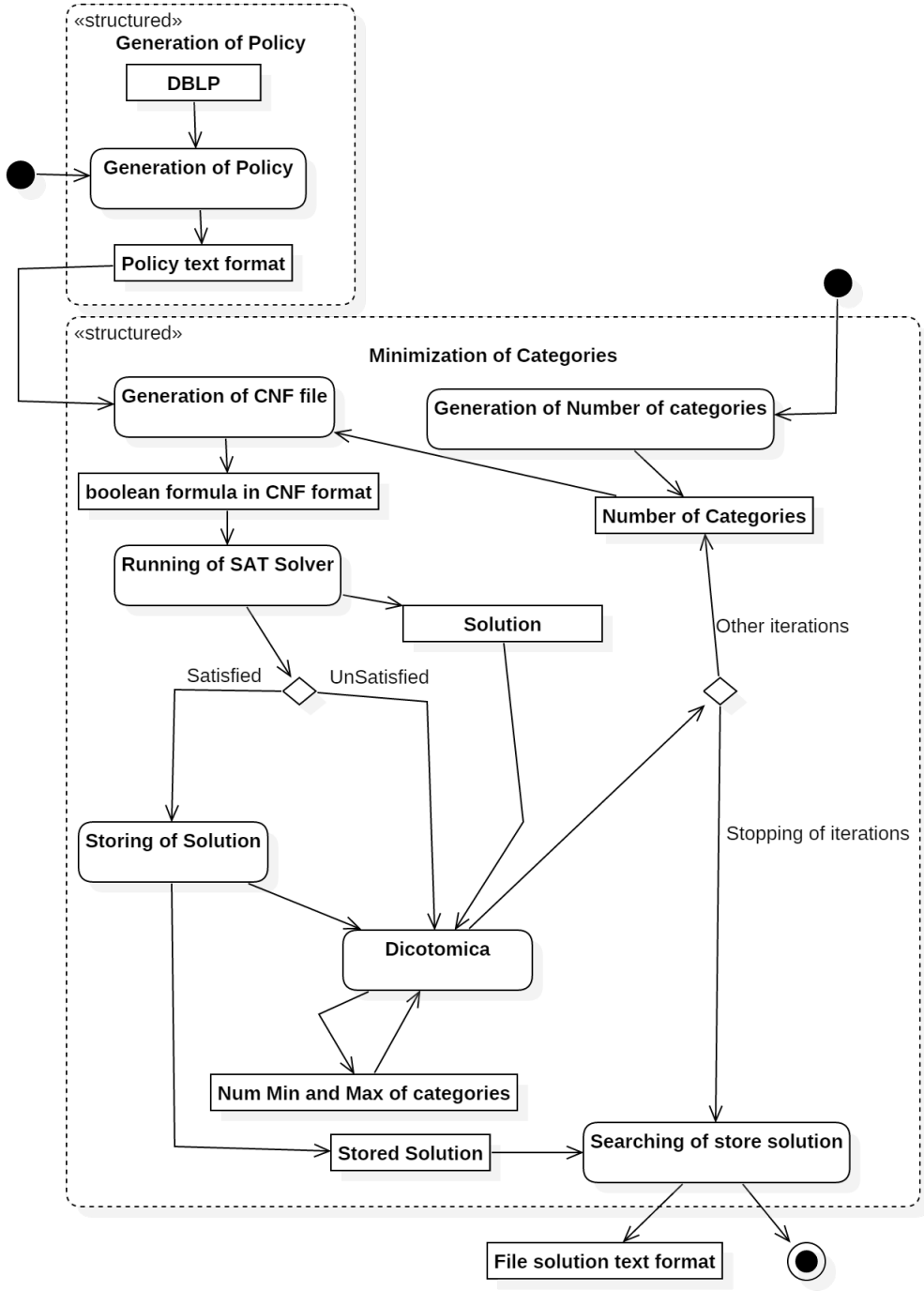


Figure 2.16: *Software Architecture*

Definition 2.6.2 (Dominance Relation)**.** *Let E be a set of elements belonging to the system. Let C be a set of categories that can be assigned to elements. Let* $e_1 \in E$ *be an element in the System,* $e_2 \in E$ *another element in the System,* $C_1 \subseteq C$ *categories assigned to* e_1 *and* $C_2 \subseteq C$ *categories assigned to* e_2*. We say that* e_1 *dominates* e_2 *iff* $C_2 \subseteq C_1$*. We represent this relation of dominance like* $< e_1, e_2 >$*, where* e_1 *is the dominant element and* e_2 *is the dominated element.*

In Figure 2.14 the *dominance relation* is shown between elements α and β. The element α is the dominant element ($\alpha_c \subseteq C$ represents the set of categories assigned to element α) and element β is the dominated element ($\beta_c \subseteq C$ represents the set of categories assigned to element β). The arrow indicates the direction of the relation, from dominant to dominated element. The *dominance relation* ($< \alpha, \beta >$) indicates that element α can access element β ($< \alpha, \beta > \implies \beta_c \subseteq \alpha_c$).

The *Dominance relation* is transitive. In order to formalize this concept, we define the *direct dominance relation* and the *indirect dominance relation* (transitive property of *Dominance relation*).

Definition 2.6.3 (direct dominance relation and indirect dominance relation)**.** *Let E be a set of elements belonging to the system. Let C be a set of categories that can be assigned to elements. Let* $e_1 \in E$ *be an element in the System,* $e_2 \in E$ *another element in the System,* $C_1 \subseteq C$ *categories assigned to* e_1 *and* $C_2 \subseteq C$ *categories assigned to* e_2*.* $|e_1|$ *be the number of categories of* e_1 *and* $|e_2|$ *be the number of categories of* e_2*.*

We say that an element e_1 *dominates an element* e_2 *with a* direct dominance relation *iff* $C_2 \subseteq C_1 \wedge (|e_1|{=}|e_2| \vee |e_1|{=}|e_2| + 1)$*.*

We say that an element e_1 *dominates an element* e_2 *with an* indirect dominance relation *iff* $C_2 \subseteq C_1 \wedge |e_1|{=}|e_2| + n$*, with* $n > 1$*.*

A system that uses MCS and composed by several elements creates a complex lattice structure of categories with many *dominance relations*. Figure 2.7 shows a system with several elements (*Subjects* and *Objects*) and with three categories (*NUC*, *EUR* and *US*). The circles represent the system elements (a circle represents all elements of the system with the same categories). The labels assigned to elements represent the categories of those elements and the label $\{\}$ represents the elements without categories. The elements without categories are not protected by MCS and can be accessed by all elements of the system (from the MCS perspective). In Figure 2.7 an element with an arrow towards another element can access that element (*direct dominance relation*). For example, an element with *category set* equal to $\{$*NUC,EUR,US*$\}$ can access an element with *category set* equal to $\{$*NUC,EUR*$\}$. Moreover, an element can access all elements reachable from it by a path (set of arrows that join an element with another). For example, an element with *category set* equal to

{*NUC,EUR,US*} can access an element with *category set* equal to {EUR} (*indirect dominance relation*).

The lattice structure has the *join* and *meet* properties which represent important aspects for our model. In Figure 2.7, an element that can access some elements has a set of categories that represent a *join* of those elements. For example the elements with categories {*NUC,US*} is the join element for elements with category {*NUC*} and {*US*}. In the same way, when an element can be accessed by more elements, it has to be the *meet* element of those elements. For example, In Figure 2.7 the element with category {*EUR*} is the meet element of the elements with categories {*NUC,EUR*} and {*EUR,US*}.

In Docker system, the categories or groups of categories are not replaced with a name (they are used with the default name given by SELinux). The categories *NUC*, *EUR* and *US* shown in Figure 2.7 can be used in Docker as $c0$, $c1$ and $c2$. In the following, we will use the categories in the same way as they are used in Docker.

Our study solves the *Category Minimization* problem for complex systems. We can find a solution of the *Category Minimization* problem for the system shown in Figure 2.7 and also for more complex systems but for a better description, we will show our study using a simple system. We will use the system shown in Figure 2.15, where the circles represent the elements, the arrows represent the *dominance relation* (they are all *direct dominance relations*) and the labels represent the element name.

We solve the *Category Minimization* problem using a SAT Solver and defining the policy rules in CNF format. The Boolean Satisfiability Problem (SAT) determines if a way exists to satisfy a given boolean formula. It tries to substitute all variables of boolean formula with value TRUE and FALSE in order to find if a combination of variables exists for which the boolean formula is TRUE. If a way exists to have the boolean formula equal to TRUE, the SAT problem is satisfiable, otherwise the problem is unsatisfiable (since the SAT problem is NP-complete and its resolution could take more time, it is possible to set a time-out for problem resolution. If a solution is not found before the time-out, the problem is considered unsatisfiable).

We expressed our boolean formula in CNF format. Conjunctive normal form (CNF) is a conjunction of clauses, where a clause is a disjunction of literals. A literal is an atomic formula or its negation. A CNF formula can be also seen like the AND of ORs. We create the boolean formula assuming to assign a fixed number of categories to the elements. For example, we try to use the categories from $c0$ to $c3$. In this way, four variables which represent the categories from $c0$ to $c3$ are assigned to each element. The variables represent the categories assignable to each element (the elements have a variable for each category that could be assigned to them). For example, category $c0$ of element $e1$ will be represented by a variable named $e1\_c0$. When the SAT problem

is satisfiable, the variables with *value* equal to *TRUE* represent the categories that have to be really assigned to that element and the variables with *value* equal to *FALSE* represent the categories that have not to be assigned to that element. For example, if variable $e1\_c0$ is TRUE, it means that category $c0$ has to be assigned to element $e1$ and when variable $e1\_c0$ is FALSE, it means that category $c0$ has not to be assigned to element $e1$. In the boolean formula other variables are also used that are only useful for the problem definition. Their values are not useful for the final solution. Moreover, when the problem is unsatisfied, the value of the variables (all variables) are not considered because the problem does not have a solution (more categories may be needed to solve the problem).

The boolean formula is built from policy rules following three properties: (i) *Sources* can access *Targets* (according to the policy), (ii) *Sources* cannot access *Targets* (according to the policy), (iii) *Sources* cannot access *Sources*

There are two other properties that are implicit and so they are not expressed in the boolean formula: (i) *Targets* cannot access *Sources* and (ii) *Targets* cannot access *Targets*

The *Targets* are passive elements (i.e., *Data Volume Containers*) and so they cannot access other elements.

In the system shown in Figure 2.15 there are three Sources $S1, S2, S3$ and two Targets $T1, T2$ and the policy rules associated with the system (see Listing 2.6) allow $S1$ to access $T1$ (row 1), $S2$ to access $T1$ and $T2$ (row 2), $S3$ to access $T2$ (row 3). This policy configuration is rapresented by arrows (*dominance relations*) in Figure 2.15. .

Listing 2.6: *System policy*

```
1 S1:T1
2 S2:T1,T2
3 S3:T2
```

The system policy rules (i.e., policy rules shown in Listing 2.6) are converted into a boolean formula which will be the input of the SAT Solver.

