Fondo Investimenti per la Ricerca di Base (FIRB).
Programma Strategico Tecnologie abilitanti per la Società della Conoscenza-ICT

ART DECO
adaptive infrastructures for decentralised organisations

Preliminary report on devised methodologies and tools for verification, testing and QoS evaluation

Antonia Bertolino, Guglielmo De Angelis, Felicita Digiandomenico, Eda Marchetti, Ioannis Parissis (ISTI-CNR)
Luciano Baresi, Carlo Ghezzi, Dino Mandrioli, Raffaela Mirandola, Matteo Rossi, Paola Spoletini (PoliMi)
Vittorio Cortellessa, Paola Inverardi, Herny Muccini, Patrizio Pelliccione (UnivAq)
Marcello Bruno, Massimiliano Di Penta, Valentina Mazza (UniSannio)

Abstract

Methodologies and tools for verification, testing and Quality Assurance assessment are nowadays important activities in the process development, which start as soon as a software product is decided to be built. The purpose of the Verification and Validation (V&V) techniques is to ensure that the final product satisfies the original requirements and design and that it fits the intended usage. Due to the peculiarity of the ART DECO infrastructure, in this deliverable we classify the V&V techniques into two different groups: Off-line and On-line Verification and Validation. For each selected methodology the required background knowledge and the technical details are provided as well as a brief description of the related work. Finally in specific sections we described the possible application of the V&V techniques inside the ART DECO project

Data	07/25/2007
Tipo di prodotto	Deliverable R.A.10.1
Stato	Final version
Unità responsabile	Scuola Superiore di Studi Universitari e Perfezionamento Sant'Anna Pisa
Unità coinvolte	ISTI (Scuola S. Anna), PoliMI, UnivAq (UniSannio), UniSannio
Autore da contattare	Eda Marchetti (eda.marchetti@isti.cnr.it)
Versione	1.0

Change history

Version	Date	State	Notes
0.5	01/31/2007	Draft	List of the possible V&V techniques
0.8	04/30/2007	Draft	Selection of possible V&V approaches
0.9	06/30/2007	Draft	Specific proposals for the ART DECO infrastructure
1.0	07/25/2007	Final	Final version

Contents

1 Introduction

Since the 1980, the widespread use of dynamic, adaptive and mobile component-based service oriented applications has led a large part of the software engineering to focus its attention on verification, testing and Quality of Services evaluation. Attributes as quality, usability, safety and other characteristic aspects of software applications have been largely investigated and methodologies for validating and verifying them are being developed.

Software Verification and Validation (V&V) is nowadays an important activity in the process development, which starts as soon as a software product is decided to be built. Software V&V is conducted throughout the planning, development and maintenance of software systems, including knowledge-based systems, and may assist in assuring appropriate reuse of software. The purpose of the V&V is to ensuring that the final product satisfies the original design and requirements (Verification) and that fits the intended usage (Validation).

Due to the peculiarity of the ART DECO infrastructure, in this deliverable we classify the V&V techniques and methodologies into two different groups:

- Off-line Verification and Validation;
- On-line Verification and Validation.

The first group uses a (formal) description or a model of the system, which abstracts from the behavior and the architecture of the system. The V&V activities may include: the formal verification of the system space, the structural analysis, the test cases generation, the performance verification. Referring to the classical process lifecycle all these activities, as well as the model definition, start from the analysis phase and proceed till the system integration. They do not involve the operative phases of the development system. Some of the purposes of the Off-line V&V methodologies include: provide a prediction of the system behavior, once operative; define the tools useful for guiding the project choices; establish the corrective actions and improvements that can decrease the overall time and cost request for delivering a system.

On the other side the On-line V&V methodologies analyze the system at run-time. They mainly check if the system is working as established in the specifications. On-line data are collected during the execution of the system by means of monitoring techniques. Comparing with the Off-line V&V techniques, the On-line methodologies do not provide an exhaustive verification of the system space even if it is thinkable to readapt some of the off-line techniques for the on-line V&V. It is important to notice that both modification of the system model and the data collection during the run-tim are expensive activities in terms of time. In the cases where the analyzed system requires strict and quick reactions it could not be possible to properly apply the on-line V&V techniques.

In the rest of this deliverable, the ART DECO proposals will be presented according to above structuring.

1.1 Reference Architecture

In this deliverable we refer to the ART DECO logical architecture schematized in Figure 1.1 (more details are in the Deliverable R.A.9.1). This Figure aims to describe the logical dependencies and interactions with peripheral devices and services provided by different participants. It foresees that each ART DECO participant has a similar and modular structure composed by three layers each one containing specific components. The identified layers are:

- **Device access layer** which is represented in Figure 1.1 at the lowest level. At this level the physical devices (such as rfids or sensor networks) interact by means of *Operation Manager* components. They represent mainly the interfaces provided by ART DECO for abstracting from the physical differences between different devices.
- **Logical layer** in which the main component is the *Logical Object*. This component provides the abstraction of a single device so that it can be considered as a functional element. Every Logical Object interacts with a single instance of Operation Manager. It forwards events concerning the specific physical device from the Operation manager to the Application layer.
- **Application layer**, which contains two main components: the *Application Object* and the *Workflow manager*. The Application Object receives data from Logical Object and defines functionalities and operations that can be externally invoked by means of service calls. The *Workflow manager* manages the behavioral aspects of the Application Layer.

At all the levels communication between components is performed through events raised towards the higher abstraction layer and through the invocation of commands on components of the lower abstraction layer.

Referring to the Figure 1.1, each participant provides externally a set of services which can be used, composed and coordinated in the ART DECO infrastructure for implementing different and more complex functionalities. In particular the specialized participant *Workflow Manager* (positioned at the highest level in Figure 1.1) coordinates and composes the services provided by the ART DECO participants.

Considering this dynamic and compositional view of the ART DECO infrastructure, which foresees a large number of evolving components and services interacting remotely in a self-governed mode, we further classify the Off-line and On-line V&V methodologies into three levels as highlighted in Figure 1.1:

- The V&V at Service Level which involves the services provided by each participant and the top most Workflow Manger (the green band in Figure 1.1),

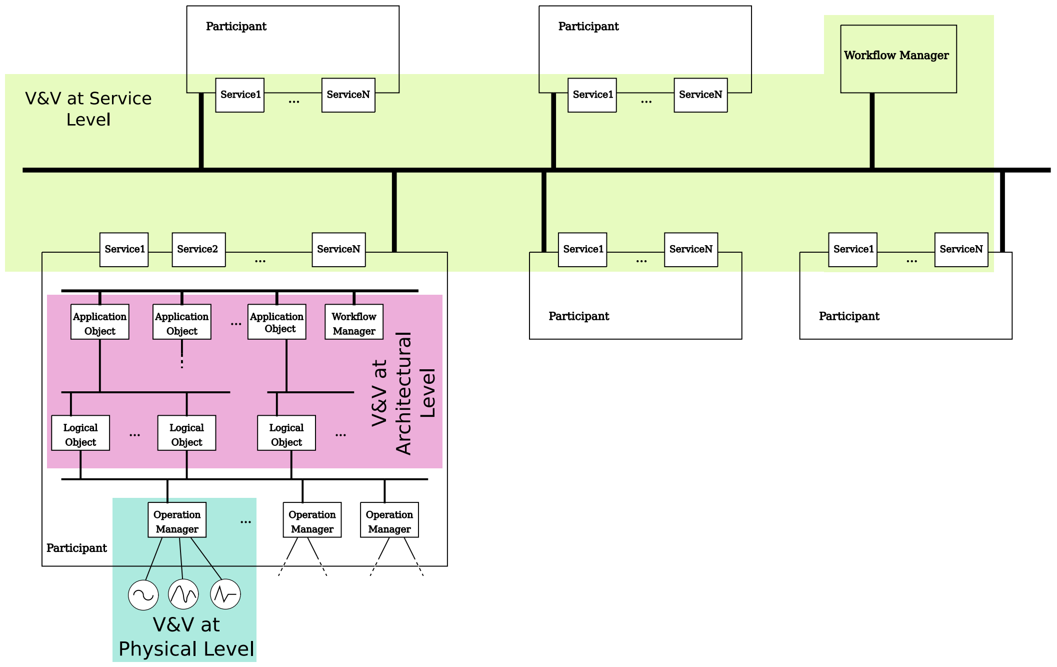


Figure 1.1: V&V level on the ART-DECO architecture structure.

- The V&V at Architectural Level which considers a single participant per time and involves the components of the Application and Logical layer (the pink band in Figure 1.1).
- The Physical Level which is focused on the devices access layer and involves the Operation Manager associated to physical devices (the light blue band in Figure 1.1).

1.2 Deliverable Outline

In this Deliverable we overview the devised methodologies and tools for verification, testing and Quality of Services assessment, classifying them into off-line and on-line V&V. In particular we have divided the Deliverable into three self-contained parts, each related to a different V&V topic, excluding the first which is an introductory section and the last which is the conclusion. Each part is then divided into two sections:

- The overall background knowledge section(s), in which details of the specific V&V techniques are provided with a brief description of the related works;
- A more specific section in which we described the possible application of the V&V techniques inside the ART DECO framework. In this deliverable these specific sections are always called **Within ART DECO**

Specifically:

Chapter 2: Abstract Modeling In this chapter we present a modeling formalism for describing a system representation for V&V purposes. Within ART DECO we adopt a solution that can be suitable for different purposes: describe the various aspects of the ART DECO architecture (from physical sensors to very high-level services) and automate and make more efficient analysis techniques for the off-line and on-line V&V. For facing these exigences, the ART DECO formalism is a set of three complementary ones that are used in conjunction to cover the various facets of the framework. Specifically we adopted:

- **DUALL**Y, an architecture description language;
- ArchiTRIO, a UML-compatible formal language based on the TRIO temporal logic;
- KLAPER, a modeling language that captures performance information.

In Section 2.4 the intregration of the above mentioned formalisms within ART DECO framework is provided.

Chapter 3 Off-line Verification and Validation In this chapter a selection of possible techniques applicable for the off-line V&V is provided. In particular we divided them into three groups:

- *Model checking:* It is one of the most promising techniques to facilitate early defect detection in requirement specifications. This technique is based on building a, typically finite, model of a system and checking if the model possesses the desired/required properties. In ART DECO we use model checking techniques at different abstraction levels, with the purpose of supporting from services verification to RFID devices verification.
- *Quality of Service assessment*: QoS is a set of qualitative and quantitative characteristics of a system, which encompasses classical dependability attributes (such as reliability, availability, safety), as well as performance and security aspects. The ART DECO project envisions a fully distributed information system, based on a middleware that will support peer-to-peer and GRID-based architectures, which offers services with desired or accepted levels of QoS. It is therefore utmost to apply, in the ART DECO context, methods and techniques which are able to provide quantitative assessments of relevant QoS measures, mainly dependability and performance. related ones.
- *Testing*: Testing is an important and critical part of software development, consuming even more than half of the effort required for producing deliverable software. Beside exhaustive verification techniques, such as model checking, many times a testing phase is required for facing the complexity of the applications to be verified or evaluating specific qualities and properties of the software. In this deliverable we focus on test cases generation, execution and

test result analysis. In particular considering the test cases generation we propose testing techniques focused on the verification of functional and non functional properties.

In each of the sections called "Within ART DECO" a preliminary plan for integrating and exploiting the above mentioned formalisms within the ART DECO framework is provided.

Chapter 4 On-line Verification and Validation The dynamic composition of the modern system architectures requires that the design-time V&V is supported also by a complementary mechanism to allow for the analysis of the evolving system at run-time. In this chapter an analysis of two different run-time approaches to be possibly employed in ART DECO is provided:

- *Software Systems Run-time Monitoring* In general terms monitoring is the activity of observing and checking that a system is running according to some specified functional and non-functional requirements. It is a quite delicate and expensive activity that asks for careful planning. It is necessary to develop strategies that in a particular context maximize chances of discovering faults still not excessively burdening software performance. In this deliverable two different monitoring approaches, namely Mosaico and WSCoL (and its monitoring engine) are presented.
- *Search-based Testing of Service Level Agreements* Roughly the Service Level Agreements (SLAs) is a form of contract between service providers and consumers, and its violation would cause lack of satisfaction for the consumer and lost of money for the provider. For this reason, before offering a SLA, a service provider would limit the possibility that it can be violated during service usage. In this deliverable we explore the use of Genetic Algorithms (GAs) to generate test data causing SLA violations.

In section 4.3 we show how monitoring and SLA online testing can be used for on-line verification of the ART DECO architecture and application.

2 Abstract Modeling

In order to address the verification issue in a complex system, it is crucial to first identify a modeling formalism rich enough to capture the important features of the system under description, one that can guide the verification phase for example by helping designers determine *what* precisely should be verified.

Formalisms and analysis techniques are tightly intertwined. On the one hand, every formalism carries its set of verification techniques and tools. Hence, choosing a formalism limits the range of the analysis that can be performed on the models created with it. Conversely, fixing a verification technique restricts the set of formalisms that can be used to model the system to be analyzed. In addition, it is well known that there is usually a trade-off between the expressiveness of a formalism and the level of automation of the analysis techniques that can be performed with it.

As a consequence, formalisms (very roughly) range between those that are more "model-oriented" (more expressive, but for which it is harder to build automated analysis techniques and tools) and those that are more "verification-oriented" (less expressive, associated with highly automated analysis techniques).

In the ART DECO project both dimensions are needed: on the one hand, very expressive formalisms that can describe the various aspects of the ART DECO architecture (from physical sensors to very high-level services obtained from the composition of lower-level ones; that is, all the elements depicted in Figure 1.1; on the other hand, highly automated and efficient analysis techniques for the off-line and on-line verification of the aforementioned aspects, as shown in Figure 1.1. Hence, there cannot be only *one* formalism employed in the ART DECO framework, but, rather, a set of complementary ones that are used in conjunction to cover the various facets of the framework. In addition, methods (preferably formal in nature) to "switch" from a formalism to a different one should be included in the ART DECO framework to fully exploit the advantages of each formalism while mitigating its shortcomings.

In the rest of this chapter we introduce three formalisms that address different modeling problems and different verification techniques:

- **D**U**AL**Ly, an architecture description language;
- ArchiTRIO, a UML-compatible formal language based on the TRIO temporal logic;
- KLAPER, a modeling language that captures performance information.

The chapter is structured as follows: Section 2.1 presents the **D**U**AL**Ly framework; Section 2.2 briefly introduces the ArchiTRIO formal language through an example;

Section 2.3 describes the main features of the KLAPER language; finally, Section 2.4 discusses the advantages of including all three languages in the ART DECO framework and proposes an approach to integrate them within ART DECO.

2.1 DUALLy

The main limitation of existing architecture description languages ADLs is that each of them provides specific notations and languages casted to a particular analysis technique, leaving other techniques unexplored. Supposing an industry is interested in model-checking and fault tolerance analysis, a complete result is obtained only using two different ADLs. The process is complicated and made worse by the fact that each ADL uses a different notation for SA specification, thus making difficult any integration.

Two problems hamper the success of strategies based on traditional ADLs: ($P1$) languages used by ADLs are generally formal and sophisticated, making difficult their integration in industrial processes, ($P2$) it is impossible to construct an ADL able to support every kind of analysis, since any analysis technique requires additional analysis-specific notations and models. To overcome these problems we propose the framework depicted in Fig. 2.1. In particular, in order to solve the problem $P1$ a UML profile able to model the core architectural elements obtained by merging UML 2.0 and ADLs through a UML-based notation for Software Architecture descriptions is given (see *Dually Core* in the figure). This can be used as a reference modeling notation to describe architectures. In a basic scenario, a software architect can document its architecture by drawing a diagram, can check the diagram conformance to the *Dually Core*, and can run analysis tools to validate the architectural model.

To cope with the problem $P2$ the framework allows for the extension of the core notation enabling the introduction of analysis techniques. In fact, what we expect in a typical scenario is that the software architect needs a more expressive architectural model, for documentation or analysis purposes. The proposed framework handles this scenario, by permitting the addition of new modeling elements, new diagrams, or integrating different analysis tools. For example, in Fig. 2.1 two extensions are defined to support model checking (MC) and fault-tolerant (FT) analysis. For this purpose, specific operators are used to extend the constructs of the *Dually Core* giving place to *Dually MC* and *Dually FT* modeling languages respectively. Once the extensions are defined, the SA models described by means of the *Dually Core* can be incremented with the information that then will be necessary for performing analysis tasks by means of proper tools. For instance, in Fig. 2.1 the models *Model MC* and *Model FT* are created enabling model checking and fault-tolerance analysis respectively.

The extensions expressed at language definition level permits to maintain semantic relations between the produced different models (see the dotted arrow between *Model MC* and *Model FT* in the figure). In this way, if manual modifications on one of the extended models occur (for instance to implement the feedback obtained from given analysis tool) there is the possibility to identify the parts of the other models that should be consequently modified.

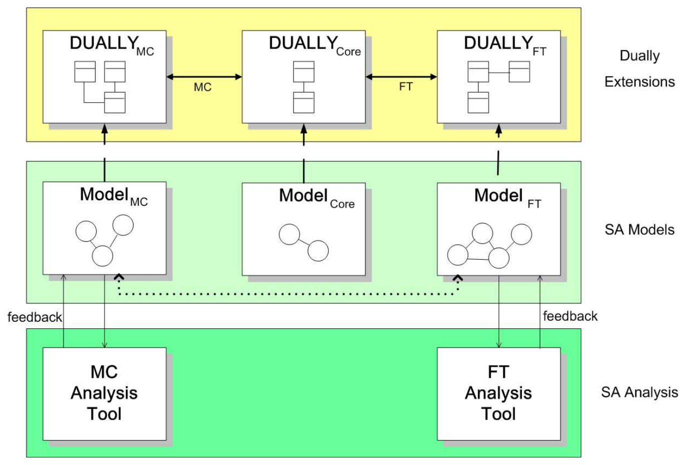


Figure 2.1: **DUALLy** Conceptual Model

In summary, in **DUALLy**:

i) the starting point consists in *identifying a core set of architectural elements always required*; then,

ii) *a UML profile is created to model the core architectural elements previously identified*

iii) *extensibility mechanisms (through model transformation techniques) are provided to add modeling concepts needed for specific analysis.* Finally,

iv) *semantic links mechanisms are kept between different notations.*

In the sequel of this section, we will briefly illustrate the **DUALLy** UML profile and its extensibility mechanisms.

2.1.1 The DUALLy UML profile

The **DUALLy** profile is depicted in Figure 2.2 and is defined in a ≪profile≫ stereotyped package modeling core architectural concepts. This profile is not meant to create a perfect matching between UML and architectural concepts. Instead, it proposes a practical mean to model their software architectures in UML, while minimizing effort and time and reusing UML tools.

2.1.2 Extending the DUALLy profile

The weaving operation [22], typically exploited for database metadata integration and evolution, can be used for setting fine-grained relationships between models or metamodels and executing operations on them based on semantic links. Furthermore, the integration of models or metamodels can be performed by establishing correspondences

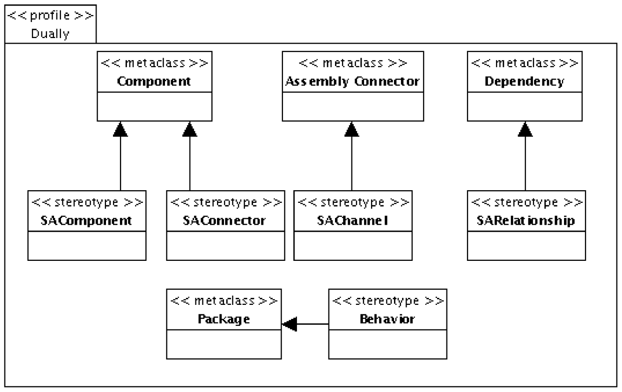

<< profile >> Dually
<< metaclass >> Component
<< metaclass >> Assembly Connector
<< metaclass >> Dependency
<< stereotype >> SAComponent
<< stereotype >> SAConnector
<< stereotype >> SAChannel
<< stereotype >> SARelationship
<< metaclass >> Package
<< stereotype >> Behavior

Figure 2.2: The **DUALLy** profile

among them by means of weaving associations specifically defined for the considered application domain. The description of such links consists of precise models conforming to appropriate weaving metamodels obtained by extending a generic one (inspired by [55]) with new constructs needed for the integration purposes.

In **DUALLy**, the Abstract State Machines (ASMs) [31] notation is used for the formal specification and execution of model transformations. ASMs bridge the gap between specification and computation by providing more versatile Turing-complete machines, and in the context of **DUALLy** is used as a formal and flexible platform on which to base a hybrid solution for model transformations: on one hand they combine declarative and procedural features to harness the intrinsic complexity of such task [48]; on the other hand, they are mathematically rigorous and represent a formal basis to analyze and verify that transformations are property preserving (as in [80]). ASM based model transformations start from an algebra encoding the source model and return an algebra encoding the target one.

For instance, in Figure 2.3 the algebraic encoding of the **DUALLy** profile, graphically depicted in Figure 2.2, is given. Such canonical encoding, with some minor considerations, enables the formal representation of any model (conforming to a specified metamodel) which can be automatically obtained. Moreover, the encoding contains all required information to translate the final ASM algebra into the corresponding model.

Further Readings

More detailed information on **DUALLy** can be found in Reference [74, 53, 52].

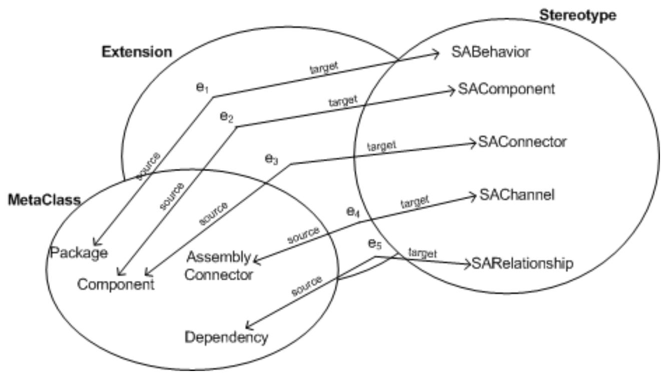


Figure 2.3: Algebraic encoding of **DUALLy** profile

2.2 ArchiTRIO

This section illustrates through a brief example some ideas about the ArchiTRIO approach to modeling systems. More precisely, it shows a possible ArchiTRIO formalization of a fragment of the conceptual model presented in [10], which is extended in [13], and which is the basis of the ART DECO architecture depicted in 1.1. This conceptual model defines the mechanisms through which data is collected from physical sensors and sent to higher-level, more sophisticated, autonomous objects for consumption and elaboration.

2.2.1 Modeling with ArchiTRIO

ArchiTRIO is a UML-compatible formal language [97], whose underlying philosophy is to approach the problem of modeling complex software-centric systems in a so-called "lightweight" manner [103]. More precisely, the idea behind the ArchiTRIO language is that users should be able to approach the modeling of complex systems with the familiar, widely-used, semi-formal UML notation (or subsets thereof), and introduce formal statements only when (if) and where needed.