The boolean formula is written in CNF format and is composed by several logical formulas joined by AND ($\wedge$) operator. We will explain the structure of the boolean formula showing the logical formulas created in order to satisfy the three properties shown above. In the logical formulas in order to explain their structure, we will use only category $c0$ (the other categories will be implied). The categories really used in the boolean formula depend on the categories that we fix for each element (i.e., if we fixed four categories for each element, the categories used in the logical formulas will be the categories from $c0$ to $c3$). In order to explain the boolean formula, we will refer to Listing 2.6.

The property *Sources can access Targets* (the first ones) defines that a *Source* can access a *Target* only if it is allowed by the policy. A *Source* in

order to access a *Target* has to have all the categories of the *Target*. For example, considering row 1 of Listing 2.6 and taking into account only category $c0$, if *Target* $T1$ has category $c0$ then, even *Source* $S1$ has to have category $c0$. This is defined in the boolean formula as follows: $(\neg T1\_c0 \vee S1\_c0)$, where $T1\_c0$ represents the category $c0$ of the *Target* $T1$ and $S1\_c0$ represents the category $c0$ of the *Source* $S1$. If *Target* $T1$ has other categories, the formula is extended adding (in AND) the other logical expressions, in a similar way (in the new logical expression the category $c0$ has to be replaced with the other categories).

The property *Sources cannot access Targets* (the second one) defines that a *Source* cannot access a *Target*. A *Target* in order to not be accessible by a *Source* has to have at least one different category than the *Source*. For example, considering Listing 2.6, *Source* $S1$ cannot access *Target* $T2$. The property is defined in the boolean formula using two logical expressions. The first expression (taking into account only category $c0$) is: $< 1 > (T2\_c0 \vee \neg S1\_T2\_c0) \wedge (\neg S1\_c0 \vee \neg S1\_T2\_c0)$, where $T2\_c0$ represents category $c0$ of *Target* $T2$, $S1\_c0$ represents category $c0$ of *Source* $S1$ and $S1\_T2\_c0$ is an auxiliary variable useful only for the problem definition that is used in order to write the boolean formula in CNF format. When its value is equal to TRUE, it means that *Source* $S1$ has not category $c0$ and *Target* $T2$ has category $c0$. If *Target* $T2$ has other categories, the $< 1 >$ formula is extended adding (in AND) other logical expressions similar to $< 1 >$ (In the new logical expressions category $c0$ has to be replaced with the other categories). Moreover, all auxiliary variables are joined (using ORs operator) in another logical expression and considering to use the categories $c0$ and $c1$, the expression is: $< 2 > (S1\_T2\_c0 \vee S1\_T2\_c1)$ (if other categories come into play, their auxiliary variables have to be added). The expressions $< 1 >$ and $< 2 >$ are joined using AND operator and the resulting expression permits to define the property. Moreover, when only category $c0$ is used, expression $< 2 >$ simply becomes as follows: $S1\_T2\_c0$.

The property *Sources cannot access Sources* (the third one) defines that a *Source* cannot access another *Source*. A *Source* in order to be not accessed by another *Source* has to have at least one different category than the other *Source*. For example, considering Listing 2.6, *Source* $S1$ cannot access *Source* $S2$. The property is defined in the same way as the property *Sources cannot access Targets*, taking into account *Source* $S1$ and *Source* $S2$.

The final boolean formula is the join (in AND) of the three logical formulas relating to the properties. The boolean formula generated and stored in a CNF file becomes the input for the SAT Solver. The SAT Solver generates in output a text file. The output defines the categories that have to be assigned to each system element (*Source* or *Target*) in order to satisfy the policy in input. For example, Listing 2.7 shows a probable assignment of categories for the system shown in Figure 2.15 according to the policy in Listing 2.6.

Listing 2.7: *System output*

```
1 S1:c0,c2
2 S2:c0,c1
3 S3:c1,c3
4 T1:c0
5 T2:c1
```

The categories assignment of Listing 2.7 satisfies all properties explained above. *Source* $S1$ can access *Target* $T1$ ($S1$ has category $c0$) but cannot access *Target* $T2$ ($S1$ does not have category $c1$) and the *Sources* $S2$ and $S3$ ($S1$ does not have the categories $c1$ and $c3$).

Source $S2$ can access *Targets* $T1$ and $T2$ ($S1$ has categories $c0$ and $c1$) but cannot access *Sources* $S1$ and $S3$ ($S2$ does not have category $c2$ and $c3$).

Source $S3$ can access *Target* $T2$ ($S3$ has category $c1$) but cannot access *Target* $T1$ ($S3$ does not have category $c0$) and *Sources* $S1$ and $S2$ ($S3$ does not have the categories $c0$).

Listing 2.6 shows a simple policy associated with the little system in Figure 2.15. Our technique has also been verified with more complex policies associable with larger system. In order to generate complex policies, we used DBLP, a computer science bibliography that provides open bibliographic information on major computer science journals and proceedings [53]. DBLP provides the information about authors and their publications in XML format. We used DBLP in order to create the input policy file in this way: the authors are the *Sources* and the publications are the *Targets*. An author (*Source*) can access his publications (*Targets*). The authors can be co-authors with other authors and both authors and co-authors (*Subjects*) can access the common publications (*Targets*). Each row of the policy file is composed by an author and its list of publications. The policy created is more similar with the real system policy than the policy created in random way. In a real system policy, usually, a group of *Sources* can access the same groups of resources (*Targets*). For example, in a company, the managers can access all files useful for the company management and these files cannot be accessible by the IT that can instead access other files useful for the technical aspects. This scenario can be represented very well by a policy generated using DBLP (all divisions will be represented by groups of co-authors that can access the common files (publications)) but not very well by a policy generated in a random way beacause the *Targets* accessible by the *Sources* are too sparse (the numbers of common *Targets* between the several *Subjects* do not reflect a real scenario). DBLP is useful to generate a complex and real policy similar to a real scenario and using it, we checked our technique for complex and real systems.

2.6.2 Software Architecture

We developed a software in order to verify our technique. The software is composed by two parts and its architecture is shown in Figure 2.16. The first part of the software (*Generation of policy*) takes in input the DBLP XML file and generates the policy in text format (i.e., see Listing 2.6). The policy generated will be the input for the second part of the software (*Minimization of Categories*). The second part of the software takes in input the policy in text format and generates (*Generation of CNF file*) the boolean formula (*boolean formula in CNF format*) according to the policy and the *Number of Categories* (the fixed number of categories which we are assuming to assign to the elements). The *Number of Categories* is generated, at first, in random way (*Generation of Categories*) and it is less than or equal to the number of the *Sources*. The *boolean formula in CNF format* will be the input for the SAT solver (*Running of SAT Solver*) which will generate in output a solution of the problem with a specific *Number of Categories* (if the problem is *satisfiable*) and when a solution of the problem with that *Number of categories* does not exist, the output will be *unsatifiable*. When the problem is *satisfiable*, the solution is stored and then, the *dichotomic search* algorithm [2] is applied in order to generate a new *Number of Categories* used to create a new *boolean formula in CNF file* (the new *bolean formula* is generated from the *policy text format* used also before and the new *Number of Categories*). When the problem is *unsatisfied*, the *dichotomic search* algorithm is directly applied (it does not exist a solution to store). The *dichotomic search* algorithm is applied in order to search the solution that uses the minimum number of categories to satisfy the input policy. The *dichotomic search* algorithm checks whether it is possible to generate a new *Number of Categories* to search a better solution. When the *dichotomic search* algorithm can generate a new *Number of Categories*, the software continuously generates a new *boolean formula in CNF format*, otherwise, the software checks whether a stored solution exist. If a stored solution exists, the software outputs the solution (i.e., see Listing 2.7) and terminates, otherwise the software outputs *unsatisfied* and terminates (a categories assignment that satisfies the input policy does not exist).

2.6.3 Experiment results

The experiment on our technique have been run using an Amazon EC2 instance. This instance (*m4.10xlarge*) has 40 virtual CPUs, 160 GiB of Ram and a dedicated bandwidth of 4000 Mb/s. The experimental results are shown in

[2] dichotomic search algorithm: it is an algorithm of binary search used on a set of ordered elements that starts comparing the middle element with searched element. If the elements are equal, the algorithm terminates; if middle element is lower than searched element, the algorithm continues on following elements; if middle element is greater than searched element, the algorithm continues on previous elements; if all elements are discarded, the algorithm terminates (the element is not found).

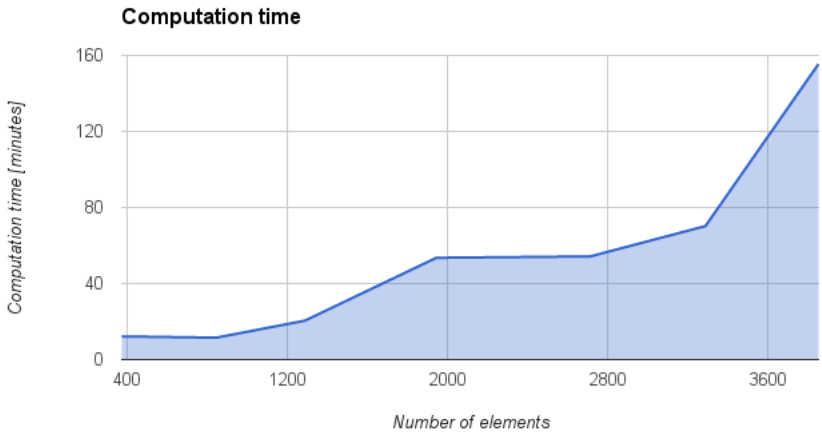


Figure 2.17: *Computation time in Minutes*

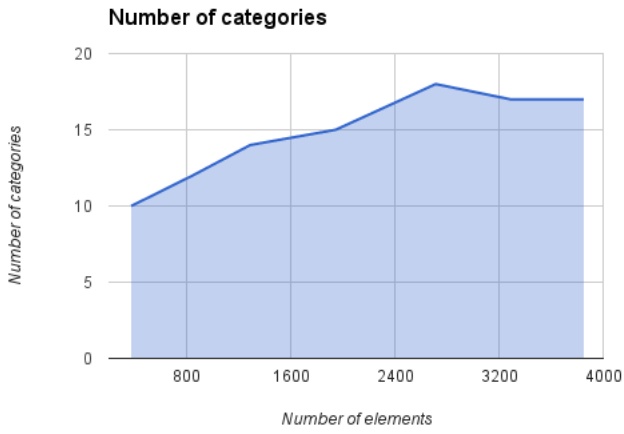


Figure 2.18: *Number of categories used by the system in order to satisfy the policy*

Figure 2.17 and 2.18. We executed the tests considering the number of the *Sources* present in the system. The range of *Sources* that we considered is between 50 and 600.