While a presentation of the ArchiTRIO language is outside the scope of this document, (we refer the interested reader to [98, 97] for further details) let us illustrate therough a brief example how the language could be used for the high-level modeling of the ART DECO infrastructure.

The example is taken from [13, 10], and is not repeated here for the sake of brevity. Basically, the proposed architecture is a three-tier structure in which OperationManager objects are closest to the physical sensors collecting data from the field; LogicalObjects, instead, offer an intermediate logical abstraction of the sensors, which is a "bridge" between the low-level (i.e. closest to the field) OperationManagers and the high-level (i.e. closest to the "business logic") notion of ApplicationObject.

Figure 2.4 shows the UML (in fact, ArchiTRIO) classes representing the aforementioned elements and some of their features[1]. For example, Figure 2.4 shows that a LogicalObject has an attribute op_mans of type SetOfOperationManager which, as the name suggests, is simply a set of objects of type OperationManager. Hence, every LogicalObject is related to (i.e. it refers to) a set of OperationManagers. Attribute op_mans is marked with stereotype «state», since it represents a time-dependent feature that is piecewise constant (i.e. if at a certain instant the LogicalObject is associated with a certain set of OperationManagers, it remains so in a non-null interval of time). Also, the «state» stereotype indicates that the value of the attribute *can* change over time (on the other hand, if one wanted to model that the value of an attribute does not change over time, this would be marked as «TI», as is the case for example of attribute events of class FunctionalElement).

Having introduced the elements above, one could use ArchiTRIO formulas to state constraints on them. For example, a simple constraint might be that, at any instant, an OperationManager cannot be related to more than one LogicalObject. Then, one could introduce the following formula in class LogicalObject:

$$\begin{array}{l} \forall om : \mathsf{OperationManager}, lo_1, lo_2 : \mathsf{LogicalObject} \\ \quad \mathrm{Alw}(\quad (om \in lo_1.\mathrm{op\_mans} \wedge om \in lo_2.\mathrm{op\_mans}) \Longrightarrow lo_1 = lo_2 \quad) \end{array} \tag{2.1}$$

Without delving too much into the details underlying the logic part of the ArchiTRIO language (further details can be found in [97]), let us remark that, at its core, ArchiTRIO is a (higher-order) metric temporal logic with an implicit notion of time (similar to the TRIO logic from which it derives [41]). Hence, formula (2.1) states that in *any* instant (i.e. *always*) an OperationManager (i.e. an element of type OperationManager) can belong to the op_mans set of at most one LogicalObject[2].

Let us notice that the notion of Set is formally defined in ArchiTRIO using mechanisms that are similar to those classically used for abstract data types (see, for example, [67]).

Classes OperationManager, LogicalObject and ApplicationObject are all specialization of class FunctionalElement, from which they inherit operation subscribe. In fact, a FunctionalElement is an object that can "generate events" to be propagated to other FunctionalElements that "subscribe" to them. Notice that classes OperationManager and LogicalObject refine operation subscribe, and (implicitly) place a constraint on the type of the second argument of the operation (i.e. f_el); for example, the diagram shows that only LogicalObjects can subscribe to the events generated by an OperationManager (such a constraint could also be explicitly laid out through a logic formula).

As mentioned above, a FunctionalElement is statically associated with a set of events that it can notify to subscribers. This is represented through (time-invariant, i.e. «TI») attribute notifiable_events of class FunctionalElement. As shown in Figure 2.4, every

[1] The diagram was drawn using Rational® System Developer v.7 by IBM®, not an ArchiTRIO-specific editor. Hence, the graphical representation is the one adopted by the IBM tool.

[2] Notice that this would resolve any ambiguity arising from using a UML association to express this constraint. Using an association would leave the question open if the "link" is static or dynamic; formula (2.1) (and the semantics of the «state» stereotype) clarifies this: the link can be dynamic, but still at any instant an OperationManager cannot "belong" to two LogicalObjects.

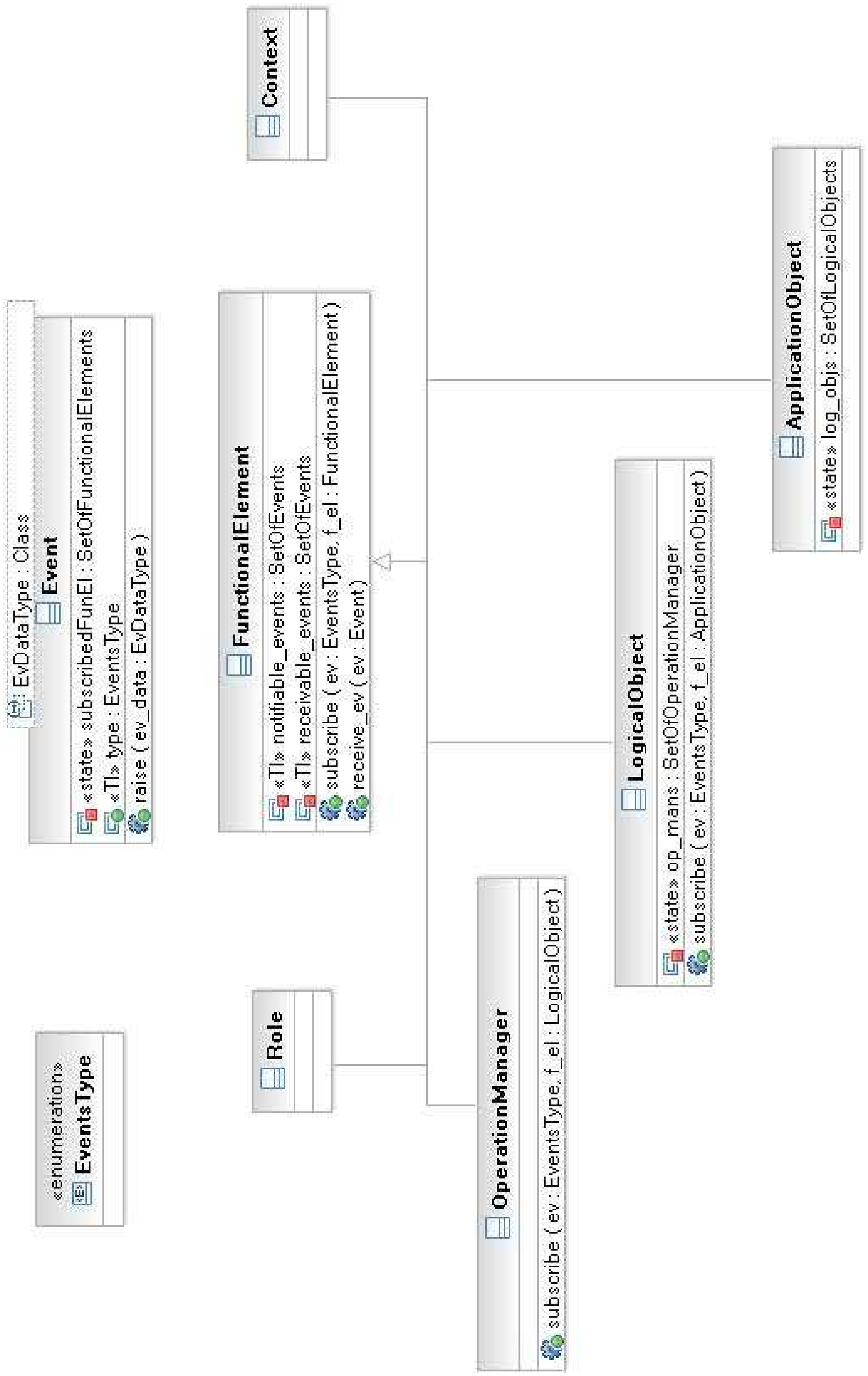

Context
EvDataType : Class
Event
«state» subscribedFunEl : SetOfFunctionalElements
«TI» type : EventsType
raise (ev_data : EvDataType)
FunctionalElement
«TI» notifiable_events : SetOfEvents
«TI» receivable_events : SetOfEvents
subscribe (ev : EventsType, f_el : FunctionalElement)
receive_ev (ev : Event)
ApplicationObject
«state» log_objs : SetOfLogicalObjects
LogicalObject
«state» op_mans : SetOfOperationManager
subscribe (ev : EventsType, f_el : ApplicationObject)
«enumeration»
EventsType
Role
OperationManager
subscribe (ev : EventsType, f_el : LogicalObject)

Figure 2.4: ArchiTRIO class diagram representing the key elements of the low-level ART DECO architecture.

an Event is parametric with respect to the kind information (i.e. the kind of data) it carries, which is represented by class Event having a parameter EvDataType which is a class (i.e. a type, in ArchiTRIO terms).[3] Every event has a (dynamic) set of subscribers, subscribedFunEl, which are of type FunctionalElement, and which are notified of the occurrence of the event (with data ev_data) when operation raise is invoked on the object of type Event. Hence, a FunctionalElement can also receive events (through operation receive_ev) from other FunctionalElements, and it is also (statically) associated with the set of events it can receive, as represented by «TI» attribute receivable_events.

One could state a "timeliness" property for the dispatching of an event to its subscribers after a raise operation is invoked through for example the following ArchiTRIO formula of class events:

$$
\begin{array}{l}
\forall r : \mathsf{raise}, f\_el : \mathsf{FunctionalElement}, evd : \mathsf{EvDataType} \\
\quad (\quad (r.\mathrm{start}(evd) \wedge f\_el \in \mathrm{subscribedFunEl}) \\
\qquad\quad \Longrightarrow \\
\qquad \mathrm{WithinF}(\quad \exists r\_ev : \mathsf{FunctionalElement.receive\_ev} \quad (f\_el.r\_ev.\mathrm{invoke}(evd)), \\
\qquad\qquad\qquad \mathtt{D\_DISPATCH} \;) \\
\quad)
\end{array}
\tag{2.2}
$$

Formula (2.2) states that any time a notification of the event (with data *evd*) starts (which is represented by the logic predicate $r.\mathrm{start}(evd)$), within D_DISPATCH time units (where
D_DISPATCH is a system-dependent constant[4]) all FunctionalElements f_el that subscribe to the event[5] must be sent the corresponding information (which is represented by an instance r_ev of operation FunctionalElement.receive_ev being invoked on f_el with parameter *evd*).

To conclude this section, let us remark that the fragment of formalization above is by no means intended to suggest how the development of the architecture of the ART DECO platform should be carried out. It is just meant to be a very small exercise in formalization with the ArchiTRIO language, using an ART DECO-related example; its intent is to highlight some of the features of the ArchiTRIO language that might be relevant and useful in the ART DECO project.

2.3 Performance Modeling with KLAPER

The main goal of the KLAPER (Kernel LAnguage for PErformance and Reliability Analysis) methodology is to support the model-based analysis of the effectiveness of

[3] Notice that this does not imply that an Event can carry only atomic information: if EvDataType is a "tuple" (i.e. the cartesian product of other types, which might also be represented as a "record"), then an instance of Event carries non-atomic data.

[4] A possible way to represent this would be to introduce D_DISPATCH as a parameter of class Event; however, filling out all the details of the model is beyond the scope of this document, and we will not delve any further in this issue.

[5] To be precise, all FunctionalElements that subscribe to the event *when the notification starts.*

adaptable component-based (C-B) applications, with a focus on the assessment of their performance and reliability attributes. To this end, it leverages MDD-based model transformation methodologies and tools to support the construction of a model for the performance/reliability analysis of adaptable C-B applications, starting from information extracted from the application design artifacts. With respect to analogous methodologies, the KLAPER-based methodology peculiarities can be summarized as follows:

- it explicitly addresses the modeling of adaptable applications;
- it is centered around the definition of an intermediate language (KLAPER) whose goal is to facilitate the translation from design-oriented to analysis-oriented models.

In particular, with regard to the second point, the goal of the intermediate language is to support the splitting of the complex task of deriving an analysis model (e.g. a queueing network) from a high level design model (expressed using UML or other component-oriented notations) into two separate and presumably simpler tasks:

1. extracting from the design model only the information that may be relevant for the analysis of some QoS attribute and expressing it in the notation provided by the intermediate language;
2. generating an analysis model based on the information expressed in the intermediate language.

Hence, KLAPER has been designed so as to capture in a lightweight and compact model only the relevant information for the performance and reliability analysis of static C-B systems, while abstracting away irrelevant details. In this way, it can be used as a bridge in a model transformation path from design-oriented to analysis-oriented models. We point out that KLAPER is neither a language to be used for the system design (notations like UML are better suited for this purpose) nor an analysis language (specific notations exist for this purpose, e.g. stochastic process algebras). KLAPER has been designed as an intermediate ï¿½distilledï¿½ language to help define transformations between design-oriented and analysis-oriented notations, filling the large semantic gap that usually divides them. In this perspective, it may also be seen as a conceptual model that can drive the construction of a performance/reliability analysis model of a C-B system.

To leverage existing MDD-based tools, KLAPER is defined as a MOF metamodel [65]. To support the distillation from the design models of a C-B system of the relevant information for performance/reliability analysis, KLAPER is built around an abstract representation of such a system, modeled (including the underlying platform) as an assembly of interacting Resources. Each Resource offers (and possibly requires) one or more Services. A KLAPER Resource is thus an abstract modeling concept that can be used to represent both software components and physical resources like processors and communication links. To bring performance/reliability related information within such an abstract model, each activity in the system is modeled as the execution of a Step that

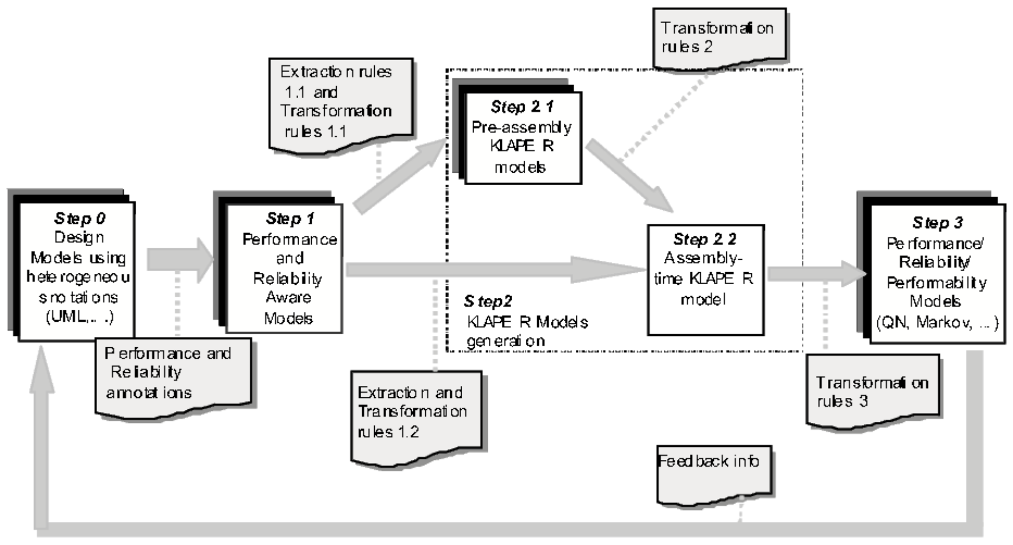

Step 0
Design Models using heterogeneous notations (UML,...)
Performance and Reliability annotations
Step 1
Performance and Reliability Aware Models
Extraction rules 1.1 and Transformation rules 1.1
Step 2.1
Pre-assembly KLAPER models
Transformation rules 2
Step 2.2
Assembly-time KLAPER model
Step2
KLAPER Models generation
Extraction and Transformation rules 1.2
Step 3
Performance/ Reliability/ Performability Models (QN, Markov, ...)
Transformation rules 3
Feedback info

Figure 2.5: KLAPER-based transformation framework

may take time to be completed, and/or may fail before its completion. The relationship between offered and required services is represented separately by means the Binding concept. This allows a clearer separation between the specification of the Resources and the description of how they are composed. Given the unifying concept of Resource adopted by KLAPER, a set of bindings can be used to model an assembly of application-level components or, similarly, the deployment of application components on the underlying platform. Moreover, the adaptation of a system obtained by some kind of reconfiguration can be simply modeled at the KLAPER level as a change in the set of bindings. Figure 2.5 illustrates the main steps of the general KLAPER-based transformation framework.

The input of our framework (Step 0) is represented by a set of component models and by a "glue-logic" model expressed by possibly heterogeneous notations. In general, design models may lack performance and/or reliability information that is necessary to derive meaningful analysis models. Therefore (Step 1), design models must be annotated with missing information about non-functional attributes. For example, if the design model is expressed in UML, annotations can be added following the OMG standard SPT or QoS profiles. In this way we obtain what we call Performance/Reliability-aware models. At Step 2 we generate KLAPER models from the design models with performance/reliability annotations. We distinguish a pre-assembly time case (Step 2.1) from an assembly time case (Step 2.2): in the former case information about how the selected resources and services are assembled is not yet available, while this information is available in the latter case. Hence at step 2.1 we can only map models of isolated elements of a component-based system onto corresponding KLAPER models, but we cannot specify bindings (at the KLAPER model level). To perform this step it is first necessary to extract from the design models information that is necessary for generating the target analysis model (extraction rules 1.1). To this end it could be useful that design model elements expose a proper analysis-oriented description, referred to as analytic interface in the literature; its goal is to enrich the original element with information that supports predictions about the performance and/or reliability of an architectural ensemble that element is part of. Then, we use suitable transformation rules to generate the corresponding KLAPER models (transformation rules 1.1). The main missing elements in the resulting models of Step 2.1 concern the Bindings between offered and required services that depend on how the single elements are assembled. Of course, without this information, we cannot use the KLAPER model to carry out any analysis of the overall assembly. When this information becomes available, we can specify all the needed Bindings, so getting a complete assembly model (transformation rules 2). On the other hand, if assembly information is already available when we start the generation of the KLAPER model, we may directly generate the overall KLAPER model (extraction and transformation rules 1.2). Finally, at Step 3, we can generate from this overall model a performance, reliability or performability model (transformation rules 3) expressed in some machine interpretable notation, and then we can solve it using suitable solution methodologies. As outlined in the introduction, we can use the MDA-MOF facilities for the definition of transformation rules to/from KLAPER models, provided that a MOF metamodel exists for the corresponding source/target model. Specifically, these transformations can be defined as a set of rules that map elements of the source metamodel onto elements of the target metamodel.

2.4 Within ART DECO

ArchiTRIO and KLAPER are formalisms that complement each other well, in that the former is well-suited to describe real-time properties of systems, while the latter is more focused on performance aspects. In addition, they are both capable of representing architectural features of systems, and their graphical representation is based on the UML. Hence, ArchiTRIO and KLAPER offer different, complementary views of a system architecture and are natural candidates to be integrated with each other.

DUALLy, instead, provides a means to allow different architectural modeling formalisms to "communicate" with each other. In the **DUALL**y framework, designers can describe different aspects of system architectures using formalisms tailored towards those aspects, and then "switch" from one view to the other using (formal) **DUALL**y transformations.

As a consequence, one could use the **DUALL**y framework as the centerpiece of the interaction between ArchiTRIO and KLAPER, as illustrated in Figure 2.6.

From a technical point of view, the integration suggested above is well-founded. In fact, **DUALL**y requires that a UML meta-model is available for the languages to be integrated in the framework, so that suitable extensions to the **DUALL**y core can be created, as outlined in Section 2.1. In this regard, KLAPER is already provided with such a meta-model [65], while ArchiTRIO (which, from a purely graphical point of view, adds very little to UML2, in the form of ArchiTRIO-sepcific stereotypes) can be easily given one.

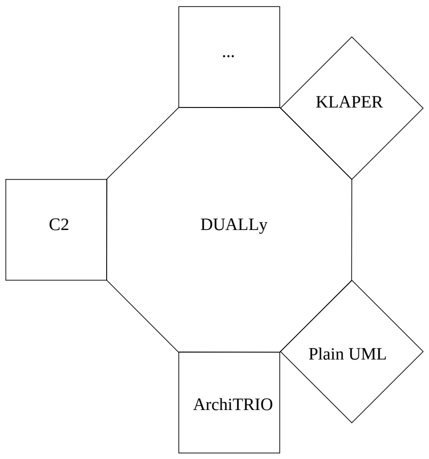


Figure 2.6: Integration of modeling techniques in ART DECO.

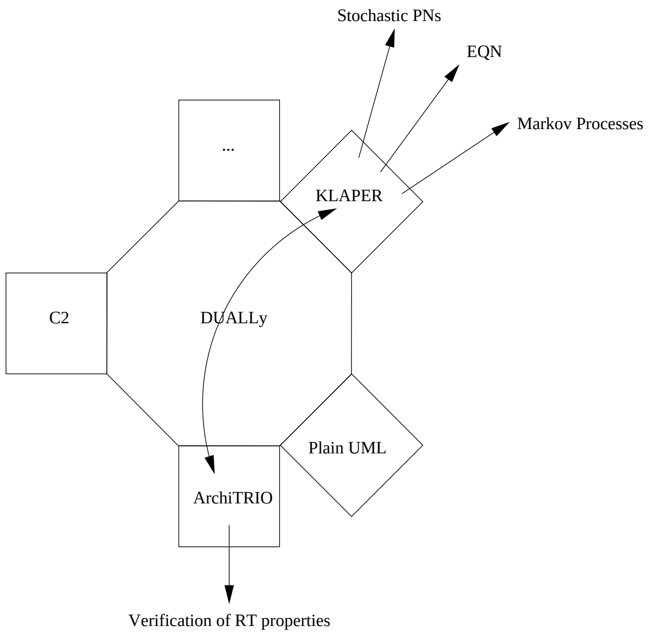


Figure 2.7: Interaction among modeling techniques and between modeling techniques and verification in ART DECO.

Diverse and powerful formalisms such as ArchiTRIO and KLAPER, integrated in the **DUALLy** framework, will permit to cover all modeling aspects of the ART DECO architecture depicted in Figure 1.1, be they “functional” or “non-functional”.

Integrating ArchiTRIO and KLAPER in the **DUALLy** framework has the added advantage that, as depicted in Figure 2.7, in this setting one could combine real-time modeling and verification provided through ArchiTRIO with the performance analysis provided by KLAPER through its target tools.

3 Off-line Verification and Validation

Computers are finding increasing applications in the field of the control of real time and safety-critical systems, such as avionic systems, medical systems and plant control systems. These kinds of applications require the development of techniques for the early detection of errors, that could otherwise be very costly or even catastrophic. Moreover these applications are often complex and need to be verified under different points of view and with different goals. In this section we overview different off-line V&V techniques that analyze different aspects of the system. In particular, we concentrate on model checking, Quality of Services assessment and and testing.

3.1 Model Checking

Model checking [45] is one of the most promising techniques to facilitate early defect detection in requirement specifications. This technique is based on building a, typically finite, model of a system and checking if the model possesses the desired/required properties. In literature, model checking techniques are applied at several levels of abstraction from the architectural level to the implementation. At each level of abstraction the properties that can be checked obviously varies. From the opposite point of view, at a more abstract level system details are masqueraded and both the verification time and the needed resources can be reduced. Based on these considerations in ART DECO we plan to use model checking techniques at different abstraction levels, with the purpose of supporting from services verification to RFID devices verification.