The experiments show that a policy with a number of elements (*Sources* and *Targets*) between 370 (50 *Sources* and 320 *Targets*) and 842 (100 *Sources* and 742 *Targets*) requires a computation time in order to find an assignment of categories according to the policy that is around 12 minutes and the number of categories used by the system is between 10 and 12. A policy composed by 1285 (200 *Sources* and 1085 *Targets*) elements requires a computation time of 20,47 minutes and the system has to use 14 categories in order to satisfy the policy. A policy composed by a number of elements between 1942 (300 *Sources* and 1642 *Targets*) and 2712 (400 *Sources* and 2312 *Targets*) requires 54 minutes of computation time and the system has to use a number of categories between 15 and 18 in order to satisfy the policy. A system with a policy composed by a number of elements between 3287 (500 *Sources* and 2787 *Targets*) and 3852 (600 *Sources* and 3252 *Targets*) has to use 17 categories in order to ensure system security according to the policy. The computation time grows faster than the previous case. A policy with 3287 elements is computable in about 71 minutes and with 3852 elements is computable in about 156 minutes.

The results show that our technique permits to use few categories even in systems that have a policy with many elements (*Sources* and *Targets*). For example, a system with a policy composed by 3852 elements can use 17 categories to ensure system security. The software requires few minutes to find a solution for policies with few elements (i.e., between 370 and 842 elements) and few hours to find a solution for policies with many elements (i.e., 3852 elements).

The computation times that we obtained show the efficiency of our technique considering that this type of tool is used in phase of system installation (in this phase the system structure is decided and the system policy is applied), when a waiting time of a few hours is not critical. We can say that the developed technique permits to ensure the security of a system saving many resources (*categories*) and the computation time needed to find a solution is not critical.

2.7 Related Work

The Linux Containers (as *Docker containers*) permit a lightweight virtualization but are less secure than hypervisor virtualization. For these reasons, several studies have been done in order to improve their security. An approach similar to Docker Policy Module (DPM) but for Android third-party apps has been proposed and the security implications involved have been studied [51], [17]. The Docker scenario appears to fit quite well with this proposal,

with the additional advantage that we expect image developers (as opposed to image users and app developers) to be more familiar with the role and impact of MAC policies.

In this research we presented some tecniques in order to improve the use of SELinux. SELinux is a Mandatory Access Control based on labels. The approaches based on labels is often difficult to undertand and to manage. Other techniques, based on MAC, have been implemented to permit the creation of policies in simple way: *TOMOYO* [39] and *AppArmor* [7].

AppArmor is based on file path and associates a security profile to each program in order to restrict its capabilities. It is used in Ubuntu Linux distribution and is supported by Docker. A framework named LiCShield [47] improves the security of Docker. It protects the Linux containers and their workloads building a security profile (based on AppArmor rules) and protects the execution of a given container tracing and analyzing the container executions. Moreover, a study about the Host security is Host-based Intrusion Detection Systems (HIDS) [35] which learns the usual behaviors of the application and detects unusual behaviors.

2.8 Evaluation of Security Enhanced Docker

Docker provides virtualization containers that permit a lightweight virtualization, but is less secure than other types of virtualization. It leverages Linux kernel security features such as *kernel namespaces* to isolate users, processes, networks and devices, and *cgroups* to limit resource consumption. Moreover, *Docker* in the Fedora distribution uses SELinux to ensure container isolation. The container isolation as realized in *Docker* allows containers to have more privileges than necessary for performing their work creating attack risk.

We presented two techniques in order to improve the use of SELinux in *Docker*: *Docker Policy Module* (DPM), which is based on SELinux *Type Enforcement* (TE), and *Category minimization problem*, which is based on SELinux *Multi Category Security* (MCS). These improvements permit more isolation for containers.

Docker Policy Module (DPM) provides SELinux module for Docker image. A module permits to have a specific *security context* for the associated image. Moreover, a module defines specific rules for the processes inside the containers that are run with that module. A module also permits to improve containers isolation assigning specific *security contexts* and restrictive rules to processes inside the containers.

The *category minimization problem* technique manages the categories in a better way than the default use of SELinux categories in *Docker*. Using this technique, each element in the system receives only the categories needs in order to perform its work and also *Data volume containers* are protected.

Moreover, the system uses the minimum number of category needed to ensure system security according to the system policy. The experiments showed that the *Category minimization problem* permits to save many resources (*categories*) which remain available for the system (1300 elements need only 14 categories). Moreover, the computation time shows the feasibility of this solution (1300 elements need 22 minutes). These techniques satisfy the *principle of least privileges* ensuring more secure systems.

CHAPTER 3

J-CO query language

3.1 Introduction

The buzzword *Big Data* is used with several possible meaning. The obvious one is "volume", but another meanings are related to the adverbs "variety" ([25, 44, 48]): complex analyses require to integrate several data sets coming from different sources of information.

Very often, these sources of information are *Open Data* portals, where public administrations publish data sets concerning several aspects of territories and citizenship. There is not a standard for those data sets: in spite of the fact that usually they are JSON collections ([45]) or CSV files or XML documents, every single data set has a specific structure, with specific field names, even though they describe similar topics. Often, these data sets contain geo-referenced information.

Other sources of (possibly geo-referenced) information could be descriptions of networks and (such as water networks, electricity networks, etc.) and environment descriptions (streets, buildings, etc.), possibly publicly available, that might be cross processed to discover useful information, or integrated with (possibly geo-referenced) Open Data.

These considerations motivate a large use of *NoSQL* databases [24, 38, 63], because traditional relational/SQL databases are unable to flexibly integrate so heterogeneous data sets. Nowadays, the most famous NoSQL DBMS is

MongoDB [18, 57]: it deals with collections of JSON objects, in such a way within the same collection objects with different structures can be gathered without any limitation.

However, the query language provided by MongoDB it is not easy to use for non programmers: it is strongly based on the object-oriented paradigm, and complex operations on collections are not easy to write. Moreover, it is even impossible to write operations that cross-combine two collections (unless an external program is written).
Furthermore, the geographical support is provided by means of *GeoJSON* [23, 26] and several functions that implements spatial operations on such a data format. The problem is that it is complicated writing complex operations on that involves several collections and, possibly, spatial operations.

To overcome this problems and reducing the distance between MongoDB and users (possibly geographers and analysts) we decided to define a novel flexible query language for heterogeneous collections of possibly geo-referenced JSON objects, named *J-CO* (which stands for JSON Collections). Our vision is the following: We want to provide users with a declarative query language, that is able to express complex query processes, that could be executed several times; the operators allow to directly work at the *collection level*, i.e., they express transformations on collections, for example, by filtering objects or aggregating objects in two or more collections; the operators natively deal with spatial representation and provide high level spatial operations. We were inspired by early works [21, 59] about a query language for heterogeneous, altough structured, collections of geo-referenced data.
Ambitiously, we can say that we are trying to repeat what happened with SQL in the 70s, i.e., enabling database technology for non-programmer users with a declarative query language.

J-CO is not complete, in the sense that we think about it as a continuously growing language: as far as new needs arise, new operators can be defined and added to the language. The approach is then open to new developments.

In this research, we present the basic and minimal operators we considered necessary in the J-CO query language, for which the data model and the execution model are devised. We will show the language and its use by means of a running example. This research is the first step on this research line, and our goal is to validate the approach, showing the fundamental operators and their applicability. In our future work, we will address various application areas, in order to identify specific needs and define new operators.

The remainder of this Chapter is organized as follows. in Section 3.3, we present the data model and a toy running example that will be exploited along with this Chapter. Section 3.4 presents the execution model on which J-CO relies; thus way, Section 3.5 can effectively introduce the operators. A complete example is presented in Section 3.6, to illustrate the potential application

of J-CO. Section 3.7 presents the implementation of the prototype and discuss experimental results. Finally, Section 3.8 discusses related works and Section 3.9 draws the conclusions.

3.2 Background concepts

This Section presents the background concepts useful to understand the next Sections. *Open Data* [12] is a data that can be freely used and ridistributed by anyone. The users have to cite the source and can share the source using the same license. *Open Data* have to be: *available and accessible* in a convenient and modifiable form (i.e., JSON), preferably using an internet download; *reusable and redistributable* and also mixed with other data. Moreover, they have to provide a license that permits the re-use and redistribution.

Open data are usually released in JSON format. JSON (JavaScript Object Notation) [9] is a lightweight text format for data exchange. A JSON object is a set of name/value pairs. An object starts with left brace ({), follows with a set of name/value pairs (i.e., $name : value$) separated by comma (,) and ends with right brace (}). Listing 3.1 shows an example. The term *value* can assume several types: *string*, *number*, *object*, *array*, *boolean* and *null* (according to programming language conventions). An *array* is structured as follows: $[value(, value)*]$, where * indicates the presence of zero or more of that element.

Listing 3.1: *JSON Example*

```
{
  "name":"nameA",
  "surname":"surnameA",
  "age":"20"
}
```

JSON is a standard also used for data storage and is supported by several databases. MongoDB [10], the database used in our tests, supports JSON format. MongoDB is an open-source document database. It support JSON standard for data storage and its record is a document which is a data structure composed by field and value pairs. MongoDB supports the data types supported by many programming languages and the use of document embedded and array (as values) reduce expensive joins providing high performance. Moreover, it supports a rich query language which requires little programming knowledge.

MongoDB also supports Geo-spatial data and Geo-spatial query (it provides some functions in order to query Geo-spatial data) using GeoJSON standard. GeoJSON [8] provides several structures to manage Geo-spatial data and between them, in J-CO query language, we use *GeometryCollection* in order to storage Geo-spatial information of our objects.

GeometryCollection is structured as shown in Listing 3.2.

Listing 3.2: *GeometryCollection Example*

```
{
  "type": "GeometryCollection",
  "geometries":[
                {
                  "type": "Point",
                  "coordinates": [100.0, 50.0]
                },
                {
                  "type": "LineString",
                  "coordinates": [ [101.0, 50.0],
                                   [102.0, 51.0] ]
                }
               ]
}
```

It is composed by a field *type* with value equal to *GeometryCollection* and a field *geometries* that is an array containing the geometry objects shown below.

The geometry objects can be: *Point*, *LineString*, *Polygon*, *MultiPoint*, *MultiLineString*, *MultiPolygon*.

Point cordinates are the longitude,latitude (see Listing 3.3 for example).

Listing 3.3: *Point Example*

```
{
  "type": "Point",
  "coordinates": [100.0, 50.0]
}
```

LineString coordinates are an array of positions (see Listing 3.4 for example).

Listing 3.4: *LineString Example*

```
{
  "type": "LineString",
  "coordinates": [ [100.0, 50.0],
                   [101.0, 51.0] ]
}
```

Polygon coordinates are an array of positions (at least four positions) where the first and last positions represent equivalent points (see Listing 3.5 for example).