The techniques that we plan to use are the following three:

- BPLEL2BIR, a translator of BPEL, the well known standard for specifying web service workflows, into BIR, the input language of the model checker Bogor;
- Charmy, which allows the verification of architectural model described in UML-based notation, using Property Sequence Chart notation for expressing properties;
- TRIO2Promela, a translator of TRIO pure descriptive specifications into Promela, the input language of the model checker Spin.

In the following subsections we present the three approaches here introduced.

3.1.1 BPEL2BIR

We present an approach for the formal verification of workflow-based compositions of web services, described in BPEL4WS. Workflow processes can be verified in isolation, assuming that the external services invoked are known only through their interface. It is also possible to verify that the actual composition of two or more processes behaves correctly. We can verify deadlock freedom, properties expressed as data-bound assertions written in WS-CoL, a specification language for web services, and LTL temporal properties. Our approach is based on the software model checker Bogor [100], whose language supports the modeling of all BPEL4WS constructs.

More precisely, we look at web services in two different ways. First, since web services live in an open world and they normally belong to different administrative domains and may become available dynamically, the services retrieved from the open environment and composed to build a higher-level service must be treated as black boxes. Their internal behavior is not visible externally. The workflow process only knows such services through their interface specification, which describes their expected behavior. In such a case, we can only verify a workflow as a stand-alone process, i.e. we perform *intra-service* verification, where external services are abstracted by their specification. In other cases, however, we are allowed to open the black box. This may happen, for example, in the case of services developed within the same department or consortium, or in the case of open-source services. In such cases, the dynamically published services can be viewed as glass boxes, which expose their internals. As a consequence, verification can take advantage of the detail information that becomes available from the external service. It is thus possible to achieve *inter-services* verification, by formally analyzing properties of service compositions.

This dual-mode approach can be exploited for the verification of *local* and *global* properties. Indeed, in the first case one may be interested only in verifying data-bound or reachability properties within the workflow of a single process. In the latter, verification may focus on the behavior of the whole composition, for example by proving that a certain temporal property on the exchange of messages among business partners holds.

The main aspects that characterize our approach with respect to existing ones :

- it supports the analysis of both a stand-alone BPEL4WS process and of a composition of web-based processes;
- it supports the specification and verification of properties described in WS-CoL and in Linear Temporal Logic ;
- it covers all of BPEL4WS constructs (except those dealing with time);
- it uses a novel extensible model checker (Bogor);
- it offers significant efficiency gains, in term of the size of the model, over previous verification systems.

Further details on the presented approach and on the translation of BPEL in Bogor can be found in [23].

3.1.2 Charmy

Charmy allows the specification of a software architecture by means of both a topological (static) description and a behavioral (dynamic) one [61] (see Figure 3.1, A, and B). To increase the acceptability of our framework in industrial contexts we use a UML-based notation.

Charmy allows the specification of the *SA topology* in terms of components, connectors, and relationships among them, where components represent abstract computational subsystems and connectors formalize the interactions among components.

The internal behavior of each component is specified in terms of Charmy *state* and *sequence diagrams*. The Charmy notation for state machines (described later in Section 3.1.2) permits to specify the intra-component and inter-component behaviors of architectural components and connectors (i.e., the internal behavior of architectural elements and their integration, respectively). Charmy automatically checks the architectural conformance between the different models that compose the Charmy notation (e.g., two states with the same name cannot be introduced in the same state diagram; each state diagram can have one and only one initial state; for each send (receive) message in a component, there must exist at least a receive (send) message in another component). *Sequence diagrams* are used to specify how components communicate. The Charmy notation for sequence diagrams (described in Section 3.1.2) permits to graphically choose the desired communication between two different components (e.g. synchronous, asynchronous, deferred synchronous, multicast communication, etc...). Different arrows are used for representing different kinds of communication.

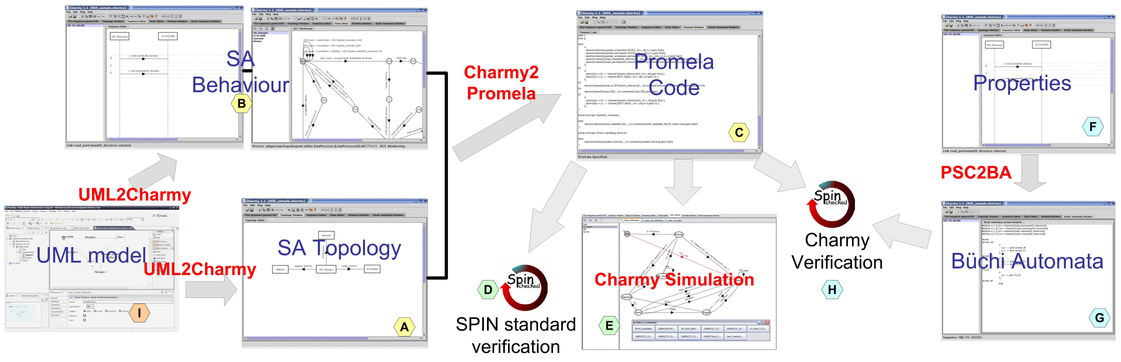


Figure 3.1: The Charmy Framework

Once the SA specification is available the *Charmy2Promela* translation feature is used to obtain from the model-based SA specification a formal executable prototype in Promela (as graphically depicted in Figure 3.1, C). Promela is the specification language of SPIN [72] and allows for the verification of concurrent systems communicating via either *messages* or *shared variables*. On the generated Promela code we can use the SPIN standard features to find, for instance, deadlocks or parts of states machines that are unreachable, or to simulate the architecture (Figure 3.1, D). However, by using the

SPIN standard simulation feature, simulation results are provided in terms of SPIN outputs that are difficult to be deciphered from a non-SPIN specialist. To make this analysis more appealing for industrial projects, we then provide the Charmy *Simulation* feature (Figure 3.1, E) that interprets SPIN results in terms of Charmy state machines.

In order to model-check the architectural model compliance to given properties, we introduce the Property Sequence Chart (PSC) notation for expressing properties [5, 4] and the Psc2Ba *algorithm* for translating PSC models into Büchi Automata (BA), an automata-based formal notation, which will be introduced later. PSC diagrams are an extended subset of UML 2.0 sequence diagrams. They are more complex of the Charmy sequence diagrams used for expressing the communication between components because they must be able to express sophisticated temporal properties (Figure 3.1, F). The Psc2Ba *algorithm* automatically translates PSC into Büchi automata then the SPIN model-checker is used to validate the temporal properties on the Promela code (Figure 3.1, G and H). Note that this translation process is fully automated.

In order to improve industrial usability, a recently introduced feature allows software designers to draw structural and behavioral diagrams with a standard tool notation (such as Omondo or IBM/Rational modeling tools) and then automatically generate Charmy-compliant diagrams (Figure 3.1, I).

The Charmy Notation

Components and connectors are represented by using boxes while lines represent communication channels between them. The behavior of each component is further specified by means of state and sequence diagrams.

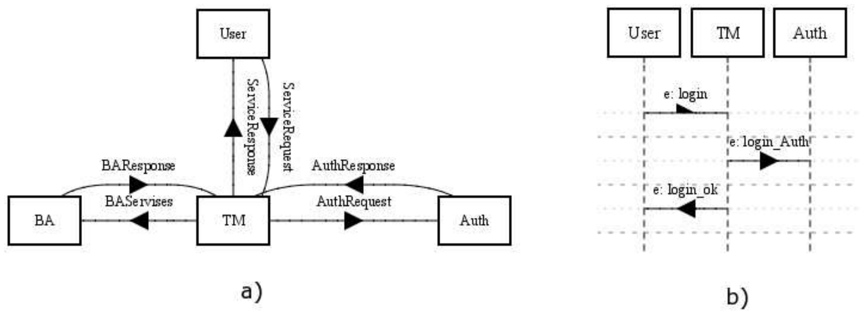


Figure 3.2: a) Topology Diagram formalism; b) Sequence diagrams formalism

Architectural Topology: to describe the architectural topology we use a subset of the UML component diagram. Figure 3.2.a illustrates the notation we use for the topology diagram. An architectural component is drawn with the familiar UML 2.0 notation for components. A connector can be seen as a complex coordination element or as a simple communication channel. Complex connectors are modeled using the UML notation for components, while an architectural channel is represented by an association line between architectural components. A coloring schema can be used to conceptually relate together sets of components, connectors, and channels.

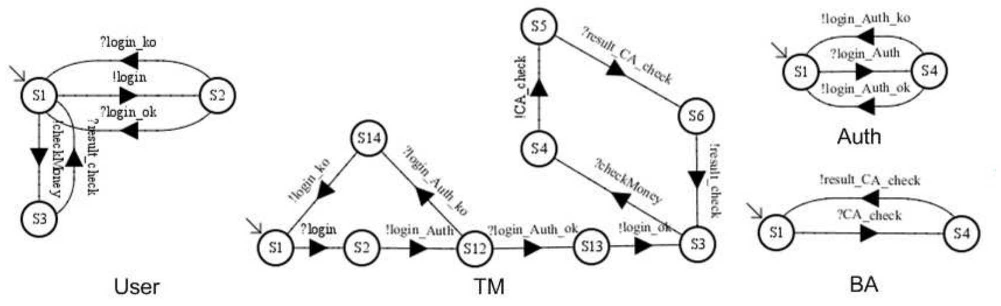


Figure 3.3: State diagrams formalism

State and Sequence diagrams: to describe state and sequence diagrams we use a notation general enough to encompass those used in current software development practice and rich enough to allow for analysis.

State diagrams are described using a State Transition Diagram notation close to the Promela syntax. The notation is shown in Figure 3.3 where labels on arcs uniquely identify the architectural communication channels and a channel allows the communication only between a pair of components. The labels are structured as follows:

$$`[`guard`]`event`(`parameter\_list`)``/`op_1`;`op_2`;`\cdots`;`op_n$$

where *guard* is a boolean condition that denotes the transition activation i.e. two state machines are synchronized only if the guard condition becomes true. The elements that can be written into the guard are variables local to the state machine or variables shared by all the state machines. An *event* can be a message sent or received (denoted by an exclamation mark “!" or a question mark “?", respectively), or an internal operation (τ) (i.e. an event that does not require synchronization between state machines). Both sent and received messages are performed over defined channels *ch*. An event can have several parameters as defined in the parameters list. $op_1, op_2, \cdots, op_n$ are the operations to be executed when the transition fires.

Components interact according to the semantics of their communication expressed in a sequence diagram. For example, if two components communicate over a synchronous channel, their composition is synchronous. Whenever two components are ready to send (receive) the same message, the components must synchronize.

The notation used for sequence diagrams is shown in Figure 3.2.b). Charmy sequences use a UML notation, stereotyped so that: (i) rectangular boxes represent instances of the architectural components defined in the topology (ii) arrows represent messages exchanged between two components under the constraint that what defined in the state machines must be respected. More precisely, let C_1 and C_2 be two components and $\{m_1, m_2, \cdots, m_n\}$ the set of messages sent by C_1 and received by C_2. It is not allowed the definition in the sequence diagram of a message $m_j \notin \{m_1, m_2, \cdots, m_n\}$ with C_1 sender and C_2 receiver.

The different kinds of communication are expressed by using different kinds of arrows.

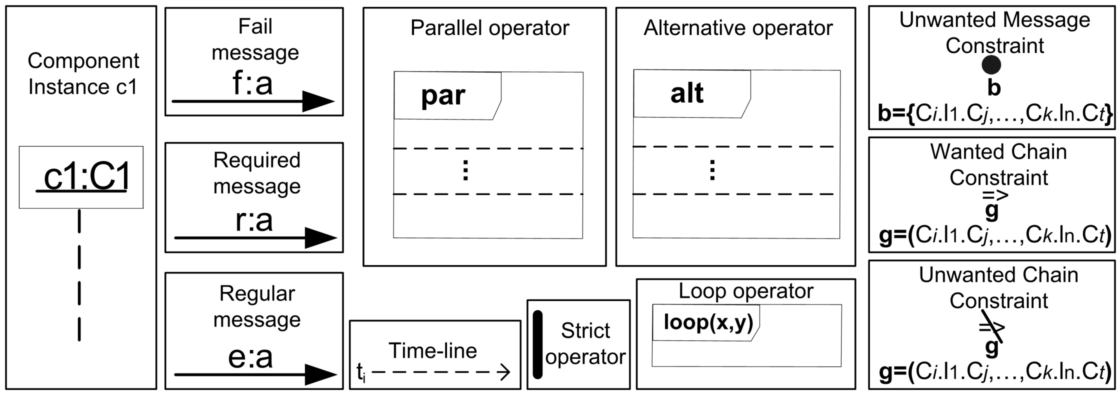


Figure 3.4: PSC notation

For instance in Figure 3.2.b are represented two different arrows: the filled arrow for synchronous communication and the half arrow for asynchronous communication. It is not required to specify for each message the kind of communication, since a synchronous communication is assumed by default.
Property Sequence charts: in our framework, properties are described by using the PSC notation (Figures 3.4). PSC is a scenario-based visual language that is an extended graphical notation of a subset of UML2.0 Sequence Diagrams. PSC can express a useful set of both *liveness* and *safety* properties in terms of messages exchanged among the components forming a system. Finally, an algorithm, called Psc2Ba, translates PSC into Büchi automata.

PSC uses a UML notation, stereotyped so that: (i) each rectangular box represents an architectural component, (ii) each arrow defines a communication line (a channel) between two components.

In order to clearly distinguish between mandatory, forbidden, and provisional messages, PSC provides three different types of messages to specify that a message can be mandatory, forbidden, or provisional:

- *Required messages:* are identified by "**r:**" prefixed to the labels. It is mandatory for the system to exchange this type of messages.
- *Fail messages:* the labels are prefixed by "**f:**". They identify messages that should never be exchanged. Fail messages are used to express undesired behaviors.
- *Regular messages:* the labels of such messages are prefixed by "**e:**". They denote messages that constitute the precondition for a desired (or an undesired) behavior. It is not mandatory for the system to exchange a Regular message, however, if it happens the precondition for the continuation has been verified.

Between a pair of messages we can specify whether other messages can occur (loose relation) or not (strict relation). Graphically, the strict relation is a thick line that links the messages pair (as in Figure 3.4).

Constraints are introduced to define a restriction in what can happen between the message containing the constraint and its predecessor (i.e. past constraint) and its successor (i.e. future constraint). Restrictions specify either a chain of messages (chain constraints) or a set of messages that the system must not to exchange (unwanted messages constraints). Informally, an unwanted messages constraint is satisfied iff all the set of messages specified as unwanted messages are not exchanged.

Unwanted messages constraints, are graphically represented as filled circles, see Figure 3.4. Wanted and unwanted chain constraints are graphically represented as arrows and crossed arrows, respectively (see Figure 3.4).

As showed in Figure 3.4 *parallel*, *loop*, and *alternative* operators are introduced with a UML 2.0 like graphical notation. Informally, the parallel operator allows a parallel merge between the behaviors of the two operands. The messages arguments of the operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved. The loop operator allows the operand to be repeated a number of times included between provided lower and upper bounds. The alternative operator has been introduced to have the possibility of specifying alternative sequences of messages.

More detailed information on Charmy can be found in Reference [93, 39].

3.1.3 TRIO2Promela

We have developed model checking algorithms for verifying strong temporal constraints for safety critical systems. The formal language TRIO [64], given its ability to easily state quantitative temporal properties, is a good candidate to express the system to be verified. Moreover, exploiting the modular characteristics of TRIO [86], the verification of strict time constraints can be more easily achieved for complex systems.

TRIO is a metric temporal first order logic with both past and future modalities and therefore it is well suited for describing such systems. Moreover, since TRIO can be extended with traditional object oriented constructs, it simplifies the description of large scale specifications. However, TRIO is very expressive and, in general, undecidable. Therefore, we define a decidable fragment of the logic, disallowing variables, considering the natural numbers as time domain and limiting all the other domains to finite domains. We chose to exploit the well-known model checker Spin [71], since its on-the-fly algorithm is very efficient. Hence we propose a practical translation from TRIO to Promela, the input language of Spin, giving the theoretical foundation of our approach. Briefly, the idea is to separate the past and the future components of the specification applying the Gabbay separation theorem [115], and to translate the past, that since the temporal domain is the set of natural numbers, works on finite prefix, with a deterministic Büchi automaton [96], and the future component with an alternating automaton [87]. In both cases, a finite set of timers is added to the automata to manage the metric. Then the two components are merged together through a composition operation [96]. Since the resulting automaton still allows alternation, a technique to directly simulate the automaton in Spin is proposed.
Notice that this technique is in general very useful when it is applied to models that allow parallelism, like statecharts [68]. Finally we propose a technique to deal with the

modular aspects of TRIO, taking advantages from the connections and the dependencies among modules, and proposing strategies to overcome the problems that arise from complex structures and circular dependencies. The translation of non modular TRIO is implemented in Trio2Promela [24].

3.1.4 Within ART DECO

In this section we have proposed three different approaches to perform model checking on different kinds of complex systems. Namely, the techniques we proposed are the following:

- BPLEL2BIR, a translator of BPEL into BIR; This translator allows the autho,atic verification of BPEL worflows.
- Charmy that verifies Property Sequence Chart on architectural model described in UML-based notation;
- TRIO2Promela, a translator of TRIO into Promela.

All these approaches are complementary and can be applied in ART DECO project at different levels or at the same level to check different aspects. In particular, BPLEL2BIR could be used at the application level for verifying ART DECO services expressed in BPEL. Charmy and TRIO2Promela could be both used at the abstract level. These two approaches checks differen aspects of the model and can be integrated:

1. Charmy focuses on interactions between different components and in general parts of the system ignoring internal behaviour of these parts;
2. TRIO2Promela focuses on internal behaviour of components and on how a single component reacts to stimulus received by its environment.

Moreover, we complement these approaches with a model checking technique at physical level, where we represent the sensor domain using finite state machines.

3.2 Quality of Service assessment

Quality of Service can be defined as a set of qualitative and quantitative characteristics of a system, which encompasses classical dependability attributes (such as reliability, availability, safety), as well as performance and security aspects. Given the increasing dependence of our society on computerized interconnected systems and services, it is more and more required to assess QoS properties of such systems to justifiably trust their operation. The ART DECO project envisions a fully distributed information system, based on a middleware that will support peer-to-peer and GRID-based architectures, which offers services with desired or accepted levels of QoS. It is therefore utmost to apply, in the ART DECO context, methods and techniques which are able to provide quantitative assessments of relevant QoS measures, mainly dependability and performance related ones.

In this section we (shortly) review the main characteristics of approaches for system dependability and performance validation at the early phase of software development.

In view of the intensive and complex nature of modern industrial systems, the design and operation of these systems require methodologies and techniques to select the optimal design alternative and operational policy. An obvious implication for the introduction of QoS analysis since the design phase of software systems is that it must be based on predictive analysis methodologies applied to some suitable system model. Modeling (both analytical and simulative) approaches for dependability and performance evaluation have been proven to be useful and versatile in all the phases of the system life cycle. A model is an abstraction of a system "that highlights the important features of the system organization and provides ways of quantifying its properties neglecting all those details that are relevant for the actual implementation, but that are marginal for the objective of the study". Several types of models are currently used in practice. The most appropriate type of model depends upon the complexity of the system, the specific aspects to be studied, the attributes to be evaluated, the accuracy required, and the resources available for the study. During the design phase, models allow to make early and objective design decisions by comparing different alternative architectural solutions and selecting the most suitable one (among those obeying other design constraints), and to highlight problems within the design. This early validation of the concepts and architectural choices avoids wasting time and resources before realizing whether the system fulfills its requirements or needs some re-design. Once design decisions are made, models allow predicting the overall behavior of the system (for instance as a basis for building a case for the acceptance of the system). For an already existing system, models allow an "a posteriori" dependability and performance analysis, to understand and learn about specific aspects, to detect possible design weak points or bottlenecks, to perform a late validation of the dependability and performance requirements (this can also be useful in certifying phase) and to suggest sound solutions for future releases or modifications of the systems. The modeling also represents an effective tool to foresee the effects of the system maintenance operations and of possible changes or upgrades of the system configuration. In view of these appealing features, ART DECO will greatly benefit from the application of model-based validation of QoS measures.

Of course, the above discussed positive aspects come with some negative counterparts. The most critical problem is complexity: it is not an easy task at all to reflect in a model all the relevant aspects of a complex system. Besides the ability to capture in the model such complex system behaviour, the problem is even exacerbated from the complexity of the model solution procedure. Moreover, models of complex systems usually require many parameters (the meaning thereof is not always intuitive for designers), and require determining the values to assign to them (usually by way of experimental tests), which may be very difficult. Obtaining values for all the parameters can be impossible during the preliminary design phases of the system. To cope with this problem, sensitivity analysis are performed, to identify those parameters to which the system is highly sensible. Indeed, sensitivity analysis allows evaluating a range of possible system scenarios by varying the values of model parameters, to determine the trends in the consequent variations of the analysed dependability figures. Another problem could be represented by the lack of sufficient expertise of the application designers in dependability analysis methodologies. To overcome this problem, supporting tools have been extensively proposed in the last years. Particular relevance is played by those tools, whose key idea is to define a model transformation that takes as input some "design-oriented" model of the software system (plus some additional information related to the QoS attributes of interest) and (almost) automatically generates an "analysis-oriented" model, that lends itself to the application of some analysis methodology.

The rest of this section is structured in three parts. First, modeling methodologies and tools that have been developed over the last decades to quantitatively assess dependability indicators (mainly reliability, safety, availability and performability) are reviewed. Then, a methodology for reliability prediction starting from a UML model of the software architecture is introduced. Finally, performance analysis based on the Software Performance Engineering framework is presented.

3.2.1 Dependability Modeling Methodologies and Tools

In this section, we shortly describe the main characteristics of the various classes of modelling methodologies that have been developed over the last decades to provide dependability engineers the support tools for defining and solving models. A distinction can be made between methodologies that employ combinatorial models (non-state space models) like fault-trees and reliability block diagrams, and those based on state space oriented representations (state space models), such as Markov chains and Petri net models, depending on the nature of their constitutive elements and solution techniques [81, 88, 9, 112].

The combinatorial approaches do not require the enumeration of system states and offer extremely simple and intuitive methods for the construction and solution of the models. However, they are inadequate to deal with systems that exhibit complex dependencies among components, and can not deal with repairable systems. Fault-Trees and Reliability Block Diagrams are among the two most popular combinatorial approaches. Fault-Trees are a deductive modeling and analysis technique based on the study of the events that may impair the dependability of a system [81, 54, 112]. Reliability Block Diagrams [76, 81, 107] are especially meant to evaluate the reliability related characteristics of systems composed of multiple components connected in series or in parallel. State space models can be either deterministic, if their behavior is exactly determined, or stochastic, if they have probabilistic nature. Because of the unpredictability characterizing the occurrence of the failure events, probabilistic state space models are more appropriate in dependability evaluation.