Listing 3.5: *Polygon Example*

```
{
  "type": "Polygon",
  "coordinates": [[ [101.0, 0.0],
                    [102.0, 0.0],
                    [102.0, 1.0],
                    [101.0, 1.0],
                    [101.0, 0.0]
```

```
                ]]
}
```

MultiPoint coordinates are an array of positions and the *MultiPoint* object has a field *type* equal to *MultiPoint*. *MultiLineString* coordinates are an array of LineString coordinate arrays and the *MultiLineString* object has a *type* equal to *MultiLineString*. *MultiPolygon* coordinates are an array of Polygon coordinate arrays and the *MultiPolygon* has a *type* equal to *MultiPolygon*.

3.3 Data Model

The basic concept on which we rely is the one of *JSON* object. JSON (JavaScript Object Notation) is a de facto standard serialized representation for objects. Fields (object properties) can be simple (numbers or strings), complex (i.e., nested objects), vectors (of numbers, strings, objects).

As far as spatial representation is concerned, we rely on the GeoJSON standard. In particular, we assume that the geometry is described by a field named *geometry*, defined as a *GeometryCollection* objects type in GeoJSON standard. The absence of this top-level field means that the object does not have an explicit geometry.
As an example, consider the object with name `"buildingA"` reported in Listing 3.6. The `geometry` field describes the polygon representing the footprint of the building on the ground.

The following definition defines the concepts of *collection* and *Database*.

Definition 3.3.1 (Collections and Databases)**.** *A* Database db *is a set of collections* $db = \{c_1, \ldots, c_n\}$*. Each collection* c *has a name* c.name *(unique in the database) and an instance* Instance(c)$= [o_1, \ldots, o_m]$ *that is a vector of JSON objects* o_i*.*

Thus, we need operators (see Section 3.4) to transform collections and get new collections. Our language should satisfy the *closure property*.

Example 1 (Running Example)**.** *In order to illustrate the data model and, in the next sections, the execution model and the J-CO operators, we provide a running example.*

Suppose we have a sample database named ***ToyDB****. Within it, We have three toy collections: the first one is named* `Buildings` *(shown in Listing 3.6); the second one is named* `WaterLines` *(see Listing 3.7). Finally, the third one is named* `Restaurants` *(see Listing 3.8).*

Listing 3.6: *Collection* ***Buildings***

```
[{ "name":"buildingA",
   "city":"city A",
   "address":"address A",
```

```
   "geometry":{"type":"GeometryCollection",
              "geometries":[
                 {
                  "type": "Polygon",
                  "coordinates": [
                     [[100.0, 0.0],
                      [101.0, 0.0],
                      [101.0, 1.0],
                      [100.0, 1.0],
                      [100.0, 0.0] ]]
                 }

              ]

           }

 },

 {"name":"buildingB",
  "city":"city B",
  "address":"address B",
  "geometry":{"type":"GeometryCollection",
              "geometries":[
                 {
                  "type": "Polygon",
                  "coordinates": [
                     [[20.0, 0.0],
                      [21.0, 0.0],
                      [21.0, 1.0],
                      [20.0, 1.0],
                      [20.0, 0.0] ]]

                 }
            ]
      }
 }]
```

Listing 3.7: *Collection* ***`WaterLines`***

```
[{"name":"WaterLineA",
  "city":"city A",
  "geometry":{"type":"GeometryCollection",
              "geometries":[
                {
                 "type": "LineString",
                 "coordinates":
                    [[90.0, 0.0],
                     [103.0, 1.0],
                     [104.0, 0.0],
                     [105.0, 1.0]]
              }
        ]
```

```
},

{"name":"WaterLineB",
 "city":"city A",
 "geometry":{"type":"GeometryCollection",
           "geometries":[
              {
               "type": "LineString",
               "coordinates":
                  [[102.0, 10.0],
                   [103.0, 2.0],
                   [104.0, 1.0],
                   [102.0, -1.0]]
              }
       ]
},

{"name":"WaterLineC",
 "city":"city C",
 "geometry":{"type":"GeometryCollection",
           "geometries":[
              {
               "type": "LineString",
               "coordinates":
                  [[104.0, 10.0],
                   [105.0, 2.0],
                   [109.0, 1.0],
                   [110.0, -1.0]]
              }
       ]
 }
}]
```

Listing 3.8: *Collection* ***Restaurants***

```
[{"name":"RestaurantA",
  "city":"city A",
  "address":"address A"
 },

 {"name":"RestaurantB",
  "city":"city B",
  "address":"address C"
 },

 {"name":"RestaurantC",
  "city":"city C",
  "address":"address D",
  "geometry":{"type":"GeometryCollection",
           "geometries":[
              {
               "type": "Polygon",
               "coordinates":[
```

```
                    [[20.0, -10.0],
                     [21.0, -10.0],
                     [21.0, -11.0],
                     [20.0, -11.0],
                     [20.0, -10.0]]]
                }
            ]
        }
    }]
```

J-CO is able to query the informations inside ToyDB *database by using the operators shown in Section 3.5. We will use the* ToyDB *database and its informations, in the next Sections in order to explain J-CO query language.*

3.4 Execution Model

Queries will transform collections stored in MongoDB databases, and will generate new collections that will be stored again into these databases, for persistency. for simplicity we call such databases as *Persistent Databases*

Definition 3.4.1 (Query Process State)**.** *A* state s *of a query process is a tuple* $s = (tc, IR)$*, where* tc *is a collection named* Temporary Collection*. while* IR *is a database named* Intermediate Results database*.*

Definition 3.4.2 (Operator Application)**.** *Consider an operator* op*. Depending on the operator, it is parametric w.r.t. input collections (present in the persistent databases or in* IR*) and, possibly, an output collection, that can be saved either in the persistent databases or in* IR*.*
The applications of an operator op*, denoted as* $\overline{op}$*, is defined as*

$$\overline{op} : s \to s'$$

where both domain and codomain are the set of query process states. The operator application takes a state s *as input, possibly works on the temporary collection* $s.tc$*, possibly takes some intermediate collection stored in* $s.IR$*; then, it generates a new query process state* s'*, with a possibly new temporary collection* $s'.tc$ *and a possibly new version of the intermediate result database* $s'.IR$*.*

The idea is that the application of an operator starts from a given query process state and generates a new query process state. The *temporary collection* tc is the result of the operator; alternatively, the operator could save a collection as *intermediate result* into the IR database, that could be taken as input by a subsequent operator application.

Definition 3.4.3 (Query)**.** *A query* q *is a non-empty sequence of operator applications, i.e.,* $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$*, with* $n \geq 1$*.*

Table 3.1: *Type of operators*

	Operators
Start operators	GET COLLECTION OVERLAY COLLECTIONS JOIN COLLECTIONS MERGE COLLECITONS INTERSECT COLLECTIONS SUBTRACT COLLECTIONS
Carry on operators	FILTER GROUP BY SET INTERMEDIATE AS SAVE AS DERIVE GEOMETRY

Thus, the query is a sequence of operator applications; each of them starts from a given query process state and generates a new query process state, as defined by the following definition.

Definition 3.4.4 (Query Process)**.** *Given a query* $q = \langle \overline{op}_1, \ldots, \overline{op}_n \rangle$*, a query process* QP *is a sequence of query process states* $QP = \langle s_0, s_1, \ldots, s_n \rangle$*, such that* $s_0 = (tc : [], IR : \emptyset)$ *and, for each* $1 \leq i \leq n$*, it is* $\overline{op} : s_{i-1} \rightarrow s_i$

The query process starts from the empty temporary collection $s_0.tc$ and the empty intermediate results database $s_0.IR$. Thus, the GeCo query language must provide operators able to start the computation, taking collections from the persistent databases, while other operators carry on the process, continuously transforming the temporary collection and possibly saving it into the persistent databases. But the query could be complex and composed by several subtasks, thus the temporary collection could be saved into the intermediate results database IR. At this point, a new subtask can be started by the same operators that can start the query, which can take collections either from persistent databases or from the intermediate result database as input, giving rise to a new subtask.

For this reason, we identified two classes of operators (see Table 3.1): *start operators* and *carry on operators*, that will be described in the next section.

As a final consideration, observe that the reason why the intermediate results database IR is part of query process states is *isolation*: it exists only during the query process and in case of parallelism, each parallel process has its own intermediate results database.

3.5 J-CO Operators

J-CO operators can be classified into two groups, reported in Table 3.1: *Start operators* and *Carry on operators*.

Table 3.2: *Meaning of terms*

Terms	Meaning of terms
dbName	name of a persistent database
collectionName	name of a collection
fieldName	name of a field
value	value of a attribute
dbName.collectionName	collection inside a persistent database
collectionName.fieldName	field of a collection

Start operators can be used to start a new processing (sub)task in the query. Their inputs are collections coming from a persistent database or from the intermediate results databases *IR*. Their output is a new temporary collection computed from input collections.

Carry on operators can continue the processing (sub)task. They implicitly take the temporary collection and possibly produce a new version of it. Furthermore, they can store collections into the intermediate results database *IR*, or into a persistent databases.

Hereafter, we will make use of terms reported in Table 3.2 to introduce the syntax of operators and explain their behaviours.
Regarding the notation for the syntax of operators, we will make use of the $\star$ symbol to denote denote 0 or more repetitions and of the + symbol to denote 1 or more repetitions; square brackets denote optionality; the vertical bar | separates alternatives. For example, *([dbName.]collectionName)*+ denotes that a non empty list of collection names is required, where each one could come from a persistent database (when the optional *dBName.* is specified) or from the intermediate results database *IR* (when the optional *dBName.* is not specified).

3.5.1 Start Operators

GET COLLECTION

The **GET COLLECTION** operator permits to get a collection from a database (persistent or intermediate) and make it the new temporary collection. The syntax of the operator is:

```
GET COLLECTION [dbName.]collectionName;
```

When the *dbname* is specified, the JSON collection named *collectionName* is retrieved from the persistent database named *dbname*; otherwise, it is retrieved from the intermediate result database *IR*.

OVERLAY COLLECTIONS

The **OVERLAY COLLECTIONS** operator makes the geospatial join between two collections; the result becomes the new temporary collection. The in-

put collections can come from a persistent database or the intermediate results database *IR*. The syntax of the operator is hereafter.

```
[LEFT|RIGHT|FULL] OVERLAY COLLECTIONS [dbName.]collectionName1,
                                      [dbName.]collectionName2
[ON selectionCondition]
KEEP (INTERSECTION|RIGHT|LEFT|ALL);
```

The output of the **OVERLAY COLLECTIONS** operator is a JSON collection containing the result of the geospatial join between *dbName.collectionName1* (left collection) and *dbName.collectionName2* (right collection). For each object l_i in the left collection and an object r_j in the right collection, an object $o_{i,j}$ appears in the output collection if the geometries of l_i and r_j overlaps (intersect), and the optional *selectionCondition* (expressed on fields of the two objects) is true. The output object $o_{i,j}$ has three fields: one with the name of the left collection and contains object l_i, one with the name of the right collection and contains object r_j, a *geometry* field.

The **KEEP** clause permits to specify the content of field `geometry`. If **KEEP INTERSECTION** is specified, field `geometry` in $o_{i,j}$ represents the spatial intersection (for instance, the intersection of two crossing line is a point); if **KEEP LEFT** (resp., **KEEP RIGHT**) is specified, field `geometry` in $o_{i,j}$ represents the full geometry of the left object l_i (resp., of the right object r_j); if **KEEP ALL** is specified, field `geometry` in $o_{i,j}$ represents the union of geometries in both the left object l_i and the right object r_j.

The alternative options **LEFT**, **RIGHT** and **FULL** at the beginning of the operator slightly change the behaviour of the operator when specified. If the **LEFT** (resp., **RIGHT**) option is specified, all non-matching objects $\overline{l}_k$ in the left collection (resp., all non-matching objects $\overline{r}_k$ in the right collection) have a corresponding object $\overline{o}_k$ in the output collection, such that the field in $\overline{o}_k$ corresponding to the right (resp., left) object is not present, and its `geometry` field coincides with the geometry of $\overline{l}_k$ (resp., $\overline{r}_k$). The **FULL** option specifies that all non-matching objects $\overline{l}_k$ and $\overline{r}_k$) must have a corresponding object $\overline{o}_k$ in the output collection.

Example 2. *Consider collection* `Buildings` *shown in Listing 3.6 and collection* WaterLine *shown in Listings 3.7, stored in database* `ToyuDB`*. Suppose we are a company committed to perform maintenance of water lines. Thus, we want to know which water lines passes below which buildings in city* `"City A"`*. The query is the following:*

```
OVERLAY COLLECTIONS ToyDB.Buildings, ToyDB.WaterLines
ON Buildings.City = "City A"
KEEP INTERSECTION;
```

The operator performs a geospatial join between each object b_i *in collection* `Buildings` *and each object* w_j *in collection* WaterLines*, in such a way the*

geospatial representations of b_i and w_j intersect. In practice, we are interested in discovering which water lines passes below which building. The output collections :

```
[
 {
  Buildings:{
  "name":"buildingA",
  "city":"city A",
  "address":"address A",
  "geometry":{"type":"GeometryCollection",
              "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates": [
                      [[100.0, 0.0],
                       [101.0, 0.0],
                       [101.0, 1.0],
                       [100.0, 1.0],
                       [100.0, 0.0] ]]
                  }

            ]

       }

 },
 Waterlines:{
   "name":"WaterLineA",
   "city":"city A",
   "geometry":{"type":"GeometryCollection",
              "geometries":[
                 {
                  "type": "LineString",
                  "coordinates":
                     [[90.0, 0.0],
                      [103.0, 1.0],
                      [104.0, 0.0],
                      [105.0, 1.0]]
                 }
            ]
       }
  },
  "geometry":{
              "type": "LineString",
              "coordinates":
                 [[100.0,0.7692307692307692],
                  [101.0,0.8461538461538461]]
             }
    }
]
```

Notice that field Buildings *contains the matching object b_i coming from the*

left collection, while field `WaterLines` *contains the matching object* w_J *coming from the right collection; field* geometry *is the GeoJSON representation of the geospatial intersection between geometries of objects* b_i *and* w_j*, that in our cases is a line. Notice that we obtain only one pair, related to city* `"City A"`*, because it is the only one that actually overlays and the city of the building is the required one.*

JOIN COLLECTIONS

The **JOIN COLLECTIONS** operators makes the no-geospatial join between two collections. W.r.t. operator **OVERLAY COLLECTIONS**, the **JOIN COLLECTIONS** operator works on non-geospatial fields. The syntax of operator is hereafter.

```
[LEFT|RIGHT|FULL] JOIN COLLECTIONS [dbName.]collectionName1,
                                   [dbName.]collectionName2
ON joinCondition;
```

The output of the **JOIN COLLECTIONS** operator is a JSON collection obtained by pairing objects in both collections. For each object l_i in *dbName.collectionName1* (left collection) and for each object r_j in *dbName.collectionName2* (right collection), an object $o_{i,j}$ appears in the output collection if the *joinCondition* is true for the pair l_i, r_j. The output object $o_{i,j}$ contains two fields: the first one has the name of the left collection and contains object l_i, while the second one has the name of the right collection and contains object r_j.

Similarly to the **OVERLAY COLLECTIONS** operator, the alternative options **LEFT**, **RIGHT** and **FULL** at the beginning of the operator slightly change the behaviour of the operator, when specified. If the **LEFT** (resp., **RIGHT**) option is specified, all non-matching objects $\bar{l}_k$ in the left collection (resp., all non-matching objects $\bar{r}_k$ in the right collection) have a corresponding object $\bar{o}_k$ in the output collection, such that the field in $\bar{o}_k$ corresponding to the right (resp., left) object is not present. The **FULL** option specifies that all non-matching objects $\bar{l}_k$ and $\bar{r}_k$) must have a corresponding object $\bar{o}_k$ in the output collection.

Example 3. *Suppose we are the same company of Example 2. We need to associate restaurants to buildings, based on their address. We can write the following query.*

```
JOIN COLLECTIONS ToyDB.Buildings, ToyDB.Restaurants
ON Buildings.City=Restaurants.City AND
   Buildings.Address=Restaurants.Address;
```

The join condition specifies that objects must be pairs if their cities and their address coincides. The output collection is reported in the listing below. Notice that no attribute `geometry` *is present at the top level.*

```
[{
  {"name":"buildingA",
   "city":"city A",
   "address":"address A",
   "geometry":{"type":"GeometryCollection",
               "geometries":[
                  {
                   "type": "Polygon",
                   "coordinates": [
                      [[100.0, 0.0],
                       [101.0, 0.0],
                       [101.0, 1.0],
                       [100.0, 1.0],
                       [100.0, 0.0]]]
                  }
              ]
         }
  },

  {
   "name":"RestaurantA",
   "city":"city A",
   "address":"address A"
  }
}]
```

The join in example can be useful in order to get restaurant position information from its address.

MERGE COLLECTIONS

The **MERGE COLLECTIONS** operator merges the content of two or more collections into the temporary collection. Here is its syntax.

```
[ALL] MERGE COLLECTIONS [dbName.]collectionName
                        (,[dbName.]collectionName)+;
```

When the option **ALL** is not specified, the operator removes duplicate objects, possibly coming from different collections. When the **ALL** option is specified, duplicate objects are not removed. The operator exploits the heterogeneous nature of JSON collections: objects with different structure can be stored together without any limitation.

INTERSECT COLLECTIONS

The **INTERSECT COLLECTIONS** operator makes a set-intersection between collections and puts the resulting collection into the temporary collection.

```
INTERSECT COLLECTIONS [dbName.]collectionName1,
                      [dbName.]collectionName2;
```

The **INTERSECT COLLECTIONS** performs a deep equality matching: only identical objects present in both the input collections matches and only one single occurrence of them is put into the output collection.

SUBTRACT COLLECTIONS

The **SUBTRACT COLLECTIONS** operator makes a setoriented -subtraction between collections and puts the resulting object into the temporary collection.

```
SUBTRACT COLLECTIONS [dbName.]collectionName1,
                     [dbName.]collectionName2;
```

The **SUBTRACT COLLECTIONS** returns a JSON collections containing all objects in *dbName.collectionName1* without an identical object (based on deep equality matching) in *dbName.collectionName2*.

3.5.2 Carry on Operators

This group encompasses operators whose input collection is the temporary collection and possibly generate a new version of it. They are suitable to perform continued transformations on one single collection, thus avoiding the need to continuously refer to collections.

FILTER

The **FILTER** operator permits to filter objects in the temporary collection, according to some selection conditions, and possibly change the structure of selected objects. The operator is designed to deal with the heterogeneous nature of JSON collections; for this reason, the syntax is more articulated than other operators.

```
FILTER
 (CASE:
  (fieldName = value (,fieldName = value)* |
   WITH fieldName (,fieldName)* |
   WITHOUT fieldName (,fieldName)*)+
    [WHERE selectionCondition]
    [PROJECT fieldName (,fieldName)*])+
(KEEP OTHERS|DROP OTHERS);
```

The **FILTER** operator contains one or more **CASE** branches. Each **CASE** branch specifies a subset of objects to select by means a list of selectors. Three types of selectors are provided: equal conditions on fields; **WITH** selectors, that asks for objects with the specified field name; **WITHOUT** selectors, that ask for objects without the specified field name. For any objects in the input temporary collection that matches with the specified selectors, the **WHERE** clause, if specified, is evaluated and if it is false, the object is discarded, otherwise the object is kept. Finally, if the **PROJECT** clause is specified, the object is projected on

the specified list of fields, in order reduce the number of fields or extract (by means of a dot notation) fields nested in object fields.

If more than one **CASE** branch is specified, they are evaluate in the order: an object is handled by the first branch that matches with it.

The alternative clauses **KEEP OTHERS** and **DROP OTHERS** specifies what to do with objects that do not match any **CASE** branch. If **KEEP OTHERS** is specified, these objects are put into the output collection; if **DROP OTHERS** is specified, these objects are not put into the output collection.

Example 4. *Consider* Restaurants *shown in Listing 3.8 which represents some restaurant. We are only interested in the names of restaurants in city "city C" having the geometry. The query is hereafter.*

```
GET COLLECTION ToyDB.Restaurants;
FILTER
 CASE: city="city C" WITH geometry
 PROJECT name
DROP OTHERS;
```

The **GET COLLECTION** *operator retrieves the initial* Restaurants *collection from the persistent database; this collection becomes the new temporary collection. The* **FILTER** *operator carries on the process.*

Only one **CASE** *branch is sufficient for our purpose, with two selectors: the first one is an equal condition on field* City*; the second one is a* **WITH** *selector on field* geometry*. As a result, only one object in our sample* Restaurants *collection will be selected, and then projected on the field* name*. Other possibly not matching objects are dropped.*

The new temporary collection produced by the operator is hereafter.

```
[{
  "name":"RestaurantC"
}]
```

Notice the very simple structure of the resulting object.

GROUP BY

The **GROUP BY** operator groups objects in the input temporary collection based on a list of grouping fields. Here is its syntax.

```
GROUP BY fieldName (,fieldName)*
INTO fieldName
[SORTED BY fieldName (,fieldName)*];
```

The **GROUP BY** operator groups the objects in such a way a group contains all the objects $o_1, \ldots, o_n$ having the same values for field names specified after the **GROUP BY** jeywords.

For each group, an object g_k appears in the output temporary collection, such that it has all the grouping fields and a field vector containing objects $o_1, \ldots, o_n$; the name of this field is specified in the clause **INTO**.

Finally, the optional clause **SORTED BY** specifies whether to sort objects into the vector fields. If so, the following field names are the sort keys.

Notice that the output temporary collection contains as many objects as the number of groups.