Stochastic models can be further classified in Markovian and non-Markovian according to the underlying stochastic process [43, 70, 112]. A wide range of dependability modeling problems fall in the domain of Markovian models, for example when only exponentially distributed times occur. Markov chains (DTMC and CTMC), Stochastic Petri nets

(SPN) and Generalized Stochastic Petri nets (GSPN) are among the major Markovian models.

A Markov chain is a stochastic process, having discrete or countable state space, which enjoys the *memoryless property*, also known as the *Markov property*: given the current state of the model, the future evolution of the model is described by the current state, and is independent of past states. Markov chains [73, 88, 70, 112] combine an extreme versatility with well developed and efficient solution algorithms, which have been implemented in many automated tools for the performance and dependability evaluation. A state change of a continuous-time Markov chain (CTMC) is called a state transition. The dynamic behavior of a CTMC is described by the transition rate matrix $Q = \|q_{i,j}\|$, where $q_{i,j}$, for each $i \neq j$, is defined as the rate with which the Markov chain moves from state i to state j, and $q_{i,i} = -q_i = -\sum_{i \neq j} q_{i,j}$. Once the state occupation probabilities of the CTMC have been obtained, the values of the most important dependability measures can be evaluated [70, 112]. To evaluate combined metrics, such as performability measures, a reward structure can also be defined, which assigns reward values to the states (or to the transitions) of the Markov chain model, obtaining the Markov reward models [88].

The Petri net modeling paradigm has been developed with the specific purpose of representing in a compact and clear way concurrence, synchronization and cooperation among processes [95]. Very soon, Petri nets have been widely accepted because of their ability to describe the qualitative and quantitative aspects of complex systems, and also because of their intuitive and appealing graphical representation. The class of (Markovian) Stochastic Petri Nets (SPN's) [85, 9] is a very popular timed extension of the place-transition Petri nets [94]. The graphical representation of a SPN model consists of places and timed exponential transitions connected by directed arcs. Places may contain tokens, entities represented as positive integers. The state of the SPN model is the marking of the net, a vector defined by the number of tokens in each place. Each transition has an associated random firing delay with negative exponential distribution. A transition is said to be enabled in a marking if each of its input places contains at least as many tokens as the multiplicity of the corresponding input arc. As soon as a transition gets enabled, a random firing time is sampled from the distribution associated to it, and a timer starts counting from that time downto zero. The transition fires if and only if it remains continuously enabled until the timer reaches zero. When the transition fires, from each input (output) place as many tokens as the multiplicity of the corresponding input (output) arc are removed (are added) in a single atomic and instantaneous operation (atomic firing rule). As the transitions fire, the marking of the net is changed and other transitions can get enabled or aborted. The dynamic evolution of the model, which evolves from an initial marking, can be described by a direct graph, called the reachability graph, whose nodes are the markings in the reachability set, and whose arcs are labeled with the transition causing the corresponding marking change. Thanks to the memoryless property of the exponential distribution, the remaining time to the firing of transitions which remain enabled after a change of marking is exponentially distributed, whether the memory of the transitions is kept or not. It is easy to realize that the evolution of a SPN model can be represented by a CTMC, whose state space elements are in a one-to-one correspondence with the elements of the reachability set, and whose transition rates among states are equal to the firing rates of the transitions that produce the corresponding marking change in the SPN. Thus, the associated Markov chain is indeed isomorphic to the reachability graph. An SPN model can be solved in terms of the marking occupation probabilities by performing the analysis of the associate Markov chain. A reward structure can be associated to the markings of a SPN model, to evaluate some metric of interest [42].

The class of Generalized Stochastic Petri Nets (GSPN's) [1] allows for exponential and instantaneous (which fire in zero time, once enabled) transitions. Conflicts among timed transitions are solved according to the same race model as in the case of SPN's, whereas conflicts among instantaneous transitions are solved by a priority assignment, and by associating weights to instantaneous transitions at the same priority level. The solution of a GSPN model resorts again to that of a Markov chain, associated to the reduced base model of the GSPN.

Stochastic activity networks (SAN) [104] are one of the most powerful (in term of modelling capabilities) stochastic extensions to Petri nets. They have a graphical representation consisting of places, activities, input and output gates and two operators Rep and Join to compose Sub-networks together in a bigger SAN called "Composed Model". Since there is a great number of real circumstances in which the Markov property is not valid, for example when deterministic times occur, non-Markovian models are used for this type of problems.

In past years, several classes of non-Markovian approaches have been defined [28], such as Semi-Markov Stochastic Petri Net (SMSPN's) [43], Markov Regenerative Stochastic Petri Nets (MRSPN's) [40] and Deterministic and Stochastic Petri Nets (DSPN's) [2]. Three of the major methods for analytically solving the non-Markovian models are discussed in [27, 88, 63]: i) associating "supplementary variables" to non-exponential random variables, ii) using a sequence of exponential stages to approximate a non-exponential random variable (phase-type expansions) and iii) searching for embedded epochs in the system evolution where the Markov property is valid.

Many of the existing modeling techniques are supported by automated tools for the assisted construction and solution of dependability models. Here, we recall just Möbius and DEEM, which are those of specific interest for the ART DECO project.

Möbius [44] Möbius provides an infrastructure to support multiple interacting modeling formalisms and solvers. Formalisms supported are: SAN's, Buckets and Balls (a generalization of Markov chains), and PEPA (a process algebra). Möbius allows to combine (atomic) models to form the *Composed model*. To this purpose, it supports the two operators *Rep* and *Join* to compose sub-networks. It supports the transient and steady-state analysis of Markovian models, the steady-state analysis of non-Markovian DSPN-like models [105], and transient and steady-state simulation. More information can be found in the web site http://www.crhc.uiuc.edu/PERFORM.

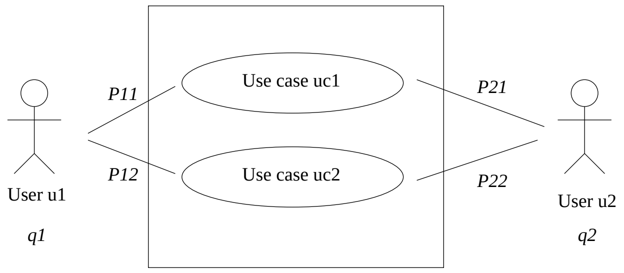


Figure 3.5: Annotated Use Case Diagram.

DEEM [29] DEEM supports the methodology proposed in [90, 30, 89] for the dependability modeling and evaluation of Multiple Phased Systems (MPS's), relied upon DSPN as a modeling formalism and on MRGP for the model solution. In its present version, the duration of the phases is deterministic and the phase model is restricted to contain only exponential and instantaneous transitions. Moreover, in every non-absorbing marking of the DSPN there is always one deterministic transition enabled, which corresponds to the phase being currently executed. Specialized solution for transient analysis is supported. More information can be found in the web site http://dcl.isti.cnr.it/DEEM.

3.2.2 Reliability prediction from UML models

The ability to validate software systems early in the development lifecycle is becoming crucial. While early validation of functional requirements is supported by well known approaches, the validation of non-functional requirements, such as reliability, is not. In this chapter we introduce a methodology that starts with the analysis of the UML model of software architecture followed by the bayesian framework for reliability prediction. We utilize three different types of UML diagrams: Use Case, Sequence and Deployment diagrams. They are annotated with reliability related attributes

Annotating Use Case Diagrams

A *Use Case Diagram* (UCD) provides a functional description of a system, its major scenarios (i.e., use cases) and its external users called actors (an actor may be a system or a person). It also provides a graphic description of how external entities (actors) interact with the system.

Figure 3.5 presents a very simple UCD annotated for the reliability assessment purposes[1]. In this case, two types of users and two use cases are considered. The annotations introduced are the same as in [108]: q_1 and q_2 represent the probabilities for users (or groups of users, each sharing similar system usage patterns) u_1 and u_2, respectively, to access the system by requesting certain services. P_{11} and P_{12} represent the probabilities that user u_1 requests the functionality f_1 or f_2, respectively. P_{21} and P_{22} have a similar meaning with respect to user u_2.

[1]Note that this type of UCD annotations have been used for performance assessment purposes in [46].

In general, the probability of executing the use case x is given by:

$$P(x) = \sum_{i=1}^{m} q_i \cdot P_{ix} \tag{3.1}$$

where m is the number of user types.

Furthermore we assume that, for each use case, the set of all the relevant Sequence Diagrams (representing main scenarios within the use case) have been identified and specified. The assignment of use case probabilities, as in equation (3.1), infers that the same probability is assigned to the execution of every Sequence Diagram (SD) within the set corresponding to the use case. But in general, given a set of SDs, not all of them will have the same probability of execution. Hence, if we are able to assign a non-uniform probability distribution to the SDs referring to the same use case, equation (3.1) gives rise to the following equation [108]:

$$P(k_j) = P(j) \cdot f_j(k) \tag{3.2}$$

where $f_j(k)$ is the frequency of the k-th overall the SDs referring to the j-th use case. Parameter $P(k_j)$ represents the probability of a scenario execution.

Annotating Sequence Diagrams

Sequence Diagrams (SDs) depict how groups of components interact to accomplish a given task. As mentioned in section 3.2.2, we assume that the behavior of each use case is given by a set of SDs. Interactions are drawn along a time axis, thus defining a partial order of execution. Hence, SDs provide specific information about the order in which events occur and can provide the information about the time required for reacting to events.

When an interaction enters component's axis (i.e., the component receives a service request), the component becomes busy. We therefore assume that a component is busy during interval of time that starts with an entering interaction and ends with the corresponding exit interaction [108]. In Figure 3.6, a Sequence Diagram specifically annotated for our reliability model is shown.

It is easy to count the number of busy periods that component C_i experiences in a given scenario (Sequence Diagram). For example component C_3 in Figure 3.6 has two busy periods, the first one from the interaction labeled l_1 to l_2, and the second one from l_3 to l_4. In general, let us denote by bp_{ij} the number of busy periods that the component C_i shows in the Sequence Diagram j. If we assume that an estimate of the failure probability θ_i for every component C_i is available, then, as a first approximation, we can get the estimate of the probability of failure θ_{ij} of the component i in the scenario j from the following equation [108]:

$$\theta_{ij} = Prob(failure\ of\ C_i\ in\ scenario\ j) = 1 - (1 - \theta_i)^{bp_{ij}} \tag{3.3}$$

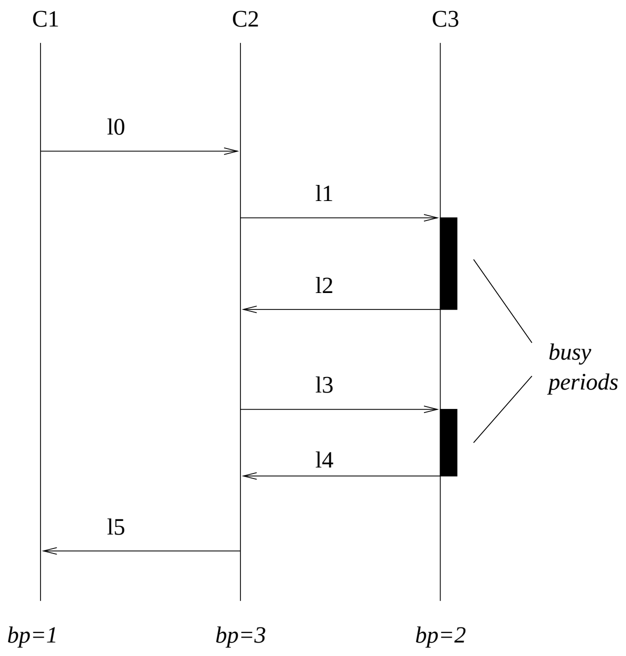


Figure 3.6: Annotated Sequence Diagram.

Annotating Deployment Diagrams

A Deployment Diagram shows the platform configuration where the software application is targeted to run. Nodes represent platform sites (e.g., workstations, PCs, etc.) and links represents hardware/logical connectors (e.g., LAN, WAN, etc.). Additional boxes represent software components and are placed into the respective sites where they are supposed to be loaded. In practice this diagram shows the mapping of components to sites.

The reliability of communication in distributed software can be critical, especially in unsafe environments. The annotation of a Deployment Diagram with the probabilities of failure over the connectors among sites (possibly a priori estimated) allows the reliability model to embed the communication failures. In fact, based on to these annotations a failure probability can be assigned to each interaction of a Sequence Diagram. The failure probability of an interaction between two remote components is the one over the connector linking the sites that host the components (it is reasonable to consider, as we do, fully reliable the communications among components residing on the same site).

Figure 3.7 shows an example of annotated Deployment Diagram including components in Figure 3.6. We number the failure probabilities over the connectors as ψ_1, ψ_2, etc., each corresponding to the failure probability of all the communications over the connector it is annotated on. Therefore, each pair of components (l, m) communicating over the connector i is subject to a failure probability ψ_i.

If we denote by $|Interact(l, m, j)|$ the number of interactions that components l and m exchange in the SD j (note that this quantity is straightforwardly obtainable by visiting

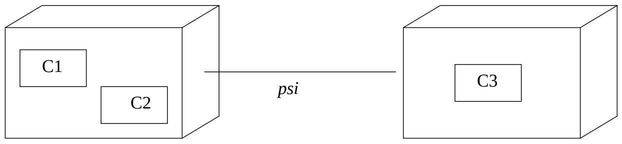


Figure 3.7: Annotated Deployment Diagram.

the SD), then the contribute ψ_{lmj} to the reliability of communication between these components (assuming it occurs over the connector i) in the scenario j is as follows:

$$\psi_{lmj} = (1 - \psi_i)^{|Interact(l,m,j)|}. \tag{3.4}$$

Equation (3.4) takes also into account two special cases:

- *Two software components are co-located in the same platform site:* the communication between them can be fairly considered totally reliable, because it does not involve any physical connection, that is $\psi_i = 0$ for these components.

- *Two components, wherever located, have no interactions:* trivially it solves into $|Interact(l,i,j)| = 0$, bringing a neutral contribution 1 to the connection reliability product.

One can argue that mapping of components to sites may not be available early in the lifecycle. In this case our model can nicely work without the contribution of communication failures (see [108]). However, if several mapping alternatives may be designed, equation (3.4) can be easily instantiated to produce the reliability of each alternative. In the latter case, the model may therefore support the comparison of different mappings based on reliability issues.

The UML-based reliability approach summarized in this section has been applied over a simplified WEB-based transaction processing system. This system, illustrated in Figure 3.8, has been modeled and then analyzed in [47]. The numerical results reported in [47] support the claim of effectiveness and robustness of the proposed approach to reliability prediction of component based systems.

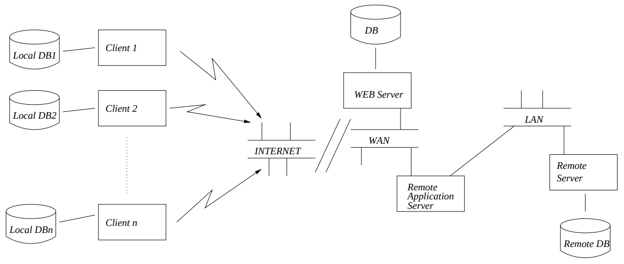


Figure 3.8: WEB-based example.

3.2.3 Performance Analysis

For performance analysis we apply an approach based on the Software Performance Engineering (SPE) framework [110]. In this framework a separation of the Software Model (SM) from its environment/machinery model (MM) is introduced. This distinction allows the designer to separately define software and machinery models and to solve their combination and, on the other hand, improves the portability of both models (i.e., the performance of a specific software system can be evaluated on different platforms, and the performance of a specific platform can be validated under different software systems). SM captures essential aspects of software behaviour and is based on Execution Graphs (EG). An EG is a graph that includes several types of nodes, such as basic, cycle, conditional, fork and join nodes. Each basic node represents a software workload component, that is a set of instructions or procedures performing a specific task. It is weighted by a demand vector that represents the resource usage of the node (i.e., the amount of each resource required to perform the task). Edges represent transfer of control. MM is the model of the hardware platform and is based on Extended Queueing Network Models (EQNM) [77]. EQNM are extensively applied in the literature for the modelling of resource contention. In order to specify and parameterize an EQNM it is necessary to define: components (i.e., service centers), topology (i.e., connections among centers) and parameters (such as job classes, job routing among centers, scheduling discipline at each service center, service demand at each service center). Components and topology are given by the system specification, while the specification of parameters needs the support of information from SM. Upon parameterization completion, the EQNM has to be solved to obtain the values of the performance indices of interest. Hence, the steps of the model solution are the following: processing the EG with reduction analysis techniques to obtain software-based parameters (i.e., job classes and routing) and mapping them onto the EQNM; solving the parameterized EQNM with classical techniques based on analysis and/or simulation [77]

The derivation of analysis models following the framework described in Section 2.3 implies the definition of MOF metamodels for EG and EQN models and of precise transformations rules as described in [65].

Figures 3.9 and 3.10 depict possible MOF metamodels for EG and EQN models, respectively. Each Node in the EG metamodel is characterized by attributes such as name, time-demand, resource-type and service-name (demand vector). The node type (control, basic, É) derives directly from the EG terminology [110]. The transfer of control is modeled through simple predecessor/successor associations or by way of more complex Control nodes modeling, for example, the possibility of selecting among different successors (branch) or the repetition of a certain number of nodes (loop).

The proposed EQN metamodel is based on the widely used notation for EQN presented in [77]. As an additional feature, we have tried to maintain in this metamodel the separation of concerns between SM and MM typical of the SPE approach. To this end we define a EQN as composed by, on one side, a set of Routing Chains modeling the workload and, on the other side, as a set of centers modeling the different kind of EQN centers. For the sake of simplicity we omit here a complete definition of the Routing chain

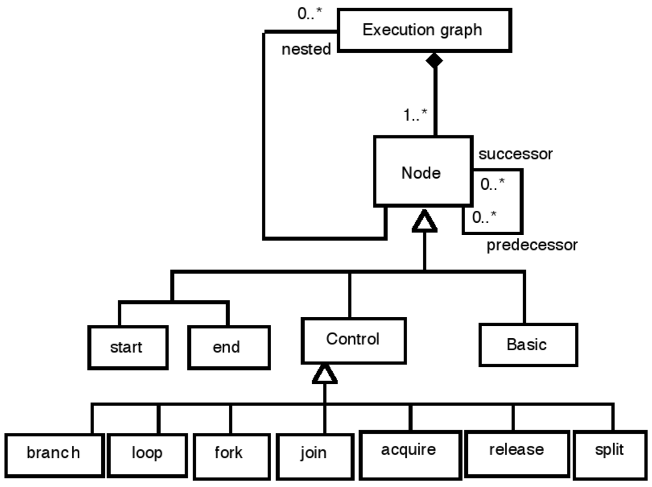


Figure 3.9: The EG metamodel

metaclass. Figure 3.10 depicts only its relationships with EGs, indicating that a routing chain is derived from an EG. The connection between these two sides of an overall EQN model is represented through the Demand association that links the different resource demands of a single EG node to the related EQN centers. A center in a EQN can be Active (with attributes such as Number of servers, Scheduling discipline, service-rate, service-distribution), Passive (whose attributes are the Token number and the service discipline) or Special modeling for example, the source or the sink of a workload, or parallel thread of execution (Fork and Join). Each Special node is, in turn, characterized by a suitable attribute; the source node attributes, for example, are the arrival rate and the arrival process distribution.

The overall transformation from a KLAPER model to a performance model based on EG and EQN consists of several steps that can be realized using QVT relations as described in [65]. At the end of this transformation process, it is possible to obtain the EG and the EQN models of the source model. Figure 3.11 shows an example of the EG model that one can obtain applying these transformations.

The final step of a transformation from KLAPER to EQN models is the EQN parametrization consisting in the construction of the routing chains modeling the EQN dynamics. Roughly speaking, a routing chain should be built for each EG derived from the behavior associated with a workload in the KLAPER model.

Figure 3.11 shows an example of the parameterized EQN model that one can obtain applying these transformations.

The obtained EQN model can be solved to obtain several results such as: the completion time for the application (or for a single application activity), the resource utilization, the better resources distribution with respect to a certain completion time and so on.

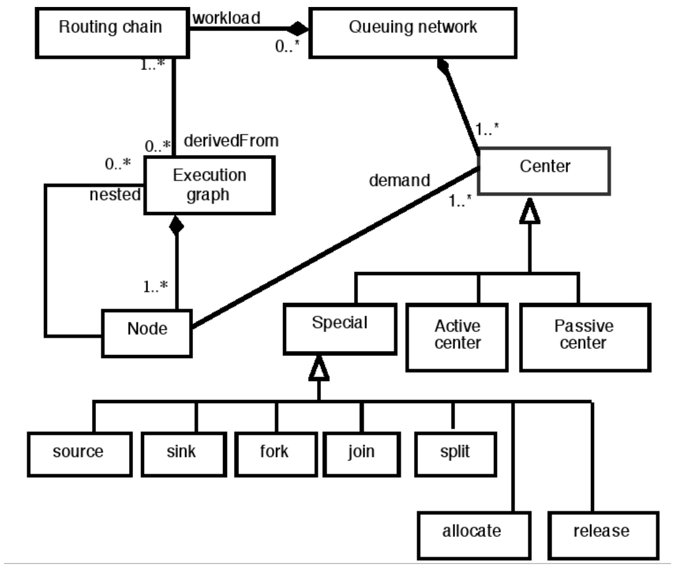


Figure 3.10: The EQN metamodel

3.2.4 Within ART DECO

Early QoS quantitative assessment in ART DECO will be pursued through model-based analysis, mainly in terms of reliability, availability, performability and performance indicators. By adopting the methods/techniques reviewed in Section 3.2.1, specific aspects of the project's developments will be verified and validated through an off-line approach. With reference to the structure depicted in Figure 1.1, this activity mainly focuses on V&V at architectural level as a support to both architectural choices and to the decision making process performed by the autonomic system manager. In addition, the component-based structure of the system architecture in 1.1 (where components are meant as basic entities that can corresponds to object, services, etc.) is well suited to apply existing approaches that, starting from the QoS of composing elements, build up QoS models of the whole architecture.

Model-based evaluation for design The design of the ART DECO architecture necessitates verification activities to be performed as soon as possible since the early phases of the design process, in order to justifiably trust the identified solutions and to make appropriate choices among several possible alternatives. Model-based validation could be promoted inside the ART DECO framework to this purpose. Both analytical and simulative models will be pursued, as a support to the verification of the adequacy of the envisaged solutions with respect to the imposed requirements, and to guide the refinement process necessary to improve on deficient choices.