Example 5. *Consider collection* WaterLines *shown in Listing 3.7. We might be interested in grouping water lines by their city. The query is hereafter.*

```
GET COLLECTION ToyDB.WaterLines;
GROUP BY city
INTO waterLineCity;
```

The **GET COLLECTION** *operator retrieves the* `WaterLines` *collection from the persistent database and makes it the new temporary collection, on which the* **GROUP BY** *operator works. Water lines are grouped by attribute* `City`*; the name given to the vector field containing grouped objects is* `waterLineCity`*. Here is the output temporary collection.*


```
[{"city":"city A",
  "waterLineCity":[
   {"name":"WaterLineA",
    "city":"city A",
    "geometry":{"type":"GeometryCollection",
                "geometries":[
                   {"type": "LineString",
                    "coordinates":
                       [[90.0, 0.0],
                        [103.0, 1.0],
                        [104.0, 0.0],
                        [105.0, 1.0]]
                   }
               ]
        }
  },

  {"name":"WaterLineB",
   "city":"city A",
   "geometry":{"type":"GeometryCollection",
               "geometries":[
                  {"type": "LineString",
                   "coordinates":
                      [[102.0, 10.0],
                       [103.0, 2.0],
                       [104.0, 1.0],
                       [102.0, -1.0]]
                  }
              ]
         }
```

```
  }]
  },
  {"city":"city C",
   "waterLineCity":[
      {"name":"WaterLineC",
       "city":"city C",
       "geometry":{"type":"GeometryCollection",
                   "geometries":[
                      {
                       "type": "LineString",
                       "coordinates":
                          [[104.0, 10.0],
                           [105.0, 2.0],
                           [109.0, 1.0],
                           [110.0, -1.0]]
                      }
                ]
         }
  }]
}]
```

The output contains an object which represent the group for city A *(first object) and an other object which represents the grouping for* city C *(second object). The first object contains a field* name *(grouping field) with value* city A *and a field* waterLineCity *which contains all objects of the Listings 3.7 with the field* city *equals to* "city A" *(grouped objects). The second object has the same structure but the grouping field is* City C.

SET INTERMEDIATE AS

The **`SET INTERMEDIATE AS`** operator stores the input temporary collection into the intermediate results database IR. The syntax of operator is:

```
SET INTERMEDIATE AS collectionName;
```

Note that *collectionName* is the name given to the new temporary collection into the intermediate results database.

SAVE AS

The **`SAVE AS`** operator saves the input temporary collection into a persistent database.

```
SAVE AS dbName.collectionName;
```

The name of the new collection stored into the persistent database *dbName* is *collectionName*.

DERIVE GEOMETRY

The **DERIVE GEOMETRY** operator is used to manage geospatial fields, when geospatial fields of JSON input are not conform to the J-CO data model. The input collection is the temporary collection created by previous operator. The syntax of operator is:

```
DERIVE GEOMETRY (POINT(latFieldName,lonFieldName)| fieldName);
```

The **DERIVE GEOMETRY** operator adds the `geometry` field to each object o_i in the input temporary collection. If the **POINT** pair is specified, the values of the two specified fields plays the role, respectively, of latitude and longitude. If a simple *fieldName* is specified, this is intepreted as a GeoJSON specification whose name is not `geometry`, or, in case of dot notation expression, it can be in some nested objects.

3.6 Complete Example

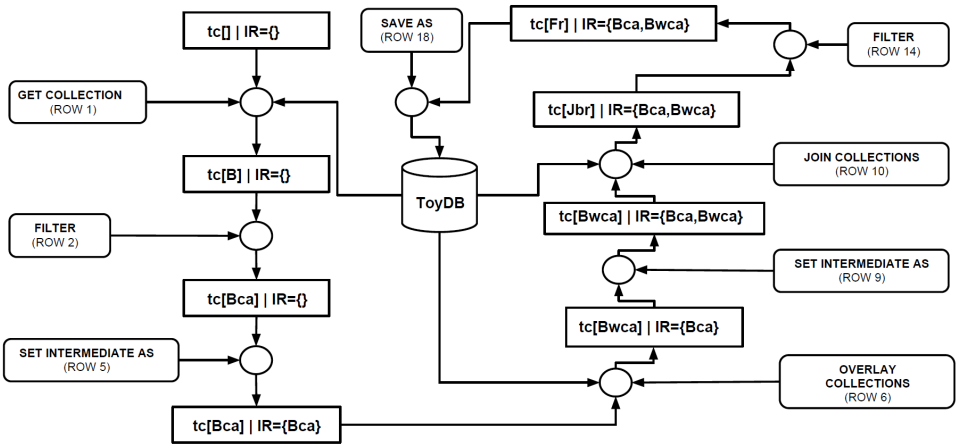


Figure 3.1: *Execution Model example.*

In order to show the effectiveness of J-CO for complex tasks, we wrote the query in Listings 3.9, based on the collections stored in our sample DB named `ToyDB`. The goal is to extract the names of restaurants located in buildings in *city A* under which some water lines pass. This query is useful for the water lines maintenance in the *city A*.

Listing 3.9: *Complete Example*

```
1 GET COLLECTION ToyDB.Buildings;
2 FILTER
3  CASE: city="city A" WITH geometry
4  DROP OTHERS;
5 SET INTERMEDIATE AS BuildingsCityA;
```

```
6  OVERLAY COLLECTIONS BuildingsCityA, ToyDB.WaterLines
7    KEEP INTERSECTION;
8  SET INTERMEDIATE AS BWCityA;
9  JOIN COLLECTIONS BWCityA, ToyDB.Restaurants
10   ON BWCityA.BuildingsCityA.address = Restaurants.address;
11 FILTER
12  CASE: WITH name
13   PROJECT Restaurant.name
14  DROP OTHERS;
15 SAVE AS ToyDB.RestaurantsWL;
```

Figure 3.1 shows the execution model of the example. It shows how the temporary collection and intermediate database IR change during the process. In Figure 3.1, for lack of space, we used abbreviated names with the following meaning:

- `B` means `ToyDB.Buildings`,
- `Bca` means `BuildingsCityA`,
- `Bwca` means `BWCityA`,
- `Jbr` means *Output of the join operator*
- `Fr` means *Output of second filter*

Moreover, with term *tc*, we indicate the temporary collection and with term *IR*, we denote the intermediate database. Rectangles represent the system status and show how the temporary collection and intermediate database IR change during the query process.

The query process works as follows:

3.6.1 *New Task*

The query starts a new task using the **`GET COLLECTIONS`** operator (which is a *Start operator*). The **`GET COLLECTION`** (see row 1) outputs the `ToyDB.Buildings` (shown in Listings 3.6), into the temporary collection. The Figure 3.1 shows that the element `B` (`ToyDB.Buildings`) has become the new *tc*.

The temporary collection is now the input collection for the **`FILTER`** operator. It keeps any buildings located in *"city A"* which have the field *geometry* (see its **`CASE`** branch). The result becomes the new temporary collection. Figure 3.1 shows that the element `Bca` (`BuildingsCityA`) has become the new *tc*. All others buildings are dropped (see the **`DROP OTHERS`** option).

The temporary collection produced by **`FILTER`** is then saved into the IR by the **`SET INTERMEDIATE AS`** operator. This operator saves *tc* into IR, naming the collection as `BuildingsCityA` (see row 5). Figure 3.1 shows that the element `Bca` (`BuildingsCityA`) has been added to IR.

3.6.2 *Subtask 1*

In row 6 the **OVERLAY COLLECTIONS** operator starts a new subtask. This operator makes a geospatial join between collection `BuildingsCityA` (previously stored in the intermediate results database IR) and collection `ToyDB.WaterLines` (retrieved from the persistent database `ToyDB`). It outputs the geospatial intersection (see **KEEP INTERSECTION** clause). The output will be saved as the new temporary collection *tc*. Figure 3.1 shows that the element `Bwca` (`BWCityA`) has become the new *tc*.

The following **SET INTERMEDIATE AS** operator in row 9 saves the temporary collection generated by the **OVERLAY COLLECTIONS** operator into IR with the name `BWCityA`. Figure 3.1 shows that the element `Bwca` (`BWCityA`) has been added to IR.

3.6.3 *Subtask 2*

In row 10 the **JOIN COLLECTIONS** operator starts the final subtask. This operator makes a join between the intermediate collection `BWCityA` (retrieved from IR) and the collection `ToyDB.Restaurants` (retrieved from the persistent database `ToyDB`). It outputs the join between buildings and restaurants which have the same address (see the **ON** clause). The output becomes the new temporary collection. Figure 3.1 shows that the element `Jbr` (output of the join) has been the new *tc*.

The temporary collection generated by **JOIN COLLECTIONS**, is then filtered by the **FILTER** operator. The operator (see row 14) keeps any restaurants which have field *name* (see, the **CASE** branch) and projects them on the name of this restaurants (see the **PROJECT** clause), generating a new temporary collection. Figure 3.1 shows that the element `Fr` (output of the **FILTER** operator) has become the new *tc*. All others objects are dropped (see the **DROP OTHERS** clause).

Finally, the current temporary collection *tc*, which indeed contains the desired restaurant names, is saved into the persistent database named `ToyDB` by the **SAVE AS** operator, with name `RestaurantsWL`. Figure 3.1 shows that the element `Fr` (output of the filter) is saved into persistence database *ToyDB*.

The reader can observe that the query is easy to read and rather intuitive as far as its execution is concerned.

3.7 J-CO Engine

In order to prove the feasibility of our approach, we developed a prototype version of the *J-CO Engine*. In Subsection 3.7.1 we present the architecture; in Subsection 3.7.2 we discuss a performance evaluation that encourages us to carry on the research.

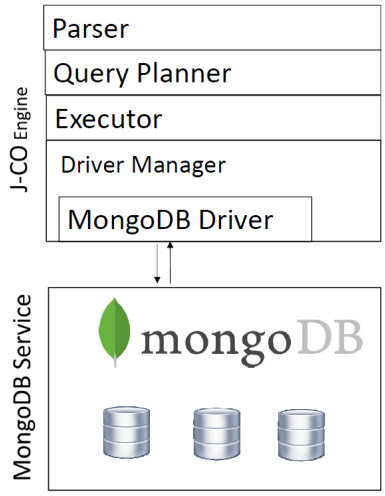


Figure 3.2: *Architecture of J-CO Engine.*

3.7.1 Architecture

J-CO Engine is written in Java; it is designed to be an external tool w.r.t. MongoDB: this choice makes the tool substantially independent of the DBMS technology, and opens the way to make J-CO engine able to operate on several others DBMSs. The architecture of J-CO Engine is reported in Figure 3.2.

The stack of components includes the *Parser*, that transform the query text into an internal representation received by the *Planner*. This component generates the execution plan by relying on an internal object-oriented representation. The *Executor* controls the execution of each single instructions in the query plan, manage main memory usage and temporary collections and the intermediate results database. When necessary, it interacts with the *Driver Manager*, that at the moment includes only the *MongoDB Driver*, but in the future will includes drivers for other DBMSs and data sources. The Driver Manager interacts with MongoDB to retrieve collections, store result collections and, when necessary (in case of low levels of available memory) transfers intermediate collections to MongoDB.

The query plan is represented as a vector of objects defined on the abstract class *JCO_Executable*, from which specific and non-abstract subclasses are derived, one for each J-CO operator (see Figure 3.3). By exploiting polymorphism, the *Executor* calls the `execute_operator` method of each object, providing references to its internal component to get input collections, store the temporary collection, the intermediate collections and save collections to the persistent database. The `execute_operator` can call these components when necessary.

The implementations of the operators make use of some libraries, neces-

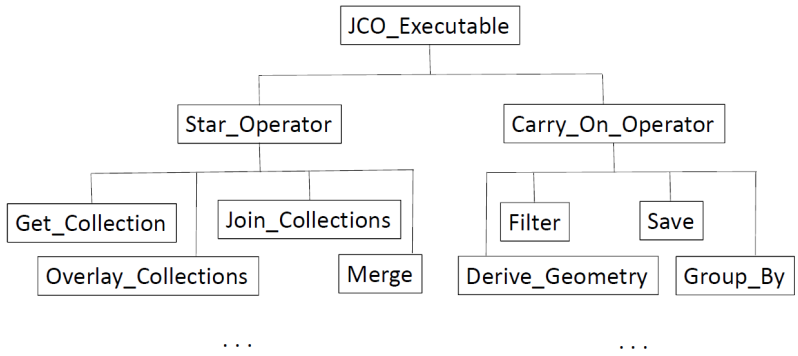


Figure 3.3: *Class hierarchy for JCO_Executable objects used to build the query plan.*

sary to an efficient implementation. In particular, *LocationTech Spatial4j*[1] and *Tsusiat Software Java Topology Suite* [2] are used to deal with spatial representations, while *David Moten's R-tree/R*-tree in memory indexing*[3] is used to build main-memory spatial indexes necessary to implement the **`OVERLAY COLLECTIONS`** operator.

Main-memory indexes are important to accelerate the execution of complex operators, such as **`GROUP BY`** and **`JOIN COLLECTIONS`**. In both case, we create main-memory indexes on the fly, so that we can perform equi-join operations (currently, we support only the equality predicate in join conditions) or grouping with linear complexity, after the indexes are built.

A similar approach is used for the **`OVERLAY COLLECTIONS`** operator. A spatial index on-the-fly over the geometry field of the left collection using the *R*-Tree* library. For each element of the right collection, a search on the index is done and all the documents of the left collection that intersect with the geometry of the element are retrieved. For each pair, a new JSON object is generated and inserted into the new temporary collection.

3.7.2 Performance Evaluation

Operation	Size of C1	Size of C2	Size of Output	Indexing time	computation time	Total time
JOIN COLLECTIONS	25360	25360	446768	194.95	62,6	257.55
OVERLAY COLLECTIONS	176	25360	21336	245.1	3455.3	3700.4
OVERLAY COLLECTIONS	176	111993	111993	870.4	14808.9	15679.3
GROUP BY 1 field	25360		6	201	212.1	413.1
GROUP BY 2 fields	25360		3058	192.9	229	421.9
GROUP BY 3 fields	25360		11854	193.6	248.1	441.7
DERIVE GEOMETRY	25360		25360		220.5	220.5

Table 3.3: *Execution times (in msec) observed during the experiments.*

[1] https://github.com/locationtech/spatial4j
[2] JTS - http://tsusiatsoftware.net/jts/main.html
[3] https://github.com/davidmoten/rtree

In order to have a first validation of our approach, we ran some experiments with the prototype of the J-CO Engine. We made the experiments on a Mac-Book Pro Retina, equipped with an Intel i7 Quad Core processor with 2,5 GHz clock-rate, 16 GB RAM e a SS hard disk.

We used two data sets available on the internet. The first one is named *restaurants.json* and contains about 26500 objects with point geometry. The second one is named *countries.geo.json* and describes the polygon geometry of 180 countries.

In Table 3.3, we report the execution times (in msec) we measured during tests; each experiment was repeated 10 times and we report the average execution times.

We tested the most critical operators, i.e., the **`JOIN COLLECTIONS`** family, the **`OVERLAY COLLECTIONS`** and the **`GROUP BY`** operators. Furthermore, we also tested the **`DERIVE GEOMETRY`** with the **`POINT`** option.

In the table, columns *Size of C1* and *Size of C2* report the number of objects in the input collections; column *size of output* reports the number of objects in the output collection; columns *Indexing time*, *Computation time* and *Total time* reports, respectively, the execution time neded for the indexing phase (when done), the execution time for the computation phase (generation of the output objects) and the total execution time of the operator.

As far as the **`JOIN COLLECTIONS`** operator is concerned, we tested it by joining the collection *restaurants.json* with itself. Notice that, even though the number of potential pairs was 25360×25360, the total execution time is only a quarter of second.

As far as the **`OVERLAY COLLECTIONS`** operator is concerned, we overlaid collection *countries.geo.json* (countries) with collection *restaurants.json* (restaurants). In order to stress the implementation, in the second experiment with the **`OVERLAY COLLECTIONS`** operator, we duplicated the *restaurants.json* collection to get to 111993 objects. The execution times show that this is the slowest operator, but 3 secs and 15 secs are acceptable times for users (notice that the time needed to build the R*-tree index is negligible).

The **`GROUP BY`** operator was tested on collection *restaurants.json* (restaurants) with three different settings: 1, 2 and 3 grouping fields. Observe that the execution times are substantially stable and less than half second is needed.

Finally, the **`DERIVE GEOMETRY`** with the **`POINT`** option has been tested on collection *restaurants.json* (restaurants). It is able to process about 150000 objects in a quarter of sec.

The results show that the approach is correct. Even though some operators could appear too complex to process, the J-CO engine is able to execute them producing results in seconds or tenth of seconds, depending on the operators. Consequently, we can expect that complex queries can be processed in minutes. Thus, the J-CO query language is effective as far as execution time is

considered.

3.8 Related Work

J-CO moves from our previous work on the problem of querying heterogeneous collections of complex spatial data [21, 59]. In those work, we proposed a database model able to deal with heterogeneous collections of possibly nested spatial objects, based on the composition of more primitive spatial objects; at the same time, an algebra to query complex spatial data is provided, inspired by classical relational algebra. W.r.t. those works, J-CO rely on the JSON standard, thus we do not define an ad-hoc data model; furthermore, J-CO abandon the typical relational algebra syntax, because it relies on a more flexible and intuitive execution model.

The adoption of NoSQL databases is motivated by the need of flexibility, as far as data structures are concerned. In interesting survey about NoSQL databases in [38], where several systems are catalogued and classified. In particular, a DBMS like MOngoDB falls into the category of *document databases*, because collections of JSON objects are generically considered as *documents*. Consequently, the query language provided by such systems does not allow complex and multi-collection transformations like those provided by J-CO (see the web sites reported in the footnote[4] for details). Readres interested in NoSQL DBMSs evaluation can refer to [63] and to [24].

As far as query language for JSON objects are concerned, the closest proposal is *Jaql* [52]. It was designed to help Hadoop [71] programmer writing complex transformations, avoiding low-level programming, to perform in a cloud and parallel environment. Flexibility and physical independence are the main goals of Jaql: in particular, its execution model is similar to our execution model, since it explicitly relies on the concept of pipe; in fact, the pipe operator is explicitly used in Jaql queries. However, it is still oriented to programmers; its constructs are difficult to understand for non programmer users, while J-CO constructs are at a higher level and truly declarative.

On the same track of query languages for improving MapReduce/Hadoop programming, we cite *ChuQL* [43], which deals with XML documents (not JSON objects, even though an XML is logically related to JSON). Compared to Jaql, ChuQL is even worst, in the sense that its constructs are still too programmatic; thus, it is not suitable for non programmers.

An interesting language is *Pig Latin* [54], a query language developed by Yahoo for writing complex analysis tasks on nested (1-NF, first normal form) data sets on top of Hadoop; thus, JSON collections are implicitly included. Pig Latin's constructs have names similar to J-CO constructs, however, it strongly

[4] MongoDB: https://www.mongodb.com/
CouchDB: http://docs.couchdb.org/en/2.0.0/

relies on the concept of variables: the result of each statement must be explicitly assigned to a variable, that can be later referred to by other statements; in contrast, J-CO provides the concepts of temporary collection and intermediate results database. The *DryadLINQ* language, presented in [72], follows a very similar approach to Pig Latin's approach.

J-CO is instead thought as a language oriented to non programmers. In our mind, we would like to replicate what happened with SQL, that enabled database querying to non technicians. Furthermore, J-CO provides operators for transforming geo-referenced objects, that is a common lack of other languages.

Since JSON and XML are both suitable for representing semi-structured documents, it is worth mentioning the mostly known languages for querying XML documents. The first to mention is *XPath* [19], that allows to write path expressions to retrieve elements in a single XML document. On the basis of XPath, a complex language designed to work on collections of XML docuemtns is *XQuery*. Among all features, it provides constructs to generate new documents, as well as the possibility to express complex queries. However, it is still oriented to programmes and not to non programmer users.

Finally, our proposal is somehow related to the world of *PolyStore* DBMS, i.e., database management systems that deal with several DBMS at the same time, each of them possibly providing a different logical model, such as relational, graph, JSON, pure-text, images, videos. An interesting work on this topic is *BigDAWG* [31, 36]. Among all features, the support to querying relies on the query languages provided by the integrated DBMSs, enriched with a couple of instructions that permit to transform data from one model to another (CAST) and to specify where to store a query results.

Even considering a more classical enterprise environment, where images and video are not considered, the concept of PolyStore DBMS is relevant. An example is the *QUEPA* (QUerying and Exploring a Polystore by Augmentation) system, that provides a solution to give a uniform view of relational DB, data warehouse, JSON data. In this perspective, J-CO could play a significant role in a PolyStore environment, since relational data, CSV files and spatial data managed by spatially augmented object-relational DB such as PostgreSQL/PostGIS can be viewed as non-nested JSON/GeoJSON objects.

3.9 Evaluation of J-CO query language

In this research, we proposed a query language, named J-CO, specifically devised to query heterogeneous collections of JSON objects stored in a NoSQL document DBMS such as MongoDB. The idea is to provide non-programmer users with a declarative query language able to handle JSON collections in such a way objects can be filtered, recombined and aggregated in a flexible

way. Geo-referenced objects can be queried by means of the `Overlay` operator, allowing the specification of complex spatial queries in a very simple way. The execution model on which the J-CO query language relies is, simple, intuitive, and suitable to express complex queries based on several subtasks. Furthermore, J-CO overcome typical limitations of query languages provided by document DBMSs, that are usually unable to combine collections in a flexible way.

The prototype we realized demonstrates the feasibility of the approach, and the performance we obtained, even though the implementation is not particularly optimized, are encouraging. We expect, at the end of development, that J-CO could become a powerful and effective tool for querying databases of JSON collections.