Model based evaluation for decisions making Besides its usage as a support to the design activities, model-based evaluation techniques could be profitably employed as a sup-

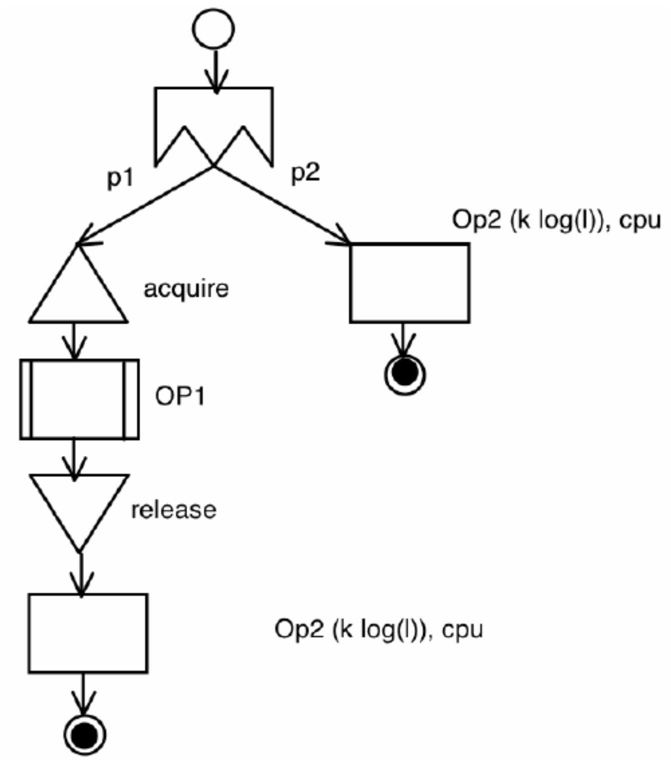


Figure 3.11: Example of EG model

port to the decision-making process performed by the autonomic system services/components. In fact, to properly react to malfunctions or simple variations of user needs and/or environmental conditions, and to optimize resource assignment, quantitative assessments of the benefits deriving from applying a certain reaction/reconfiguration are very important. Both transient (in a pre-defined interval of time following the specific reconfiguration action) and steady-state analysis are useful in this context. In fact, it might happen that during the transient period, performance is temporarily worse than before the reconfiguration, but then it becomes better when the system reaches the new steady state. However, it is an important issue to evaluate point-wise performance to prevent that the system degrades under a given acceptable level and to measure the time to reach the expected steady-state effect.

3.3 Testing

The testing phase is an important and critical part of software development, consuming even more than half of the effort required for producing deliverable software [14]. Unfortunately, often due to time or cost constraints, the testing is not developed in the proper manner or is even skipped. Beside exhaustive verification techniques, such as model checking, many times a testing phase is required for facing the complexity of the applications to be verified or evaluating specific qualities and properties of the software. Software testing, accordingly with a recent definition, can be characterized as [17]:

"Software Testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior“.

Of course the management of the testing activities depends strictly on the development process adopted for delivering the software products; however the main phases can be resumed in [17]:

Planning: As for any other process activity, the testing must be planned and scheduled. Thus the time and effort needed for performing and completing Software Testing must be established in advance during the early stages of development. This also includes the specification of the personnel involved, the tasks they must to perform and the facilities and equipments they may use.

Test cases generation: According to the test plan constraints a set(s) of the test cases must be generated by using a (several) test strategy (ies).

Test cases execution: The test cases execution may involve testing engineers, outside personnel or even customers. It is important to document every action performed in order to allow the experiments' duplication and meaningful and truthful evaluations of the results obtained.

Test results analysis: The collected testing results must be evaluated to determine whether the test was successful (the system performs as expected, or there are no major unexpected outcomes) and used for deriving measures and values of interest.

Problem reporting: A test log documents the testing activity performed. This should contain for example the date in which a test was conducted, the data of the people who performed the test, the information about the system configuration and any other relevant data. Anomalies or unexpected behaviors should be also reported.

Post-closure activities: the information relative to failures or defects discovered during testing execution are used for evaluating the performance and the effectiveness of the developed testing strategy(ies) and determining whether the process development adopted needs some improvements.

All these activities have in common the same testing purpose: evaluating the product quality for increasing the software engineering confidence in the proper functioning of the software.

In this section, considering the above subdivision of activities, we focus on test cases generation, execution and test result analysis.

In particular considering the test cases generation there are several testing techniques focused on the verification of functional and non functional properties. With particular regards to service testing, Bai *et al.* [8] defined a black-box strategy for test data generation starting from an XML schema. Recently, Martin *et al.* proposed a preliminary framework to automatically perform Web service robustness testing [83], while Fu *et al.* highlighted the need for a proper testing of exception code [58]. It is out of the scope of this deliverable providing an exhaustive survey of the possible testing approaches. An exhaustive description of testing approaches can be found in books such as [14, 25], while approaches specific to service testing are described in [50].

Among them, exploiting the background and the knowledge of the ART DECO participants, we present in this section a general view of some testing approaches for web service, web service composition and service oriented architectures in general, looking at behavioral correctness and at the reached quality of service. Moreover, we overview a number of techniques to generate test cases for both operational and pure logic models. In particular in the rest of this section we briefly presented two different testing approaches:

- A data flow-based validation method useful for verifying specific functional data properties in the service composition (Section 3.3.1).

- A Evolutionary test data generation method [51] suitable for the verification of specific non functional properties (Section 3.3.2).

The approaches are complementary and focus on two specific aspects of the service composition verification. The former exploits the data requirements a for deriving test case able to verifying overall compositional properties. Further details are provided in Section 3.3.1. The latter aims at automatically generating test cases, consisting in service composition inputs and bindings, that cause violations in the Service Level Agreements (SLA. Further details are provided in Section 3.3.2.

The two approaches address two important issues of service testing (as highlighted in [35]), i.e., security issue and the need for keeping the Quality of Service (QoS) within the SLA negotiated between service provider and consumers.

For aim of completeness within the ART DECO project we will propose a first design of an integrated testing environment, in which the derived test cases can be executed. In particular section 3.3.3 we present the first structure of a stubs generator for testing functional and non functional properties. This represents the basis for the definition of the testing environment. Its refinements and further specification will be planned during the remaining part of the ART DECO project.

3.3.1 Data Flow-based Validation of (Web) Service Compositions

This deliverable presents proposals for adapting existing validation approaches to the validation of (web) services and (web) service composition. They mainly use behavioral specifications, extracted from the BPEL description of the composition, as an input to formal verification. However proving the behavioral correctness of the system may not be sufficient to guarantee that specific data properties are satisfied [102]. Inside the project ART DECO we focus on the specific task of data validation and even modeling. The approaches proposed are general enough to be adapted to the ART DECO specific architecture. Most of the times data-flow requirements and data properties are just informally expressed in natural language, and consequently cannot be adequately verified. Data modeling represents an important aspect to be considered during the implementation of a composition, as data-flow relationships and requirements provide an alternative view of the composition problem with respect to the functional-oriented view, which should be taken in consideration during both the implementation and the testing phase. Of course the usage of data information for verification purposes is not a novelty. Several data-flow oriented test adequacy criteria have been proposed in the past. In ART DECO the purpose is to explore the possible ways of exploiting data information for the validation before and during the implementation of a web service composition. This section is structured in two main parts: an overview of recent research work on validation of web services and web service compositions; some proposal for data-flow modeling and for the validation of web services composition.

Overview of Recent Proposals

This section is a brief overview on recent investigations on WSs validation either regardless of composition issues or addressing specifically WSs composition or focusing on fault or failure models for WSs compositions.

In the Coyote framework [114], test data are selected among monitored data and other manually produced test data, according to their fault detection ability. The latter is assessed thanks to a mutation coverage criterion defined on contracts [75] (a similar approach is proposed in [106] where mutations are defined on the WSDL language). Coyote requires user-provided execution scenarios as MSCs [113] which are processed together with the WSDL specification of the WSs to automatically generate test scripts.

Another approach proposed in [21, 16] focuses on testing the conformance to a specified access protocol of a WS instance. The authors propose to augment the WSDL description with a UML2.0 Protocol State Machine (PSM) describing how the service provided by a component can be accessed by a client through its ports and interfaces. The PSM is translated into a Symbolic Transition System (STS), to which existing formal testing theory and tools can be applied for conformance evaluation, as for instance in [57] where STSs are used to specify the behavior of communicating WS ports and test data are generated to check the conformity of the effective implementation to such a specification.

Validation of compositions of WSs has been recently addressed by few investigations. Some of them [122, 124] focus on the structural coverage of the composition specification, considering that it is provided in BPEL, the standard language for programming WSs compositions. In [60], a transformation is proposed from BPEL to PROMELA (similarly to [37]). The resulting abstract model is used with the SPIN model-checker to generate tests guided by structural coverage criteria (such as transition coverage).

Compositions of WSs can also be formally verified as soon as formal models of the composition and of required properties are provided. For instance, in [91] workflows are described as Petri Nets and then simulated to verify properties such as reachability.

Similarly, a transformation of BPEL processes in Colored Petri Nets (CPN) has been proposed [123]. Another formal approach is proposed in [56]. The workflow is specified in BPEL and an additional functional specification is provided as a set of MSCs. These specifications are translated in the Finite State Processes (FSP) notation and model-checking is performed. The final goal is to detect execution scenarios allowed in the MSC description and that are not executable in the workflow and, conversely.

The above investigations use models of the composition behavior and of properties or scenarios expressing the user expectations (MSCs or state based properties such as the absence of deadlock). A different characterization of failures of WSs compositions is proposed in [119, 120] where failures are considered as interactions between WSs, similarly to feature interactions in telecommunication services, and classified as goal conflict, resource contention, deployment-ownership decisions related problem, assumption violation, information hiding, policy conflict or wrong invocation order.

The underlying models of all the above approaches - workflows, scenarios, user goals, state-based properties - focus on control. The verification of the data transformations involved in the WSs composition execution does not seem to have been explored so far. From the modelling point of view, this lack has been outlined in [82] where dependencies between data exchanged during the execution of a WSs composition are explicitly modeled by means of an ad hoc notation. There is no literature on fault models based on data for WSs compositions. However, we could mention a proposition of data fault model for workflows [102]. According to this model, data can be *redundant* if they are produced by an activity, but not used by any other activity. They can be *lost*, if the outputs of two concurrently executed activities are assigned to a single variable in a non deterministic order (so, one of the outputs may be lost) or *missing*, if an input activity expects data that are not specified as outputs of another activity. They can be *mismatched*, if the expected input data do not match with the actual data sent to an activity. *Inconsistent data* correspond to corrupted variables. Data can be *misdirected* if an activity A expects data from an activity B while A is prior to B in the workflow. Finally, data may be *insufficient* to complete the workflow goals (this is mainly a specification problem).

Using Data-flow for testing Web Services Composition

Data related models To illustrate the usefulness of data flow modelling for testing purposes, we refer to a simplified version of the Virtual Travel Agency (VTA) example used in [82]. A VTA service offers travel packages to customers, by combining two independent existing services: a flight booking service (FBS), and a hotel booking service (HBS). HBS receives the date and the location using the ports *H.request.date* and *H.request.loc*, respectively.

The ports *H.offer.cost* and *H.offer.hotel* are used for returning the cost and other hotel information. In a similar way, FBS uses the ports *F.request.date* and *F.request.loc* for receiving flight booking requests for a given time period and location, while *F.offer.cost*, *F.offer.schedule*, *F.booked.info* for returning the cost, the schedule and other flight information.

In both cases, the offer can be accepted or canceled through the customer interface,

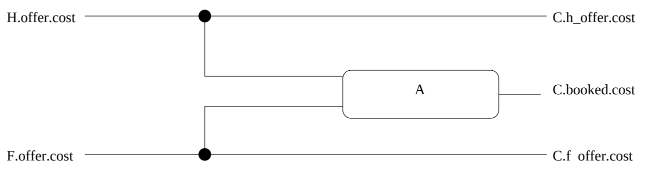


Figure 3.12: DFM example

which is a service provided by the VTA and invoking HBS and FBS. Customers can ask for a travel package providing the dates and location (output ports *C.request.date, C.request.loc*) and be informed about the proposed flight (*C.f_ offer.schedule, C.f_ offer.cost*), hotel (*C.h_ offer.hotel, C.h_ offer.cost*) and the package cost (*C.booked.cost*). Figure 3.12 provides an abstract data-flow model (DFM) expressing dependencies between the flight and hotel offered costs and the cost proposed to the customer.

According to this model, the hotel cost offered to the customer (*C.h_ offer.cost*) must be equal to the cost returned by the HBS (*H.offer.cost*) and similarly for the flight cost. The whole package cost (*C.booked.cost*) is computed from the selected hotel and flight costs by means of the function A (this function collects costs and may add various fees).

DFMs can be useful in the specification of WSs compositions, since they highlight the goal of the composition from the data point of view. Building a DFM forces modelling the implicit knowledge on data, avoiding loss of information or misdirection of data flow and highlighting the most critical data flow paths. A DFM can also be used, notably, to define test coverage criteria and test strategies. Furthermore, data-flow relationships and requirements provide an alternative view of the problem with respect to the functional-oriented view. Assuming that a DFM is provided for the WSs composition, we explore in the rest of this section the applicable verification and validation approaches that could use this model, and identify challenges for research in this field.

Data fault models, as the classification mentioned in the previous section [102], could also be useful in the validation process. In the context of WSs composition, some of the identified problems in this classification do not apply when the standard languages BPEL and WSDL are used, since these languages ensure that the data exchanged between services conform to a mutual accepted specification (see Table 3.1). In order to be able to formally identify and to automate the verification of such data problems, a model of the data used within the WSs composition is needed. Hence, data modeling is an important issue in the design, the implementation and the validation of a composition.

Kinds of models referred to Performing data-flow based validation may involve several kinds of models than can be combined in various ways. In ART DECO project we consider that one or more of the following models may be available:

- A DFM, defined before the implementation of the composition and independently from any behavioral specification of the composition. It expresses dependencies between the data handled or exchanged during the execution of the WSs composition.

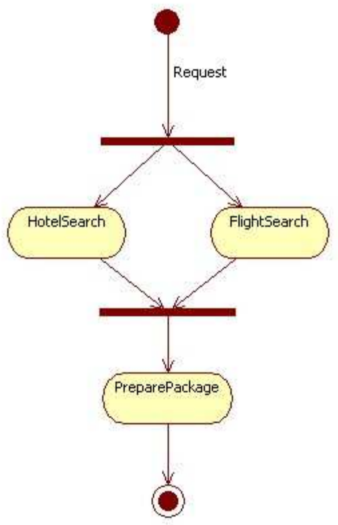


Figure 3.13: VTA composition example

Data validation problem	Definition for WS
Redundant data	Statically detected - conformance between service invocation and WSDL description
Lost data	Can occur when two data-flows involving the same variable definition merge in a single flow: one of the values can be lost.
Missing data	Statically detected conformance between service invocation and WSDL description
Missmatched data	Statically detected conformance between service invocation and WSDL description, except if there are assumptions other than those expressed in the WSDL file.
Inconsistent data	Can occur when a data-flow involving a call to WS merges, after this call, with a data flow updating a variable containing a value coming from the WS in other words, the BPEL program may uncorrectly update this variable
Misdirected data	Nothing specific to WS.
Insufficient data	Specification problem.

Table 3.1: Data validation problems for WSs

Used model		Additional models		
Initial	**Target**	**1 None**	**2 BPEL**	**3 Data fault model**
1 None	DFM	Structural coverage of DFM Black-box test generation	Static verification (BPEL vs. DFM)	Coverage of DFM/fault model Black-box test generation
2 BPEL	DFM	Structural coverage of DFM Black-box test generation	Define coverage criteria for BPEL Guide test generation (to achieve BPEL coverage)	Coverage of DFM/fault model Black-box test generation
3 None	Data properties	Usual black box testing (category partition)	Usual data-flow testing	Coverage - Guide test generation

Table 3.2: Data-flow based validation issues

- A behavioral model of the WSs composition. It is possible to derive, from this model, a DFM explicitly focusing on the service data interactions. An example WSs composition for the VTA is given in Figure 3.13 (alternatively, a BPEL process could also be provided).
- A model defining classes of faults related to data (data fault model). Table 3.1 is an example of such a classification that could be used as a fault model.
- In addition to these models, we can consider properties focusing on data, written in a formal language. While a DFM focuses on dependencies between data, properties may restrict the domain of the computed values or express relations between them (for instance, $C.h\_offer.cost + C.f\_offer.cost \leq C.booked.cost$).

Table 3.2 summarizes the above mentioned models and highlights how testing could be performed in presence of one or more of these models within the ART DECO project.

3.3.2 Evolutionary test data generation

Search-based optimization techniques have been successfully applied to tackle different testing problems, and in particular to generate testing data. Most of the relevant references on that topics are reported and discussed in a survey by McMinn [84].

In recent years, the use of metaheuristic search techniques for the automatic generation of test data has been of great interest. It is known that enumeration of all program's input is infeasible for any reasonably-sized program. Random search is unreliable and unlikely to find features of software that are not exercised by mere chance. Metaheuristic search techniques utilizes heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. The problem may be NP-complete or NP-hard, or a problem for which a polynomial time algorithm is known to exist but is not practical. Metaheuristic search techniques represent strategies ready for adaption to specific problems and are not standalone alghoritms. For test data generation, this involves the transformation of test criteria to *objective functions*. Objective functions compare and contrast solutions of the search with respect to the overall search goal. Using this information, the search is directed into potentially promising areas of the search space. These techniques have been applied to automate test data generation in the following areas:

- the coverage of specific program structures, as part of a structural, or white-box testing strategy;
- the exercising of some specific program feature, as described by a specification;
- attempting to automatically disprove certain grey-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example the worst case execution time of a segment of code.

In order to adapt a metaheuristic search technique to a specific problem, a number of different decisions have to be made. For example the way in which solutions should be encoded so that they can be manipulated by the search, so that the search will be allowed to move easily from one solution to another that shares a similar set of properties. These movements are dependent on the evaluation of candidate solutions, performed using a problem-specific objective function. The search seeks "better" solutions using knowledge and experience of previous candidates and feedback from the objective function. A good objective function is therefore critical to the success of the search. Solutions that are "better" in some respect should be rewarded with better objective values, whereas poorer solutions should be punished with poorer objective values. Whether a "better" objective value is, in practice, a higher value or lower value, is dependent on whether the search is seeking to minimize or maximize the objective function.

The metaheuristic techniques that have been used in software test data generation are Hill Climbing, Simulated Annealing and Evolutionary Algorithms.

Hill Climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighbourhood of this solution is investigated. If a better solution is found, then this replaces the current solution. The neighbourhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again, and so on, until no improved neighbours can be found for the current solution.

Simulated Annealing is similar in principle to Hill Climbing. However, by allowing for a probabilistic acceptance of poorer solutions, Simulated Annealing allows for less restricted movement around the search space. The name "Simulated Annealing" originates from the analogy of the technique with the chemical process of annealing.

Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection. Genetic Algorithms (GAs) are the most well known form of Evolutionary Algorithm. For GAs, the search is primarily driven by the use of recombination — a mechanism of exchange of information between solutions to "breed" new ones — whereas Evolution Strategies principally use mutation — a process of randomly modifying solutions. Although these different approaches were developed independently, and with different directions in mind, recent work has incorporated ideas from both.

A classification of Evolutionary Testing Techniques

The application of Evolutionary Algorithms to test data generation is often referred as Evolutionary Testing. Different techniques can be categorized on the basis of objective function construction.

Coverage-Oriented Approaches reward individuals on the basis of covered program structures. In the work of Roper [101], an individual is rewarded on the basis of the number of structures executed in accordance with the coverage criterion. The search, however, tends to reward individuals that execute the longest paths through the test object. The work of Watkins [6] attempts to obtain full path coverage for programs. The objective function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search. The direction of the search is constantly changed. However, the penalization of covered paths, in itself, provides little guidance to the discovery of new, previously unfound paths.

Structure-Oriented Approaches take a "divide and conquer" approach to obtaining full coverage. A separate search is undertaken for each uncovered structure required by the coverage criterion. Structure-oriented techniques differ in the type of information used by the objective function. These can be categorized as:

- *Branch-Distance-Oriented* approaches exploit information from branch predicates. In the work of Xanthakis et al. [121], GAs are employed to generate test data for structures not covered by random search. A path is chosen, and the relevant branch predicates are extracted from the program. The GA is then used to find input data that satisfies all the branch predicates at once, with the objective function summing branch distance values. The tester must select the path. Jones *et al.* [7] obtain branch coverage without path selection. The objective function is simply formed from the branch distance of the required branch. However, no guidance is provided so that the branch is actually reached within the program structure in the first place. McGraw *et al.* [59] alleviate this problem for condition coverage, by delaying an attempt to satisfy a condition within a branching expression until previous individuals have been already found which reach the wanted branching node.

- In *Control-Oriented* approaches, the objective function considers the branching nodes that need to be executed as desired in order to bring about execution of the desired structure. The approach of Jones *et al.* [7] to loop testing falls into this category. Here, the objective function is simply the difference between the actual and desired number of iterations. Pargas *et al.* [99] use the control dependence graph of the test object for statement and branch coverage. Let *dependent* be the number of control dependent nodes for the current target, and *executed* the number of control dependent nodes successfully executed in the required manner. A minimizing version of the objective function of Pargas *et al.* , can be computed as $dependent - executed$.

- *Combined approaches* make use of both branch distance and control information for the objective function. The work of Tracey [111] builds on previous work which used Simulated Annealing. The control dependent nodes for the target structure are identified. If an individual takes a critical branch from one of these nodes, a distance calculation is performed using the branch predicate of the required, alternative branch. The number of successfully executed control dependent nodes are used to scale branch distance values. Let *branch_dist* be the branch distance calculation performed at the branching node where a critical branch was taken. The formula used by Tracey for computing the objective function is:

  $(executed/dependent) * branch\_dist$

  Unfortunately, this scheme can lead to unnecessary local optima in the objective function landscape. Wegener *et al.* [117] map branch distance values *branch_dist* logarithmically into the range [0, 1] (hereby referred as *m_branch_dist*). The minimizing objective function is zero if the target structure is executed, otherwise, the objective value is computed as:

  $(dependent - executed - 1) + m\_branch\_dist$

  The $(dependent - executed - 1)$ sub-calculation is referred to as the *approximation level* or the *approach* level attained by the individual. However, the extra information provided by the branch distance calculation prevents the formation of plateaux at each approach level.

Evolutionary testing of non-functional aspects

Wegener and Grochmann applied Genetic Algorithm (GA) for testing the temporal correctness of real-time systems [118]. Briand *et al.* [32] used GAs for stress testing of real-time systems. However, in their case the problem was mainly to determine schedule causing failures. Garousi *et al.* [62] perform stress testing on UML models of distributed systems, also accounting for network traffic. This is also a viable solution for Web services, although it requires the Web service behavior to be modeled as a UML model, which may or may not reflect the actual behavior.

3.3.3 Stub Generator for Testing of QoS

The openness of the environment characterizing the Service Oriented Architecture (SOA) paradigm naturally led to the pursuit of mechanisms for defining Quality of Service (QoS) level agreement specifications. Nowadays the idea is widely accepted that an effective software design process cannot only focus on functional aspects, ignoring QoS-related properties. For Service Oriented systems, as well as for many other kind of complex enterprise applications [18], communication networks and embedded systems [15] it is certainly no longer possible to propose solutions without adequate consideration of their extra-functional aspects [79].