At this moment, we do not think J-CO is complete. We are planning to extend current operators with new features able to better deal with nesting, such as specific aggregation functions possibly based on geo-references. But above all, we are going to define new powerful operators for spatial analysis over big data, since territorial analysis (with the contribution of possibly geo-referenced open data published by institutions) are becoming more and more important in day-by-day life of institutions and citizens.

Conclusions

THE model-based techniques presented in this thesis have shown, also through some experimental results, the efficacy and the advantages given by their use in the design of modern systems.

SCA-PatternBox is a framework for modeling service-oriented applications with design patterns that also permits the validation and verification of their behavior. The framework allows the definition and the semi-automated application of design patterns into the design of a service-oriented software architecture. *SCA-PatternBox* facilitates the construction of a service-oriented application and helps engineers understand alternative means for achieving alternative architectures in the application's design and code. We evaluated the usability and usefulness of the framework on the Order system case study and on a quality-driven adaptation scenario of the Stock Trading System [60] where patterns and tactics required would have been difficult to apply and combine manually without the availability of a tool like SCA-PatternBox.

Security Enhanced Docker study improves the use of SELinux in *Docker*. Two techniques have been presented: *Docker Policy Module* (DPM) improves the use of the SELinux *Type Enforcement* (TE) model, in Docker, associating specific policy rules to the processes inside the containers using SELinux modules; *Category minimization problem* improves the use of the SELinux *Multi Category Security* (MCS) model, in Docker, associating specific categories with each element in the system. These improvements ensure greater system security giving to the containers only the rules that are needed in order to perform their functionality. The experiments showed that *Category minimization problem* permits to save many system resources (few categories are needed to satisfy a large policy) and the time needed to obtain a solution is not critical. Moreover, this technique also protects *Data Volume Containers*.

These techniques permit to ensure system security saving many resources.

J-CO query language was devised to query heterogeneous collections of JSON objects stored in a NoSQL document DBMS such as MongoDB. It provides non-programmer users with a declarative query language able to handle JSON collections in such a way that objects can be filtered, recombined and aggregated in a flexible way. *J-CO query language* permits to execute complex queries (also on Geo-references objects) in a simple way. The prototype demonstrates the feasibility of the approach, and the performance obtained is encouraging even though the implementation is not yet optimized.

Bibliography

[1] Apache tuscany. *http://tuscany.apache.org/*.

[2] Asmeta website. *asmeta.sourceforge.net/*.

[3] Oasis service component architecture (sca). *http://www.oasis-opencsa.org/sca/*.

[4] Patternbox website. *http://www.patternbox.com/*.

[5] Sca-patternbox repository. *https://github.com/stevencapelli/SCA-PatternBOX*.

[6] Topology and orchestration specification for cloud applications (tosca). *http://docs.oasis-open.org/tosca/tosca/v1.0/os/tosca-v1.0-os.html*.

[7] Apparmor. *http://wiki.apparmor.net/*, 2016.

[8] The geojson format specification. *http://geojson.org/geojson-spec.html*, 2016.

[9] Introducing json. *http://www.json.org/index.html*, 2016.

[10] Introduction to mongodb. *https://docs.mongodb.com/manual/introduction/*, 2016.

[11] What is docker? *https://www.docker.com/what-docker*, 2016.

[12] What is open data? *http://opendatahandbook.org/guide/en/what-is-open-data/*, 2016.

[13] Elvinia Riccobene Patrizia Scandurra Alessandro Carioni, Angelo Gargantini. A scenario-based validation language for asms. In *International Conference on Abstract State Machines, B and Z*, pages 71–84. Springer, 2008.

[14] Ian Stark Alex Blewitt, Alan Bundy. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 224–232. ACM, 2005.

[15] Patrizia Scandurra Angelo Gargantini, Elvinia Riccobene. A metamodel-based language and a simulation engine for abstract state machines. *J. UCS*, 14(12):1949–1983, 2008.

[16] Ralf Reussner Anne Koziolek, Heiko Koziolek. Peropteryx: automated application of tactics in multi-objective software architecture optimization. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 33–42. ACM, 2011.

[17] Paraboschi Stefano Bacis Enrico, Mutti Simone. Apppolicymodules: mandatory access control for third-party apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 309–320. ACM, 2015.

[18] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011.

[19] Anders Berglund, Scott Boag, Don Chamberlin, Mary F Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath). *World Wide Web Consortium (W3C)*, 2003.

[20] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[21] Gloria Bordogna, Marco Pagani, and Giuseppe Psaila. Database model and algebra for complex and heterogeneous spatial entities. In *Progress in Spatial Data Handling*, pages 79–97. Springer, 2006.

[22] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[23] H Butler, M Daly, A Doyle, S Gillies, S Hagen, and T Schaub. The geojson format. Technical report, 2016.

[24] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Record*, 39 (4):12–27, 2011.

[25] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: a survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[26] T Edwin Chow. Geography 2.0: A mashup perspective. *Advances in web-based GIS, mapping services and applications*, pages 15–36, 2011.

[27] Wuwei Shen Dae Kyoo Kim. Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 16(3):329–359, 2008.

[28] John Grundy David Mapelsden, John Hosking. Design pattern modelling and instantiation using dpml. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 3–11. Australian Computer Society, Inc., 2002.

[29] R. Stärk E. Börger. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[30] James H Cross Elliot J. Chikofsky. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[31] A Elmore, Jennie Duggan, Michael Stonebraker, Magdalena Balazinska, Ugur Cetintemel, Vijay Gadepally, J Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment*, 8(12):1908–1911, 2015.

[32] Fabio Albani Elvinia Riccobene, Patrizia Scandurra. A modeling and executable language for designing and prototyping service-oriented applications. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 4–11. IEEE, 2011.

[33] Patrizia Scandurra Elvinia Riccobene. A formal framework for service modeling and prototyping. *Formal Aspects of Computing*, 26(6):1077–1113, 2014.

[34] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2008.

[35] Somayaji Anil Forrest Stephanie, Hofmeyr Steven. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 418–430. IEEE, 2008.

[36] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. The bigdawg polystore system and architecture. *arXiv preprint arXiv:1609.07548*, 2016.

[37] Aldo Gangemi. Ontology design patterns for semantic web content. In *International semantic web conference*, pages 262–276. Springer, 2005.

[38] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[39] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. In *Linux Conference*, volume 3, 2004.

[40] David C Rine Jaeyong Park, Seok Won Lee. Uml design pattern metamodel-level constraints for the maintenance of software evolution. *Software: Practice and Experience*, 43(7):835–866, 2013.

[41] Gregor Kiczales Jan Hannemann. Design pattern implementation in java and aspectj. In *ACM Sigplan Notices*, volume 37, pages 161–173. ACM, 2002.

[42] Amnon H Eden Rick Kazman Jonathan Nicholson, Epameinondas Gasparis. Verification of design patterns with lepus3. 2009.

[43] Shahan Khatchadourian, Mariano P Consens, and Jérôme Siméon. Having a chuql at xml on the cloud. In *AMW*. Citeseer, 2011.

[44] D. Laney. 3-d data management: controlling data volume, velocity and variety. Technical Report META Group Research Note, META Group, February 2001.

[45] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. Json data management: supporting schema-less development in rdbms. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1247–1258. ACM, 2014.

[46] Henri Habrias Marc Frappier. *Software specification methods: an overview using a case study*. Springer Science & Business Media, 2012.

[47] Allouche Yair Corradi Antonio Dolev Shlomi Foschini Luca Mattetti Massimiliano, Shulman-Peleg Alexandra. Securing the infrastructure and the workloads of linux containers. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 559–567. IEEE, 2015.

[48] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt, 2013.

[49] Bill McCarty. *Selinux: Nsa's open source security enhanced linux*. O'Reilly Media, Inc., 2004.

[50] Tommi Mikkonen. Formalizing design patterns. In *Proceedings of the 20th international conference on Software engineering*, pages 115–124. IEEE Computer Society, 1998.

[51] Paraboschi Stefano Mutti Simone, Bacis Enrico. Policy specialization to support domain isolation. In *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, pages 33–38. ACM, 2015.

[52] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[53] University of Trier and Schloss Dagstuhl. Dblp. *http://dblp.uni-trier.de/*, 2016.

[54] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[55] Elvinia Riccobene Paolo Arcaini, Angelo Gargantini. Asmetasmv: a way to link high-level asm models to low-level nusmv specifications. In *International Conference on Abstract State Machines, Alloy, B and Z*, pages 61–74. Springer, 2010.

[56] Elvinia Riccobene Patrizia Scandurra Paolo Arcaini, Angelo Gargantini. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.

[57] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.

[58] Steven Capelli Patrizia Scandurra. A practical and automated approach for engineering service-oriented applications with design patterns. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, pages 684–689. IEEE, 2014.

[59] Giuseppe Psaila. A database model for heterogeneous spatial collections: Definition and algebra. In *Data and Knowledge Engineering (ICDKE), 2011 International Conference on*, pages 30–35. IEEE, 2011.

[60] Patrizia Scandurra Raffaela Mirandola, Pasqualina Potena. Adaptation space exploration for service-oriented applications. *Science of Computer Programming*, 80:356–384, 2014.

[61] Sivakumar Chinnasamy Rajeev R Raje. elelepus-a language for specification of software design patterns. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 600–604. ACM, 2001.

[62] Sudipto Ghosh Eunjee Song Robert B France, D-K Kim. A uml-based pattern specification technique. *IEEE transactions on Software Engineering*, 30(3):193–206, 2004.

[63] Hecht Robin and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In *CSC-2011 International Conference on Cloud and Service Computing, Hong Kong, China*, pages 336–341, December 2011.

[64] Motoshi Saeki. Behavioral specification of gof design patterns with lotos. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 408–415. IEEE, 2000.

[65] Mohammad Alshayeb Salman Khwaja. Towards design pattern definition language. *Software: Practice and Experience*, 43(7):747–757, 2013.

[66] Patrizia Scandurra Steven Capelli, Benedetta Nodari. Sca-patternbox: an eclipse-based design pattern editor for service component architectures. In *ECLIPSE-IT'2012-Italian Eclipse Workshop, September 20-21, 2012, Accademia Aeronautica Pozzuoli, Naples–Italy*, 2012.

[67] Lunjin Luand Sooyong Park Suntae Kim, Dae-Kyoo Kim. Quality-driven architecture development using architectural tactics. *Journal of Systems and Software*, 82(8):1211–1231, 2009.

[68] Sooyong Park Suntae Kim, Dae Kyoo Kim. Tool support for quality-driven development of software architectures. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 127–130. ACM, 2010.

[69] David Chek Ling Ngo Toufik Taibi. Formal specification of design pattern combination using bpsl. *Information and Software Technology*, 45(3):157–170, 2003.

[70] D. J. Walsh. Tuning docker with the newest security enhancements. *http://opensource.com/business/15/3/docker-security-tuning*, 2015.

[71] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[72] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.