Nevertheless, traditionally agreements have been not machine-readable. In software engineering only basic notion of agreements have been experimented by means of Interface

Description Languages [78, 79]. In recent years both industry and academia have shown a great interest on this topic. Concerning the Services Oriented technologies, Service Level Agreements (SLAs) represent one of the most interesting and active issues. SLAs aim at ensuring a consistent cooperation for business-critical services defining contracts between the provider and client of a service and the terms governing their individual and mutual responsibilities with respect to these qualities [109]. Usually a SLA contains a the technical QoS descriptions with the associated metrics. These information are referred as Service Level Specifications (SLSs). In the following a brief description of two languages for service level agreement specification are reported.

The QoS aspect of the off-line validation stage concerns an approach for the automatic derivation of test harnesses. We call such approach PUPPET . The goal is to evaluate different QoS characteristics for a service under development and before its final deployment. In particular, such approach focuses on assessing that a specific service implementation can afford the required level of QoS (e.g., latency, reliability and workload) defined in a corresponding Service Level Specification (SLS) for a composition of services (choreography/orchestration) in which the Service Under Evaluation (SUE) will play one of the roles.

The technologies beckground assumend to exists include for each service a specification describing the functional interface exported by the service (e.g. WSDL [116]), a description of the services that compose it (e.g. in terms of WSBPEL [92]), and a machine readable specification of the QoS agreement for the services in the composition. At this point, the goal of the tool for the QoS evaluation in the Off-line validation stage is to automatically generate a test harness to validate the implementation of a service before its deployment in the target environment.

The generation of the test harness proceeds through two different phases. The first one is the generation of the stubs simulating the extra-functional behavior of the services in the composition; the second one, instead, foresees the composition of the implementation of a service, called “S1i” in Figure 3.14, with the services with which it will interact. In the following, both phases will be described to give a complete overview of the approach.

The generation of the stubs consists in turn of two successive sub-steps (see Figure 3.15). In the first one the skeletons of the stubs are generated starting from the functional interface description of the service (e.g. WSDL). The generated skeletons contain no behavior. Hence, in the second sub-step the implementation is “filled in” with some behavior that will fulfill the required extra-functional properties for the service corresponding to the stub. This step is carried on retrieving the information from the SLS and applying automatic code transformation according to previously defined patterns matching each SLS with a portion of code that simulates its behavior. At the end of the first phase, a set of stubs providing the services specified in the composition according to the desired properties are available.

Back to the proposed approach illustrated by means of Figure 3.14, the second phase concerns the setting of the test harness. The goal of this step is to derive a complete environment in which to test the service. To this purpose, the SUE, “S1i” in Figure 3.14, is composed with the required service and according to the composition specified in the

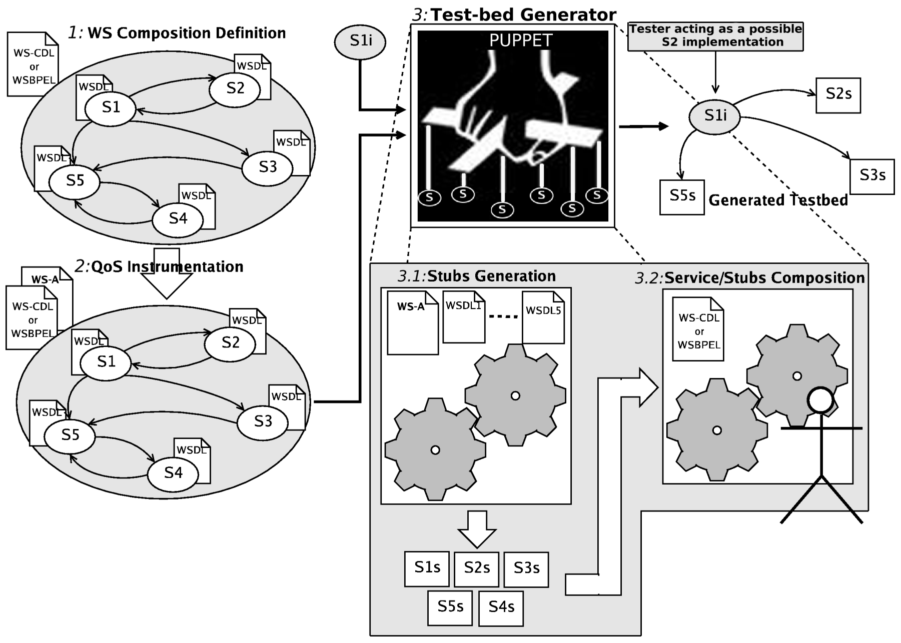


Figure 3.14: The PUPPET approach and supporting tool

choreography or in the orchestration. Even thought this phase could require the assistance of a human agent5, one of the main goal that we would like to reach is implementing a complete automatic process based on the forthcoming final WSBPEL specification.

As final result, the application of the proposed approach generates an environment for the evaluation of "S1i". The evaluation, to be carried on, will then require the availability of a tester, as also reported in Figure 3.14. This tool is beyond this specification; we can refer to the literature on the argument for possible approaches, e.g. [33]. Such a tool will have to verify that the properties specified in the QoS document (e.g. a SLAng document) are fulfilled, in addition to traditional functional testing.

In the following, a detailed description about the technologies and the tools that will be used is given. In particular, to better explain how the offline validation of QoS properties in a SOA is carried out, the description will explicitly focus on the Web Service infrastructure.

The generation process described above, exploits the information about the coordinating scenario (WSBPEL), the service description (WSDL) and the WS-Agreement document for the QoS agreement that the roles will abide to. Tools and techniques for the automatic generation of service skeletons, taking as input the WSDL descriptions, are already available and well known in the Web Services communities [3]. Nevertheless such tools only generate an empty implementation of a service and do not add any logic to the service operations.

Concretely, once a parametric mapping between specification of metric value and the executable code that will be used to characterize the services in the test harness is defined,

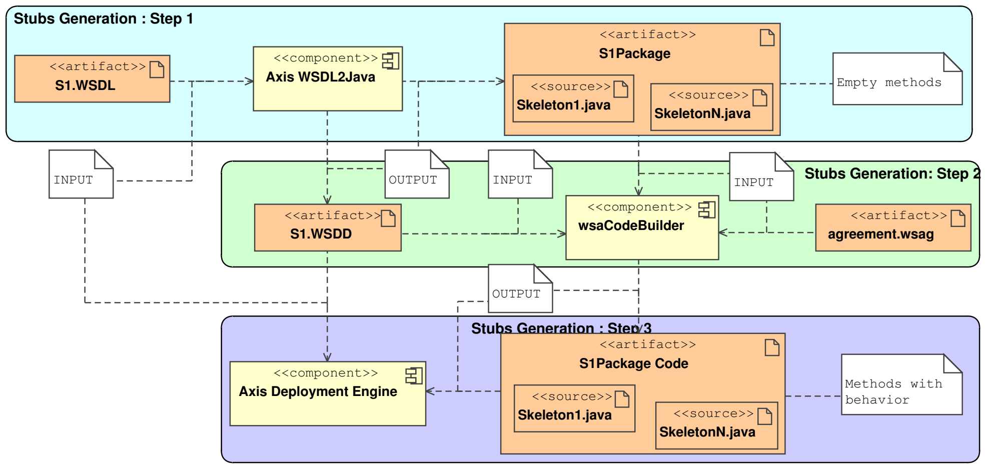


Figure 3.15: Puppet Test-bed Generator Logical Architecture

the empty implementation of a service operation are processed adding the lines of code resulting from the transformation of the service agreements specification.

Conditions on latency can be simulated introducing delay instructions into the operation bodies of the services skeletons. For each Guarantee Term in a WS-Agreement document, information concerning the service latency is defined as a Service Level Objective according to a prescribed syntax. The example in Table 3.3 reports a WS-Agreement example code for latency declaration of 10000mSec and the correspondent Java code that will be automatically generated.

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:Latency>
   <puppet:TagDelay>    10000
   </puppet:TagDelay>
   <puppet:Ditribution> normal
   </puppet:Distribution>
  </puppet:Latency>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
try{
 Random rnd = new Random();
 float val = rnd.nextFloat();
 int sleepingPeriod = Math.round(val*10000);
 Thread.sleep(sleepingPeriod);
}
catch (InterruptedException e)  {}
...
```

Table 3.3: Service Level Objective Mapping for Latency

Even though in the examples we refer to constant delays, in general it is possible to handle and generate transformation rules for more complex constraints. Indeed, by declaring the parameters that characterize a distribution in a Service Level Objective, it is possible to implement a transformation function that collects such data and instantiates the delays according to the desired distribution.

According to what described in the conceptual model, the SLA can be enforced under optional conditions describing the context. Such additional constraints are usually defined in terms of accomplishments that a service consumer as well as a service provider or the service running environment must meet: for example the latency of a service can depend on the kind of the network on which the service in deployed when the request is delivered. In these cases, the transformation function wraps the simulating behavior code-lines obtained from the Service Level Objective part with a conditional statement.

Constraints on services reliability can be declared by means of a percentage index into the Service Level Objective of a Guarantee Term. Such kind of QoS can be reproduced introducing code that simulates a service container failure.

```
...
<wsag:ServiceLevelObjective>
  <puppet:PuppetRoot>
   <puppet:Reliability>
    <puppet:TagRate>   99.50
    </puppet:TagRate>
    <puppet:Window>    2000
    </puppet:Window>
   </puppet:Reliability>
  </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
if (this.possibleFailureInWindow()){
 Random rnd = new Random();
 float val = rnd.nextFloat()*100;
 if ( val>99.50f) {
  String fCode = "Server.NoService";
  String fString="No target service to invoke!"
  org.apache.axis.AxisFault fault = new
     AxisFault(fCode,fString,"",null);
  this.incNumberOfFailure(); throw fault;  }
}
...
```

Table 3.4: Service Level Objective Mapping for Reliability

QoS specifications concerning reliability constrain the number of failures that can be seen in each of those modes within the duration of a sliding window. Table 3.4 provides an example of the transformation for reliability constraint description, assuming that the Apache Axis [3] platform is used.

```
...
<wsag:ServiceLevelObjective>
 <puppet:PuppetRoot>
  <puppet:Workload>
   <puppet:NRequest>
    20
   </puppet:NRequest>
   <puppet:WinSize>
    60000
   </puppet:WinSize>
  </puppet:Workload>
 </puppet:PuppetRoot>
</wsag:ServiceLevelObjective>
...
```

```
...
public void generateTraffic ()
 throws MalformedURLException,RemoteException{
  Random rnd = new Random();
  int sleepPeriod;
  String endpoint="http://myhost/axis/services/";
  String service="client";
  String method="planJourney";
  int winSize=60000;
  for (int i=0; i<20; i++){
     this.invokeService(endpoint,service,method);
   sleepPeriod = rnd.nextInt(winSize);
   try {
    this.sleep(sleepPeriod);
   } catch (InterruptedException e) {}
   winSize = winSize - sleepPeriod;
  }
}
...
```

Table 3.5: Service Level Objective Mapping for Workload Generator

Agreements on workload assessing can be simulated creating client skeletons for the automatic invocation of the SUE. In particular, the transformation will focus on generating client-side code that is able to guarantee that the rate at which requests can be delivered to the service, the width of a sliding time window and the maximum number of responses that should be delivered across the service interface during this period (see Tab. 3.5).

The generation process augments the stubs with a private method for the remote invocation (i.e. invokeService in Tab. 3.5) and an exported public method that triggers the emulation request stream. The transformation in Tab. 3.5 reports the code for the trigger method.

3.3.4 Within ART DECO

Concerning the testing approaches work-in-progress to be performed within ART-DECO aims at integrating the functional with non functional testing using a common platform. Specifically we select Data-flow (Section 3.3.1) and SLA Testing (Section3.3.2) as a referring method for the functional and non-functional testing approaches.

Considering the architecture underlining the ART DECO infrastructure, presented in Figure 1.1, the testing activities briefly described in Section 3.3 are focused at service level (V&V at Service Level). In particular we assumes that services exported by a participant could strongly relies on the interaction with other or third part services. In this testing activities may compromise the state of the overall system.

The solution adopted in ART DECO is to simulate the behavior of (some) of the involved services generating ad hoc testing stubs. In particular the process for stubs specification exploits the Puppet (Section 3.3.3) framework. Although this can result as imprecise, it would permit to

1. test the workflow before services are actually available, or regardless of what concrete services will be bound to the workflow. In other words, by using stubs it would be possible to adopt for off-line verification purposes the same SLA testing approach used for on-line testing.

2. reduce the cost and resource usage of testing [35].

Figure 3.16 shows our proposal for ART DECO at system testing level. It is divided into three phases: the *Test Analysis*, *Test Environment SetUp* and *Services Composition*. The process takes as inputs the BPEL process to be tested, the SLA specification, the specification of services composing the BPEL process, and possible specific variable constraints on the BPEL process.. Each phase involves actions (the rounded blocks) and artifacts (the squared blocks).

In particular during the Test Analysis first a selection of parameters to be observed during the testing phase is defined (Variable Selection). Then a data-flow diagram representing the services data interactions is developed (DataFlow Diagram Derivation). In this activity the description of BPEL Process can also be exploited.

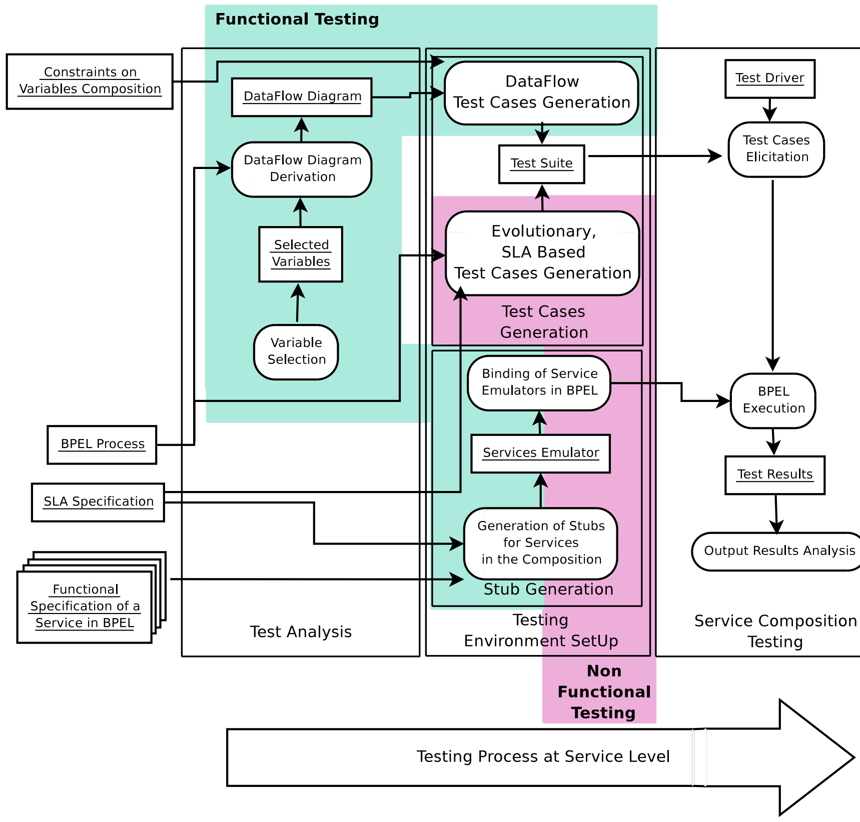

Functional Testing
Constraints on Variables Composition
DataFlow Diagram
DataFlow Diagram Derivation
Selected Variables
Variable Selection
BPEL Process
SLA Specification
Functional Specification of a Service in BPEL
Test Analysis
DataFlow Test Cases Generation
Test Suite
Evolutionary, SLA Based Test Cases Generation
Test Cases Generation
Binding of Service Emulators in BPEL
Services Emulator
Generation of Stubs for Services in the Composition
Stub Generation
Testing Environment SetUp
Test Driver
Test Cases Elicitation
BPEL Execution
Test Results
Output Results Analysis
Service Composition Testing
Non Functional Testing
Testing Process at Service Level

Figure 3.16: ART-DECO service composition testing process.

The second phase (Testing Environment SetUp) two different steps can be performed in parallel: the Test Cases Generation and the Stub Generation. In the former the selected functional and non-functional test strategies are used for deriving a test suite. In particular from one side the constants and properties expressed on the variables involved in the services composition are used together with the derived dataflow diagram, for functional test cases definition (DataFlow Test Cases Generation). From the other the BPEL Process and the SLA Specification are exploited for non-functional test case generation (Evolutionary, SLA Based Test Cases Generation). During the Stub Generation step the SLA Specification and the Functional Specification the services included in the BPEL process are used for deriving the emulator stubs for those services that can not be available for testing purposes (Generation of Stubs for Services in the Composition). Eventually those stubs are then bound together completing the implementation of the BPEL process.

During the last phase, the Service Composition Testing, the test cases are selected (Test Cases Elicitation tool) by means of a test Driver and then executed on the BPEL process(BPEL Execution). Finally test results are analyzed (Output Results Analysis).

A further application of data flow testing not included in the figure above is the use of data-flow analysis to reduce the search space for SLA testing. As highlighted by Binkley and Harman [26] data dependence analysis can be used to reduce the search space when applying evolutionary test data generation techniques. Thus will be used to (i) generate data flow test cases and (ii) to help reducing the search space for the evolutionary SLA testing.

4 On-Line Verification and Validation

Modern systems are increasingly required to be capable to evolve at run-time, in particular allowing for the dynamic plugging of new features. As an example consider the basic paradigm of service-oriented architectures where the supporting technologies tends to be more flexible and dynamic. The single centralized repository is being substituted by dedicated repositories that cooperate and exchange information about stored services on demand.

This dynamism in the composition of the architectures require that the design-time verification is supported also by a complementary mechanism to allow for the analysis of the evolving system at run-time. Hence, in this section, we analyze two different run-time approaches, monitoring and SLA online testing, and we show how they can be used for online verifying ART DECO architecture and application.

4.1 Software Systems Run-time Monitoring

In general terms monitoring is the activity of observing and checking that a system is running according to some specified functional and non-functional requirements. It is a quite delicate and expensive activity that asks for careful planning. It is necessary to develop strategies that in a particular context maximize chances of discovering faults still not excessively burdening software performance. Indeed monitoring is somehow an ambiguous word. Many different approaches related to run-time checking use this word even if they provide support for different activities. For the sake of clarity we provide below a classification of the different activities that the engineer should consider when setting a monitoring strategy. For a insightful discussion on monitoring and related taxonomy refer to [49].

- *Selection of the run-time verification language*: the first step to carry on planning run-time monitoring of a system is to identify which kind of verification is relevant in the particular context. In the simplest case we could define simple properties on the value assumed by a logical variable. Instead more complex scenarios could define properties on messages exchanged by parts of the system and on their relative order. In synthesis, this step requires to define/select a language to describe expected properties on interesting system characteristics.

- *Identification of the relevant information to be observed at run-time*: this phase concerns the identification of the characteristics that must be observed at run-time to put in place the strategy defined in the previous step. For instance in case of simple properties on value assumed by a logical variable, it will be necessary to identify which are the real variables influencing the value of the logical variable.

- *Definition and/or identification of mechanisms for run-time data collection*: in this phase the engineer should define and put in place suitable mechanisms to observe the information identified in the previous step. For instance, with reference to the variable value example, the engineer must identify how and when the value of the real variable must be observed. In general two different approaches can be adopted to retrieve run-time information. The first foresee the instrumentation of the system to monitor with additional code. The second relies instead on the availability of suitable mechanisms provided by the platform. In the latter case, code instrumentation may be still necessary if required information are not made available by the platform.

- *Definition of mechanisms for run-time checking (Analyser Engine)*: after run-time information are retrieved, it is necessary to introduce a system that it is able to distinguish correct behaviour from incorrect one. The complexity of such a system is strictly related to the selected strategy. For instance, in case of a variable the condition to be verified can specify a simple threshold value or instead could specify a possible admitted history. More complex scenarios can also be imagined.

- *Definition of recovery strategies*: it is necessary to foresee strategies to bring the system back to a correct state or to gracefully stop it, after that an erroneous condition has been identified.

- *Definition of recovery mechanisms*: recovery strategies can succeed only if suitable mechanisms are available. It is then necessary to have some kind of control on the system behaviour and possibly force it in order to recover from the error. As usual, this mechanisms can be programmed by the implementor or to a certain extent provided by the platform.

Developing an approach to run-time verification requires to engage in the different phases discussed in the list. Indeed, many approaches propose solutions only for the first four points without addressing the recovery part (which will be addressed in future work).

In the following, we present two different monitoring approaches, namely Mosaico and WSCoL (and its monitoring engine). Both the approaches cover the group of first four activities, and, moreover, WSCoL also proposes a recovery mechanism.

4.1.1 Monitoring of Architectural Properties

The MOSAICO approach consists in monitoring the run-time execution of a dynamically evolving CBS in order to analyze its perpetual compliance to selected properties. This section provides a first glimpse on the approach. Details are provided in [20].

1. **Definition of the SA**: in our approach we assume the availability of such specification and base on it all the following steps. In our hypothesis the architecture describes the relations among a set of components belonging to the kernel. At the same time, it specifies how it is possible to dynamically extend the system at run-time.

2. **Definition of Relevant Architectural Properties**: in this phase the engineer defines which are the architectural properties that a real implementation of the system must satisfy. Some of the properties could be verified statically on the SA definition, for instance by a model-checker. Nevertheless, the presence of black box components should generally suggest to complement static verification with run-time techniques. A simple example could be a certain communication pattern among components that must hold at run-time.

3. **Instrumentation and Monitoring**: this step requires to put in place mechanisms to monitor the flow of messages among the components. In general the term "monitoring" refers to watching a system while it is running. This comprehends various activities, as detailed below, and might become a quite critical and expensive process [1].

   First of all, the events to be observed at run-time so to check the defined properties must be identified. Then, the system needs to be instrumented accordingly and monitored. In the MOSAICO approach the instrumentation is carried on using Aspect Oriented Programming. Nevertheless, other approaches are possible based for instance on the use of mechanisms provided by the platform. It is worth noting that the presence of concurrent processes could make the observation of message order tricky [19].

4. **Definition of the Analyser Engine**: this is the mechanism that reveals if a property has been fulfilled or not. All the information collected must be reported to this engine. In general not all the observed events are relevant for verification purpose. Clearly, if the analyser detects a violation, it should report it to some recovery system that can bring the system back to a correct state or gracefully stop it. This final step is certainly important but is outside the scope of the present paper.

Through the described steps and the artifacts correspondingly derived, the MOSAICO approach perpetually re-iterating steps three and four permits to continuously check the compliance of the system to the properties. In particular, whenever the architecture evolves as consequence of the insertion/removal of components, the approach permits to immediately highlight violated properties.

The MOSAICO process is shown in Figure 4.1 (rounded boxes represent algorithms and applications, irregular boxes represent artifacts, colored boxes represent algorithms or artifacts developed by us).

[1]For a comprehensive survey on monitoring and related taxonomy we refer to [49].

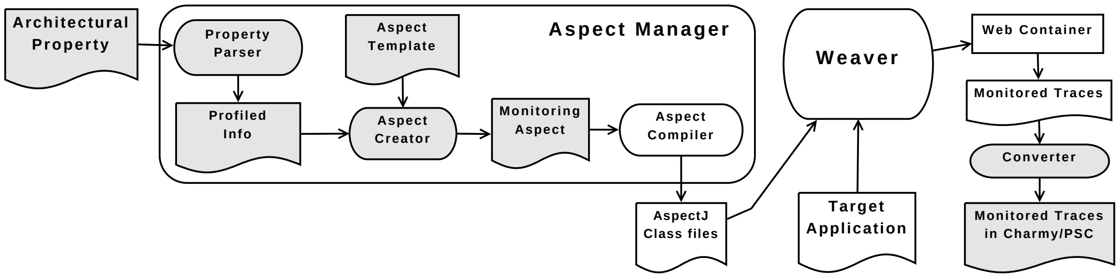


Figure 4.1: The MOSAICO AOP Approach

Further Readings

More detailed information on MOSAICO can be found in Reference [20].

4.1.2 Monitoring of BPEL Compositions

To validate running compositions of services, we propose Dynamo (Dynamic Monitoring, [12]), that provides suitable probes to oversee the execution of deployed compositions. Dynamo oversees the behavior of participating services in BPEL-like compositions.

Monitors and probes [49] are the “standard” solution for assessing the quality of applications at runtime: Dynamo borrows these concepts and stresses the idea that Web service compositions require data that come from very different sources and probes must be able to accommodate all of them. Functional and non-functional guarantees are defined in terms of pre- and post-conditions associated with the invocations of external services. Monitoring directives, called monitoring rules, comprise three parts: a monitoring location indicates where in the BPEL process the rule must be evaluated, a priority defines the level of importance associated with the rule, and a monitoring expression states the constraint on execution data. Monitoring rules are blended with the BPEL process at deployment-time. The use of external monitoring rules allows us to keep a good separation between business and control logics and also to associate different rules with the same process.

Monitoring expressions are specified in WSCoL (Web Service Constraint Language), which is a special-purpose assertion language (like JML [34]) augmented with constructs to gather data from external sources and allow the user to probe “and thus monitor” any data that can be collected while the process executes. WSCoL provides language-specific constructs for data collection and data analysis. Data collection is responsible for obtaining –either directly or through computation– the monitoring data used to check whether partner services match defined monitoring rules. WSCoL distinguishes among three kinds of monitoring data: internal variables, data that belong to the state of the running process, external variables, data obtained externally, and historical variables, i.e. monitoring data obtained from previous process executions.

Data analysis checks whether collected data comply with set requirements. WSCoL supports the typical boolean operators, typical relational operators and typical mathematical operators. The language also supports predicates on sets of values through the use of universal and existential quantifiers, and other constructs, such as max, min, avg, sum, and product.

The monitoring on the BPEL composition is performed by the Monitoring Manager, that is in charge of interpreting monitoring rules, storing the configuration with which users want to run their processes, interacting with external data collectors, and sending the data to invoke the partner services to the Service Execution Bus. A monitored BPEL process calls the Monitoring Manager (through the bus), instead of the actual partner service, whenever there is a monitoring rule selected for the invocation. The decision on whether the rule must be considered and then its evaluation is in charge of its components. The Monitoring Manager components are:

- The **Rules Manager** is responsible for managing the internal flow of the activities that must be performed.
- The **Configuration Manager** keeps track of the initial process configuration, selected monitoring rules, and all the information needed for interacting with the external services (i.e., services being monitored and external data collectors) for each active instance.
- The **Invoker** invokes the partner Web service.

When the Rules Manager receives the results of the service invocation, it interacts with the Configuration Manager to retrieve the post-condition (monitoring rules) associated with the invocation. The Rules Manager contacts the Invoker to retrieve the historical data from the external Store; it would use the Invoker also to obtain external data from Data Collectors, if needed. Once all the data are available, the Rules Manager begins its interaction with the Analyzer (through the Invoker). The Monitoring Manager uses an external Analyzer to let the user customize analysis capabilities by plugging dedicated analyzers. If the Analyzer responds with an error, i.e., the condition is not satisfied, the Rules Manager communicates it to the BPEL process "through the bus" by returning a standard fault message, as published in theWSDL description of the Monitoring Manager. If the post-condition is satisfied, the Monitoring Manager returns the original service response to the BPEL process.

4.2 Search-based Testing of Service Level Agreements

The approach described in this section focuses on the testing of Service Level Agreements (SLAs). A SLA is negotiated between a service provider and a service consumer (i.e., an integrator, or an end-user), and guarantees to the service consumer a given QoS level, sometimes depending on how much she/he is willing to pay for the service usage. In other words, a SLA constitutes a form of contract between service providers and consumers, and its violation would cause lack of satisfaction for the consumer and lost of money for the provider. For this reason, before offering a SLA, a service provider would limit the possibility that it can be violated during service usage. This document explores the use of Genetic Algorithms (GAs) to generate test data causing SLA violations. For a service-oriented system such violations can be due to the combination of different factors, i.e., (i) inputs, (ii) bindings between abstract and concrete services, and (iii) network configuration and server load. In the proposed approach, GAs generate combinations of inputs and bindings for the service-oriented system causing SLA violations. The proposed fitness combines a distance-based fitness that awards solutions close to QoS constraint violation, with a fitness inspired from what proposed by Wegener *et al.* [117] guiding the coverage of target statement by means of a proximity measure and of conditional distance. The approach has been applied to an audio processing workflow and to a service for generating charts, and in both cases it was capable to generate testing data causing SLA violations.

4.2.1 Approach

As mentioned in the introduction, this approach deals with the generation of test data for a service-oriented system[2] causing SLA violations. Let us consider the workflow in Figure 4.2 representing an image processing composite service. The service takes as input an image in a specific format, characterized by its horizontal and vertical dimension ($dim1$ and $dim2$), a Boolean value indicating whether the image needs to be posterized, and the sharpening level ($nsharpen$). According to the input options, the composite service performs some filtering operations on the image by invoking external services. For each filter (*Scale*, *Posterize*, *Sharpen*, and *Gray*), hereby referred as *abstract services*, some semantically equivalent *concrete services* are available, each one, however, ensuring different QoS (response time and resolution of the output image). Bindings are chosen using optimization approaches, such as those proposed by Zeng *et al.* [125] or by Canfora *et al.* [36], that determine the (near) optimal set of bindings that ensure a QoS constraint satisfaction and that optimize a given objective function.

At execution time, there may exist combinations of bindings and workflow inputs that cause SLA violations, i.e., violations of QoS constraints. Let us consider, for example, that the service provider guarantees to the service consumer a response time less than 30 ms and a resolution greater or equal to 300 dpi. For the workflow in Figure 4.2, for example, let us consider as inputs $posterize = true$, an image having a size smaller than 20 Mb (which constitutes a precondition for our SLA), $dim1 = dim2$ and $nsharpen = 2$. Let us also consider that the abstract services are bound to *ScaleC*, *PosterizeC*, *SharpenB* and *GrayA*. In this case, while the response time will be lower-bounded by 24 ms and therefore the constraint would be met, the resolution of the image produced by the composite service would be of 200 dpi, corresponding to the minimum resolution guaranteed by the invoked services. In other cases the scenario can be much more complex, in particular when combinations of inputs for each service invoked in the workflow can, on its own, contribute to the overall SLA violation.

In summary, the test data generation approach must be able to produce inputs and bindings that cause SLA violations. Clearly, violations of some QoS attributes, e.g.,

[2]That can be on its own a service, mentioned as composite service and described with a workflow.

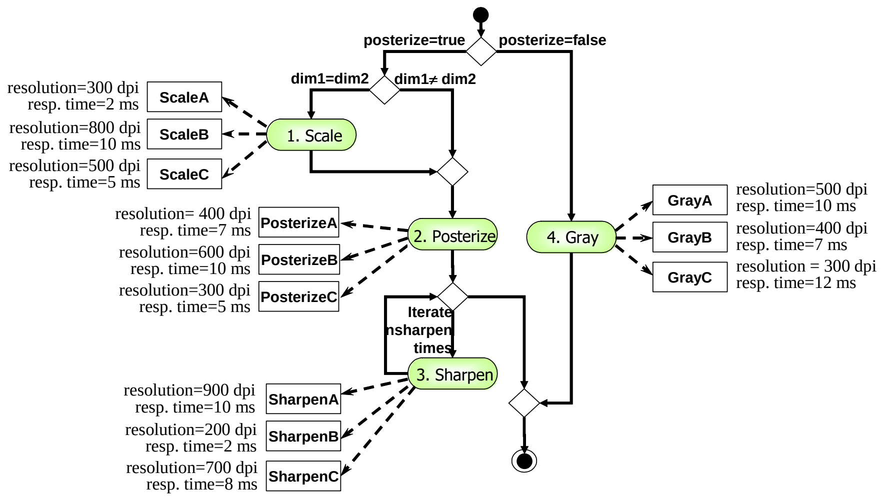

posterize=true
posterize=false
dim1=dim2
dim1≠ dim2
resolution=300 dpi resp. time=2 ms
ScaleA
resolution=800 dpi resp. time=10 ms
ScaleB
resolution=500 dpi resp. time=5 ms
ScaleC
1. Scale
resolution= 400 dpi resp. time=7 ms
PosterizeA
resolution=600 dpi resp. time=10 ms
PosterizeB
resolution=300 dpi resp. time=5 ms
PosterizeC
2. Posterize
4. Gray
GrayA
resolution=500 dpi resp. time=10 ms
GrayB
resolution=400 dpi resp. time=7 ms
GrayC
resolution = 300 dpi resp. time=12 ms
Iterate nsharpen times
3. Sharpen
resolution=900 dpi resp. time=10 ms
SharpenA
resolution=200 dpi resp. time=2 ms
SharpenB
resolution=700 dpi resp. time=8 ms
SharpenC

Figure 4.2: Example of composite service

response time or throughput, can also depend on the network and server load. Dealing with such a factor is, however, out of scope of this document and will be considered as part of our future work.

White Box Approach

Service composition white box testing can be pursued by integrators that, before offering a SLA, want to ensure that the composition is able to meet a given QoS level. They have the composition source code available, written using WS-BPEL or any traditional programming language. The test data generation process is composed of two steps, detailed in the following two subsections.

The first step aims to identify which workflow paths are likely to exhibit high values for upper-bounded QoS attributes (e.g., response time) and low values for lower-bounded QoS attributes (e.g., resolution). This step is performed by considering (i) QoS value estimates for services composing the workflow, and (ii) estimated upper-bound number of executions for each loop, as declared by the service provider. In other words, the overall QoS is estimated according to aggregation formulae defined by Cardoso [38] and then used for binding purposes by Canfora *et al.* [36]. To determine the critical paths (not to be confounded with the critical nodes for the coverage criteria, see below) for a particular QoS attribute, concrete services having the highest (or the lowest for lower-bounded attributes) value are considered. Since loops are considered to be executed a fixed number of times, the critical paths can be identified by using a linear search, without the need for using any particular heuristic.

Once a QoS-critical path has been identified, we use GAs to generate test cases that i) cover the path and ii) violate the SLA. Because expensive paths have been identified according to QoS estimates, while the GA fitness for test data generation relies on mea-

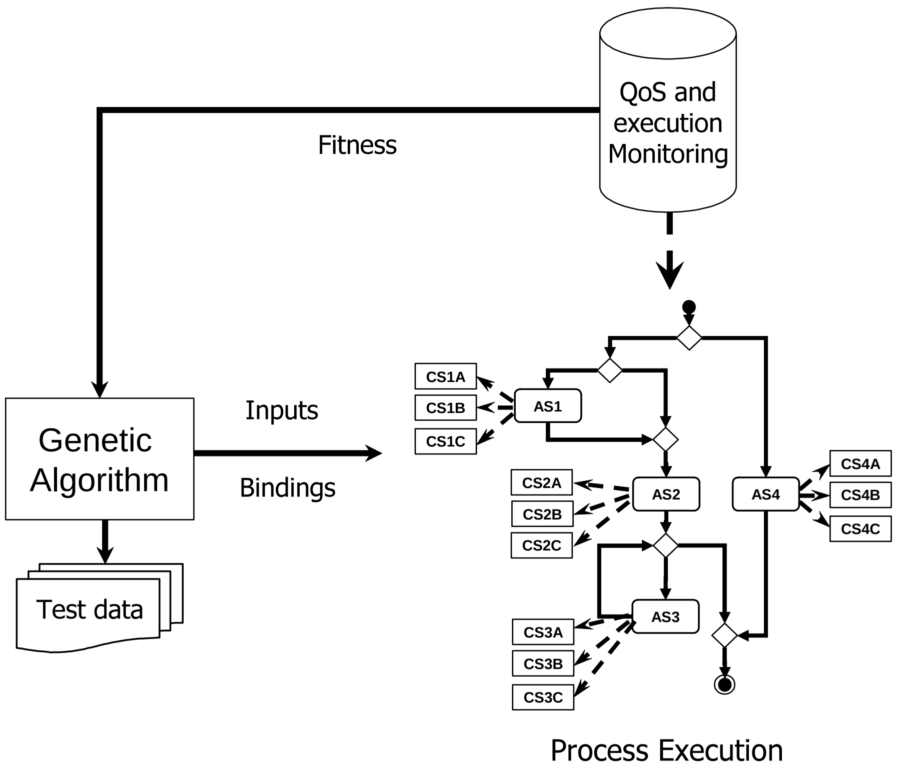


Figure 4.3: Evolutionary SLA testing approach

sured QoS values, there is no guarantee that test cases violating the SLA also follow the critical paths. Nevertheless, such paths constitute a "starting point" to search for SLA violations. The test data generation process is represented in Figure 4.3. The GA generates new individuals, intended as workflow inputs plus bindings between abstract and concrete services. The bindings are enacted on the workflow by replacing abstract end-points with concrete ones, and then the workflow is executed with the generated inputs. During the execution, the service QoS is observed through monitoring mechanisms and, together with workflow coverage information, is fed back to the GA to permit the individual's fitness evaluation.

The genome representation is composed of two data structures, as shown in Figure 4.4:

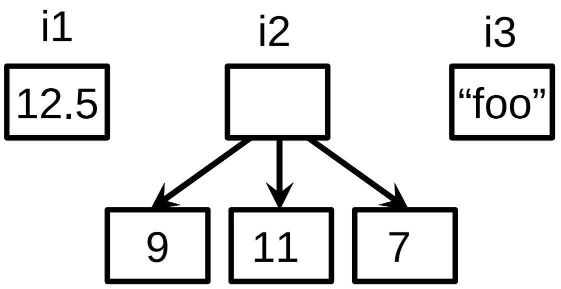
1. a forest, where each tree represents a composition input, encoded according to the XML schema defining its type, in case it is not a primitive value (integer, float, Boolean, String). Figure 4.4-b shows a representation for inputs defined according to the WSDL excerpt of Figure 4.4-a;

2. an array, containing a slot for each abstract service in the workflow.

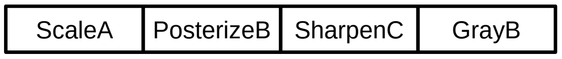
The mutation operator (Figure 4.5-a) randomly decides if mutating the inputs, the

```
  <complexType name="ArrayOf_xsd_int">
    <complexContent>
      <restriction base="soapenc:Array">
        <attribute ref="soapenc:arrayType“
              wsdl:arrayType="xsd:int[]" />
      </restriction>
    </complexContent>
  </complexType>
...
  <wsdl:message name="ServiceOperationRequest">
    <wsdl:part name="i1" type="xsd:float" />
    <wsdl:part name="i2" type="impl:ArrayOf_xsd_int" />
    <wsdl:part name="i3" type="xsd:string" />
  </wsdl:message>
```

(a) Excerpt of composite service WSDL



(b) Input encoding



(c) Bindings encoding

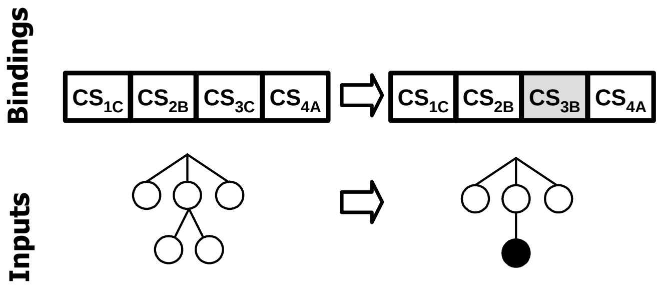
Figure 4.4: Genome representation

bindings or both. Bindings mutation is quite simple: the endpoint of a randomly selected service is changed to one of the available concrete services corresponding to such an abstract service. Regarding inputs, the operator randomly mutates a (sub)tree of one of the inputs.

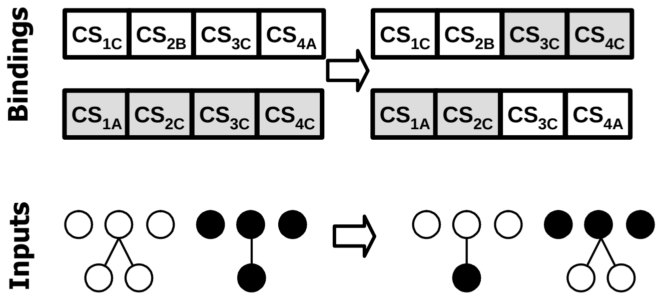
- *Sequences* are handled by generating random elements of each type defined in the sequence itself;
- *Choices* hare handled by randomly generating one of the elements it defines;
- *Occurrence indicators* are handled by generating a random number of elements between *minoccurs* and *maxoccurs*;
- *Leaves*, i.e., primitive values, are mutated as follows. *Integer* and *floats* are replaced by random values in a range specified by the tester before starting the testing data

generation. *Booleans* are mutated between true and false. For *strings* it is either possible to randomly generate a random string bound by a maximum length, or to randomly pick one or more items from a user-defined list (e.g., a dictionary). The former can be used when the service accepts as input any string, while the latter is more useful when the sequence of legal inputs is limited.

It is worth to note that the generated inputs must be, on their own, in agreement with the SLA. For instance, if the service provider guarantees that the service is able to apply a filter on an image of up to 2 Mbytes in 10 s, a larger input should not be considered as part of our testing range, unless one wants to perform robustness testing.



(a) Mutation



(b) Crossover

Figure 4.5: GA operators

The crossover (Figure 4.5-b) is also randomly applied to inputs, bindings or both. For bindings, we used the standard one-point crossover. For inputs, a random sub-tree was randomly selected, in the same position, on the two parents and then swapped producing the offspring. The selection is made through a roulette wheel selection operator. The type of GA adopted is a simple GA with elitism.

The fitness function accounts for different factors. To let the GA generate testing data that causes SLA violations, the fitness must be a function of how far an individual is from QoS constraint violation. If considering these constraints written in the form:

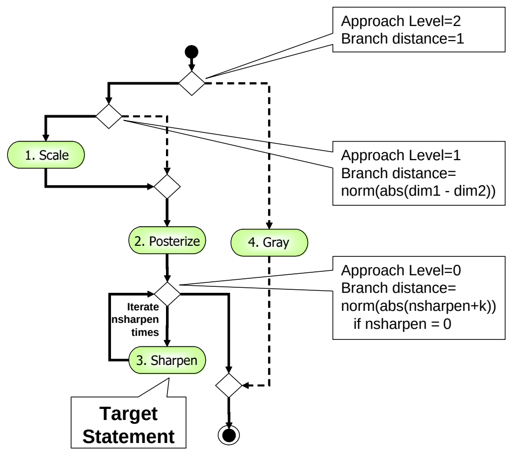


Figure 4.6: Computing the fitness defined by Wegener et al.

$$cl_i \leq th_i, \ i = 1, \ldots, n \tag{4.1}$$

where cl_i is the measured value of the $i-th$ QoS attribute for a GA individual and th_i its upper-bound. The difference between the constraint upper-bound and the actual value is expressed as:

$$D_i = th_i - cl_i \tag{4.2}$$

Equation (4.2) holds for upper-bounded QoS attributes (e.g., response time). For lower-bounded attributes (e.g., accuracy) the distance corresponds to equation (4.2) right-hand-side multiplied by -1. One could just consider such a distance as a fitness, however in many cases the QoS depends on the particular path followed in the workflow and on the particular set of services invoked. Since equation (4.2) gives no information about the path covered, it might be unable to drive the evolution towards constraint violation. Let us consider a path p estimated to be a QoS-critical path, and let us consider that, to cover such a path, statements $s_1, \cdots, s_n$ must be traversed. For statement s_j, we can consider a proximity function composed of an approach distance and a branch distance for critical nodes, i.e., for nodes pathing away from the target statement [117]:

$$P_j = dependent - executed + m\_branch\_distance \tag{4.3}$$

where *dependent* indicates the number of critical nodes the target statement depends on, *executed* indicates the number of these critical nodes that have been executed, and *m_ branch_ distance* indicates the distance, normalized in the interval [0,1] from satisfying the Boolean condition of the critical nodes pathing away from the target statement. If we consider the example of Figure 4.2, and we consider that, for instance, covering the QoS-critical path requires the execution of nodes 1,2,3 (1 time), Figure 4.6 shows approach level values for critical statements, as well as the branch distance. The branch distance is 1 if the first conditional (*posterize*) is not satistied, 0 if it is satisfied. For the second conditional *dim1-dim2*, it measures the absolute, normalized distance between the two values (if zero, then the condition is satisfied), while for the third conditional it corresponds to the normalized, absolute value of *nsharpen* (plus a constant) if $nsharpen \leq 0$, while it would be 0 if $nsharpen > 0$. It is important to note that the approach distance works well for flag-free CFGs; in presence of flags proper transformations are necessary [69]. By combining equation (4.3) with equation (4.2), we obtain the following fitness function for the $j-th$ path and the $i-th$ QoS attribute:

$$F_{i,j} = \frac{total\_gen - current\_gen}{total\_gen} \cdot P_j + \frac{current\_gen}{total\_gen} \cdot D_i \tag{4.4}$$

where *current_ gen* is the generation when the fitness is evaluated and *total_ gen* is the total number of generations fixed for the GA. The two fitness factors are dynamically weighted. First, the fitness gives more weight to the QoS-critical path coverage. Then, it tends to award constraint violation, assuming that it can be pursued once a QoS-critical path has been covered.

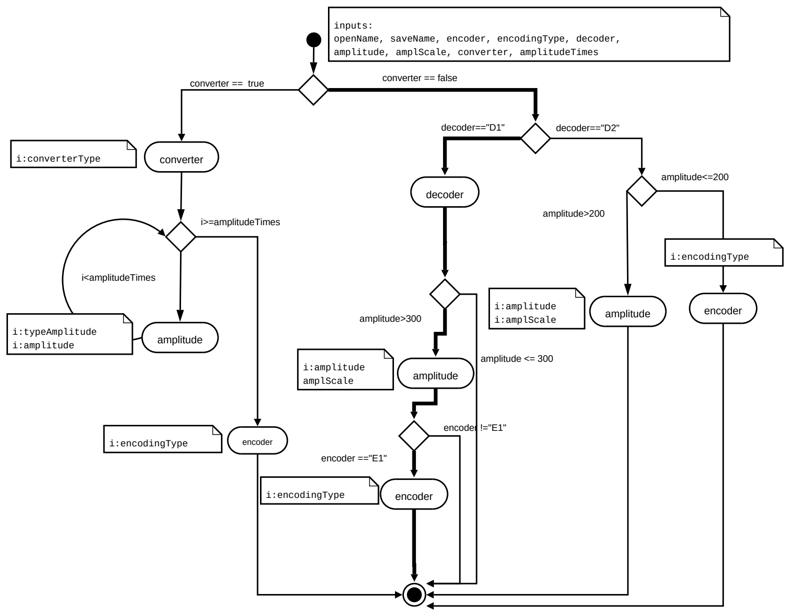


Figure 4.7: Case study: Audio processing workflow

Black Box Approach

The "white box approach" has the advantage of using information about coverage of QoS-critical paths to guide the search towards SLA violation. However, it can only be used by composite service developers or providers having the workflow source code available. When such a condition does not hold — and this may be the case of other integrators or third-party certifiers that want to test the composition before using it — test data can be evolved by only considering the QoS constraint distance, i.e., equation (4.2), as a fitness function for the QoS attribute of interest. Also, it can happen that service consumers have no control on the bindings, thus they can only test a service through its inputs.

Towards considering dependence from the context

Work in progress is devoted to take into account the dependency of QoS from factors related to the service context, i.e.:

1. Server load;
2. Network load; and
3. Number of concurrent requests.

To this aim, we intend to modify our fitness function with finer-grained models for time-dependent QoS attributes, such as response time or throughput. In such models, the attributes (for example the response time) are estimated as a function of the above mentioned factors and, in particular, a queuing model is used to estimate the response time in terms of the number of concurrent requests [66, 126].

The generated test cases will provide a measure of how well a web service can serve the request in a particular scenario of network and/or the server load. In this case, the generated test cases can be useful to determine which QoS level the service provider can guarantee, and thus support the creation of SLA proposals.

4.3 Within ART DECO

The online verification mechanisms, we proposed in the previous sections, can both be exploited, in a complementary way, in ART DECO project to verify online (at different levels) the correctness of the developed architecture and application.

In detail, the two monitoring approaches offer online verification of functional properties both at architectural (Mosaico) and service level (WSCoL), hence, they cover two different stages of the architecture proposed at Figure 1.1. Moreover WSCOL and its engine allows not only to monitor properties, but also to introduce recovery actions in the case they are violated. //The goal for ART DECO is to integrate the two approaches in the proposed application, extending them in order to monitor also non-functional properties.

Since in ART DECO context, many actors (participants) are composed at runtime and they interact among them, without knowing the implementation details of others, many of non-functional properties deal with QoS and response time. Hence, a further goal of monitoring is to extends Mosaico and/or WSCoL in order to deal with time, at least at service level. The approach we want to use takes inspiration from [11], where WSCoL is enriched with temporal predicates and functions and the WSCoL engine is modified in order to deal asynchronously with future temporal predicates. Starting from there and Mosaico, we want to propose a unique platform for monitoring the architecture at different levels.

Instead, the approach is focused on an higher level, i.e., Service Level Agreements (SLAs). A SLA is negotiated between a service provider and a service consumer (i.e., an integrator, or an end-user), and guarantees to the service consumer a given QoS level. and it constitutes a form of contract between service providers and consumers. The proposed approach exploits Genetic Algorithms (GAs) to generate test data causing SLA violations.

5 Conclusion

Verification, Testing and Quality of Services evaluation are important activities of the software development, which can be aimed at different objectives, such as verify that the software system meets the user's real needs and validating that it meets the requirements specifications.

The meaning of "meets" can be expressed with different degrees of formalization, and can be checked with different methods. In particular in this document, due to the complexity of the ART DECO infrastructure, we focused on several methodologies: Model Checking, Quality of Services assessment, Testing and Monitoring.

For each of the above mentioned topics, this deliverable reported a detailed study of the relevant techniques applicable within the ART DECO context to assure a complete and coherent Verification and Validation process applicable at different levels: Service, Architectural, and Physical. The proposed techniques have been selected from the state-of-the art and some initial investigations on their applicability to specific project needs have been provided.

The selection of the applicable V&V techniques represents the starting point for designing the prototype framework of a support environment for the off-line and on-line V&V as required for the P.A. 10.1. In general terms, the approach followed during the writing of this document was to provide both modeling facilities and a set of techniques useful both for the off-line and on-line V&V.

Overall this deliverable has provided the following main results:

- Identifying an ART DECO modeling formalism useful for V&V purposes. To exploit the DUALLy performance, two complementary formalisms have been combined for facing different ART DECO modeling exigences: ArchiTRIOand KLAPER.
- Identifying the set of techniques that can be applied within ART DECO for off-line V&V. We analyzed three different research approaches (model checking, QoS assessment, Testing) highlighting their applicability within ART DECO and the advantages that each methodology could produce.
- Identifying the set of techniques that can be applied within ART DECO for on-line V&V. We analyzed two different run-time approaches covering the system run-time monitoring and the search-based testing of Service Level Agreements.

Wherever possible the proposed approaches have been applied to running examples taken from the ART DECO project. As a future work, it is our intention to apply what proposed in this deliverable to real case studies provided by other project Units. From one side, this will confirm our preliminary selection, and from the other it will help to identify new research challenges and consequently directions to improve the proposed approaches.

In conclusion, this delivery will serve as a guideline for the definition of a preliminary prototype framework useful for V&V purposes and a valid basis for the final definition of the set of methodologies applicable inside the ART DECO project for verification, testing and QoS evaluation.

Bibliography

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[2] M. Ajmone Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In G. Rozenberg, editor, *Advances in Petri Nets 1987*, volume 266 of *LNCS*, pages 132–145. Springer-Verlag, 1987.

[3] Apache Software Foundation. *Axis User's Guide.* http://ws.apache.org/axis/java/user-guide.html.

[4] M. Autili, P. Inverardi, and P. Pelliccione. Graphical Scenarios for Specifying Temporal Properties: an Automated Approach. *Automated Software Engineering*, 2007.

[5] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06)*, Shanghai, China, May 27, 2006. ACM Press.

[6] A.Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the Fourth Software Quality Conference*, pages 300–309, 1995.

[7] H. S. B. Jones and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11:299–306, 1996.

[8] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. Wsdl-based automatic test case generation for web services testing. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pages 215–220, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[9] G. Balbo. Introduction to stochastic petri nets. In J.-P. Katoen, H. Brinksma, and H. Hermanns, editors, *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 84–155. Springer-Verlag, 2001.

[10] L. Baresi, P. Beretta, R. Fraccapani, C. Ghezzi, and F. Pacifici. Towards a model-driven approach to develop applications based on physical active objects. In *Proceedings of the Asia Pacific Conference on Software Engineering (APSEC)*, pages 173–182, 2006.

[11] L. Baresi, D. Bianculli, C. Ghezzi, sam Guinea, and P. Spoletini. A timed extension to wscol. In *Proceedings of ICWS - Industrial Track*, 2007.

[12] L. Baresi and S. Guinea. Towards dynamic monitoring of bpel processes. In *Proc. of the 3rd International Conference on Service-Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282, 2005.

[13] L. Baresi and F. Pacifici. Modello di riferimento per la sensoristica. Technical Report R.2.1, EasyLog, E-Adaptive Services for Logistics, FAR Project, 2007.

[14] B. Beizer. *Software Testing Techniques 2nd edition.* International Thomson Computer Press, 1990.

[15] A. Bertolino, A. Bonivento, G. D. Angelis, and A. Sangiovanni Vincentelli. Modeling and early performance estimation for network processor applications. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2006)*, volume LNCS 4199, Genova, Italy, 2006. Springer-Verlag.

[16] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in LNCS, pages 1–25. Springer-Verlag, 2006.

[17] A. Bertolino and E. Marchetti. *Software Testing*, volume SWEBOK Guide to the Software Engineering Body of Knowledge, chapter 5. IEEE Computer Society, Version 2004. http://www.swebok.org.

[18] A. Bertolino and R. Mirandola. Software Performance Engineering of Component–Based Systems. In *Proc. of the 4th International Workshop on Software and Performance (WOSP 2004)*, pages 238–242. ACM Press, 2004.

[19] A. Bertolino, H. Muccini, and A. Polini. Architectural verification of black-box component-based systems. In *Proc. Int. Workshop on Rapid Integration of Software Engineering techniques (RISE2006)*, volume to appear as LNCS, September 13-15 2006.

[20] A. Bertolino, H. Muccini, A. Polini1, and F. Ricci. Run-time Analysis of Dynamically Evolving Black-box Component-Based Systems. In *In Proc. the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2007*, 2007.

[21] A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In *31st EUROMICRO International Conference on Software Engineering and Advanced Applications*, pages 134–142, 2005.

[22] J. Bézivin. On the Unification Power of Models. *J. Software and Systems Modeling*, 4(2):171–188, 2005.

[23] D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify BPEL4WS workflows. In *Proceedings of the 2007 IEEE International Conference on Service-Oriented Computing and Applications (IEEE SOCA 2007)*, Newport Beach, California, June 2007. To appear.

[24] D. Bianculli, A. Morzenti, M. Pradella, P. S. Pietro, and P. Spoletini. Trio2promela: a model checker for temporal metric specifications. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Research Demo*, Minneapolis, US, May 2007. To appear.

[25] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999.

[26] D. Binkley and M. Harman. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Trans. Software Eng.*, 30(11):715–735, 2004.

[27] A. Bobbio, A. Puliafito, M. Telek, and K. S. Trivedi. Recent developments in non-Markovian stochastic Petri nets. *Journal of Circuits, Systems and Computers*, 8(1):119–158, 1998.

[28] A. Bobbio and M. Telek. Non-exponential stochastic Petri nets: an overview of methods and techniques. *Computer Systems Science and Engineering*, 13(6):339–351, 1998.

[29] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Sandrini. DEEM: a tool for the dependability modeling and evaluation of multiple phased systems. In *DSN2000 Int. Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, pages 231–236. IEEE Computer Society, June 25–28 2000.

[30] A. Bondavalli, I. Mura, and K. S. Trivedi. Dependability modelling and sensitivity analysis of scheduled maintenance systems. In *EDCC-3 European Dependable Computing Conference*, pages 7–23, Prague, Czech Republic, September 1999. (also Lecture Notes in Computer Science N. 1667), Springer Verlag.

[31] E. Börger. High Level System Design and Analysis using Abstract State Machines. In *FM-Trends 98, Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*, pages 1–43. Springer, 1999.

[32] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028, 2005.

[33] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[34] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[35] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[36] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *Proc. of the Genetic and Computation Conference (GECCO'05)*, pages 1069–1075, Washington, USA, June 2005. ACM.

[37] H. Cao, S. Ying, and D. Du. Towards model-based verification of bpel with model checking. In *Sixth International Conference on Computer and Information Technology (CIT 2006), 20-22 September 2006, Seoul, Korea*, pages 190–194, 2006.

[38] J. Cardoso. *Quality of Service and Semantic Composition of Workflows.* PhD thesis, Univ. of Georgia, 2002.

[39] Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[40] H. Choi, V. G. Kulkarni, and K. S. Trivedi. Performance modeling using Markov regenerative stochastic Petri nets. *Performance Evaluation*, 20(1–3):339–356, 1994.

[41] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM Transaction on Software Engineering and Methodology*, 8(1):79–113, 1999.

[42] G. Ciardo, A. Blakemore, P. Chimento, J. Muppala, and K. Trivedi. Automated generation and analysis of markov reward models using stochastic reward nets. In C. Meyer and R. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models, IMA Volumes in Mathematics and its Applications, volume 48*, pages 145–191. Springer-Verlag, 1993.

[43] G. Ciardo, R. German, and C. Lindemann. A characterization of the stochastic process underlying a stochastic petri net. *IEEE Transactions on Software Engineering*, 20(7):506–515, 1994.

[44] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Mobius modeling tool. In *9th Int. Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001. IEEE Computer Society Press.

[45] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[46] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from uml diagrams. In *Proc. of* 2^{nd} *IEEE Workshop on Software and Performance (WOSP2000)*, September 2000.

[47] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 302–309, New York, NY, USA, 2002. ACM Press.

[48] D. Di Ruscio and A. Pierantonio. Model Transformations in the Development of Data–Intensive Web Applications. In *CAISE '05*, volume 3520 of *LNCS*, pages 475–490. Springer, 2005.

[49] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, 2004.

[50] E. Di Nitto and L. B. (editors). *Test and Analysis of Web Services.* Springer, 2007.

[51] M. Di Penta, G. Canfora, M. Bruno, G. Esposito, and V. Mazza. Search based testing of Service Level Agreements. In *in proceedings of the Genetic and Computation Conference (GECCO 2007), to appear.* ACM PRESS, July 2007.

[52] D. Di Ruscio, P. Inverardi, H. Muccini, P. Pelliccione, and A. Pierantonio. DUALLY: an extensible UML-based Architecture Description Language. Technical report, Unpublished, University of L'Aquila, 2007.

[53] D. Di Ruscio, H. Muccini, P. Pelliccione, and A. Pierantonio. Towards Weaving Software Architecture Models. In *Proc. MBD/MOMPES Workhop within the ECBS 2006, IEEE Computer Society Press*, 2006.

[54] J. B. Dugan. Software reliability analysis using fault trees. In M. Lyu, editor, *Handbook of Software Reliability Engineering*, pages 615–660. IEEE Computer Society Press and McGraw-Hill Book Company, 1996.

[55] M. D. Fabro, J.Bezivin, F. Jouault, E. Breton, and G.Gueltas. AMW: A generic Model Weaver. In *Int. Conf. on Software Engineering Research and Practice (SERP05)*, 2005.

[56] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE*, pages 152–163, 2003.

[57] L. Frantzen, J. Tretmans, and R. d. Vries. Towards model-based testing of web services. In A. Polini, editor, *International Workshop on Web Services - Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Italy, June 9th 2006.

[58] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 23–34, 2004.

[59] C. M. G. McGraw and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27:1085–1110, December 2001.

[60] J. García-Fanjul, J. Tuya, and C. de la Riva. Generating test cases specifications for bpel compositions of web services using spin. In *International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, 2006.

[61] D. Garlan. Software Architecture: a Roadmap. In *ACM ICSE 2000, The Future of Software Engineering*, pages 91–101. A. Finkelstein, 2000.

[62] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on UML models. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 391–400, 2006.

[63] R. German. Non-Markovian analysis. In E. Brinksma, H. Hermanns, and J. P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 156–182. Springer-Verlag, 2001.

[64] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12(2):107–123, May 1990.

[65] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *Journal of Systems and Software*, 80(4):528–558, April 2007. doi:10.1016/j.jss.2006.07.023.

[66] D. Gross and C. Harris. *Foundamentals of Queueing Theory.* John Wiley & Sons, New York, NY 10158-0012, 1998.

[67] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[68] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. on Soft. Eng. and Method.*, 5(4):293–333, 1996.

[69] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Software Eng.*, 30(1):3–16, 2004.

[70] B. R. Haverkort. Markovian models for performance and dependability evaluation. In J.-P. Katoen, H. Brinksma, and H. Hermanns, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 38–83. Springer-Verlag, 2001.

[71] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[72] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, September 2003.

[73] R. A. Howard. *Dynamic Probabilistic Systems: Markov Models*, volume 1 of *Decision and Control.* John Wiley and Sons, New York, 1971.

[74] P. Inverardi, H. Muccini, and P. Pelliccione. DUALLY: Putting in Synergy UML 2.0 and ADLs. In *Proc. 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005*, 2005.

[75] Y. Jiang, S.-S. Hou, J.-H. Shan, L. Zhang, and B. Xie. Contract-based mutation for testing components. In *IEEE International Conference on Software Maintenance*, 2005.

[76] A. M. Johnson Jr. and M. Malek. Survey of software tools for evaluating reliability, availability, and serviceability. *ACM Computing Surveys*, 20(4):227–269, 1988.

[77] S. Lavenberg. *Computer Performance Modeling Handbook.* Academic Press, 1983.

[78] H. Ludwig. WS-Agreement Concepts and Use – Agreement-Based Service-Oriented Architectures. Technical report, IBM, May 2006.

[79] H. Ludwig, A. Dan, and R. Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreents. In *Proc. of Service-Oriented Computing - ICSOC 2004, Second International Conference*, pages 65–74. ACM, 2004.

[80] M. Caporuscio, D. Di Ruscio, P. Inverardi, P. Pelliccione, and A. Pierantonio. Engineering MDA into Compositional Reasoning for Analyzing Middleware-Based Applications. In *EWSA '05*, volume 3527 of *LNCS*, pages 475–490. Springer, 2005.

[81] M. Malhotra and K. S. Trivedi. Power-hierarchy among dependability model types. *IEEE Transactions on Reliability*, 43(3):493–502, September 1994.

[82] A. Marconi, M. Pistore, and P. Traverso. Specifying data-flow requirements for the automated composition of web services. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 147–156, 2006.

[83] E. Martin, S. Basu, and T. Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*, October 2006.

[84] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[85] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, 1982.

[86] A. Morzenti and P. S. Pietro. Object-oriented logic specifications of time critical systems. *ACM Transactions on Software Engineering and Methodologies*, 3(1):56–98, January 1994.

[87] A. Morzenti, M. Pradella, P. S. Pietro, and P. Spoletini. Model checking of trio specifications in spin. In LNCS, editor, *Proc. of 12th International FME Symposium*, volume 2805, Sep 2003.

[88] J. K. Muppala, R. M. Fricks, and K. S. Trivedi. Techniques for system dependability evaluation. In W. K. Grassmann, editor, *Computational Probability, Vol. 24 of Operations Research and Management Science*, pages 445–480. Kluwer Academic Publishers, The Netherlands, 2000.

[89] I. Mura and A. Bondavalli. Markov regenerative stochastic petri nets to model and evaluate the dependability of phased missions. *IEEE Transactions on Computers*, 50(12):1337–1351, 2001.

[90] I. Mura, A. Bondavalli, X. Zhang, and K. S. Trivedi. Dependability modeling and evaluation of phased mission systems: a DSPN approach. In *IEEE DCCA-7, IFIP Int. Conference on Dependable Computing for Critical Applications*, pages 299–318, San Jose, CA, USA, January 6-8 1999.

[91] S. Narayanan and S. McIlraith. Analysis and simulation of web services, 2003.

[92] OASIS. *Web Services Business Process Execution Language (WSBPEL) 2.0*, December 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[93] P. Pelliccione. *CHARMY: A framework for Software Architecture Specification and Analysis*. PhD thesis, Computer Science Dept., University of L'Aquila, May 2005.

[94] J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[95] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des Institutes fur Instrumentelle Mathematik, Bonn, 1962.

[96] M. Pradella, P. S. Pietro, P. Spoletini, and A. Morzenti. Practical model checking of ltl with past. In *Proc. of 1st International Workshop on Automated Technology for Verification and Analysis*, Dec 2003.

[97] M. Pradella, M. Rossi, and D. Mandrioli. ArchiTRIO: A UML-compatible language for architectural description and its formal semantics. In *Proceedings of Formal Techniques for Networked and Distributed Systems (FORTE2005)*, volume 3731 of *Lecture Notes in Computer Science*, pages 381–395. Springer-Verlag, 2005.

[98] M. Pradella, M. Rossi, and D. Mandrioli. A UML-compatible formal language for system architecture description. In *SDL 2005: Proc. of 12th SDL Forum*, volume 3530 of *Lecture Notes in Computer Science*, pages 234–246. Springer-Verlag, 2005.

[99] M. H. R. Pargas and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, April 1999.

[100] Robby, M. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the 9th European software engineering Conf.*, pages 267–276, 2003.

[101] M. Roper. Computer aided software testing using genetic algorithms. In *10th International Software Quality Week*, San Francisco, CA USA, 1997.

[102] S. W. Sadiq, M. E. Orlowska, W. Sadiq, and C. Foulger. Data flow and validation in workflow modelling. In *Database Technologies 2004, Proceedings of the Fifteenth Australasian Database Conference, ADC 2004, Dunedin, New Zealand, 18-22 January 2004*, pages 207–214, 2004.

[103] H. Saiedian, J. P. Bowen, R. W. Butler, D. L. Dill, R. L. Glass, D. Gries, A. Hall, M. G. Hinchey, C. M. Holloway, D. Jackson, C. B. Jones, M. J. Luts, D. L. Parnas, J. Rushby, J. Wing, and P. Zave. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, 1996.

[104] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.

[105] B. P. Shah. Analytic solution of stochastic activity networks with exponential and deterministic activities. Master's thesis, University of Arizona, USA, 1993.

[106] R. Siblini and N. Mansour. Testing web services. In *ACS/IEEE International Conference on Computer Systems and Applications*, 2005.

[107] D. P. Siewiorek and R. S. Swarz. *Reliable Computer System - Design and Evaluation.* Digital Press, 3rd edition, 2001.

[108] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj. A bayesian approach to reliability prediction and assessment of component based systems. In *Proc. of* 12^{th} *International Symposium on Software Reliability Engineering (ISSRE'01)*, 2001.

[109] J. Skene, D. Lamanna, and W. Emmerich. Precise Service Level Agreements. In *Proc. of 26th International Conference on Software Engineering (ICSE 2004)*, pages 179–188. IEEE Computer Society Press, 2004.

[110] C. Smith and L. Williams. *Performance solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison Wesley, 2002.

[111] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software.* PhD thesis, University of York, Apr 2000.

[112] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications*. John Wiley & Sons, New York, second edition, 2002.

[113] W. T. Tsai, X. Bai, R. Paul, K. Feng, , and L.Yu. Scenario-based modeling and its applications. In *IEEE WORDS*, 2002.

[114] W. T. Tsai, R. Paul, W. Song, and Z. Cao. Coyote: an xml-based framework for web services testing. In *7th IEEE International Symp. High Assurance Systems Eng. (HASE 2002)*, 2002.

[115] A. Urquhart. Many valued logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume III. Kluwer, 1986.

[116] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001. http://www.w3.org/TR/wsdl.

[117] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[118] J. Wegener and M. Grochtmann. Testing temporal correctness of real-time systems by means of genetic algorithms. In *Quality Week*, 1997.

[119] M. Weiss and B. Esfandiari. On feature interactions among web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*, pages 88–95, 2004.

[120] M. Weiss, B. Esfandiari, and Y. Luo. Towards a classification of web service feature interactions. *Computer Networks*, 51(2):359–381, 2007.

[121] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[122] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang. Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*, pages 75–84, 2006.

[123] Y. Yang, Q. Tan, Y. Xiao, F. Liu, and J. Yu. Transform bpel workflow into hierarchical cp-nets to make tool support for verification. In *Frontiers of WWW Research and Development - APWeb 2006, 8th Asia-Pacific Web Conference, Harbin, China, January 16-18, 2006, Proceedings*, pages 275–284, 2006.

[124] Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to bpel4ws test generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia*, 2006.

[125] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5), May 2004.

[126] L. Zhang and D. Ardagna. Sla based profit optimization in autonomic computing systems. In *Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings*, pages 173–182, 2004.