HAL
open science

Un meta-modèle de composants pour la réalisation d'applications temps-réel flexibles et modulaires

Joao Claudio Rodrigues Americo

To cite this version:

Joao Claudio Rodrigues Americo. Un meta-modèle de composants pour la réalisation d'applications temps-réel flexibles et modulaires. Autre [cs.OH]. Université de Grenoble, 2013. Français. NNT : 2013GRENM055 . tel-01167469

HAL Id: tel-01167469

https://theses.hal.science/tel-01167469v1

Submitted on 24 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

João Claudio RODRIGUES AMÉRICO

Thèse dirigée par **Didier DONSEZ**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

A component metamodel for designing flexible and modular real-time applications

Thèse soutenue publiquement le **4 Novembre 2013**,
devant le jury composé de :

Prof. Dr. Florence MARANINCHI
Professeur à Grenoble INP, Président
Prof. Dr. Lionel SEINTURIER
Professeur à l'Université de Lille I, Rapporteur
Prof. Dr. Jean BEZIVIN
Professeur à l'Université de Nantes, Rapporteur
Prof. Dr. Andy WELLINGS
Professeur à l'Université de York (Angleterre), Examinateur
Prof. Dr. Frédéric MALLET
Maître de Conférences à l'Université de Nice/Sophia Antipolis, Examinateur
Prof. Dr. Didier DONSEZ
Professeur à l'Université de Grenoble, Directeur de thèse
Dr. François EXERTIER
Responsable Technique, Bull, Co-Directeur de thèse
M. Frédéric SOINNE
Responsable Entreprise, Bull, Co-Directeur de thèse
M. Stephane ZENG
Ingénieur R&D, Bull, Invité

Remerciements

Ca y est, c'est fini!

Tout d'abord, je tiens à exprimer ma profonde reconnaissance aux professeurs Lionel Seinturier et Jean Bézivin d'avoir accepté d'être les rapporteurs de cette thèse, ainsi qu'aux professeurs Florence Maraninchi, Andy Wellings et Frédéric Mallet d'avoir accepté de participer à ce jury de thèse et contribuer au résultat final de ce projet.

Mes plus sincères remerciements à ceux qui m'ont soutenu et sans qui ce travail n'aurait jamais abouti :

Au Prof. Didier Donsez, mon directeur de thèse, pour m'avoir confié ce travail de recherche, ainsi que pour son aide, ses conseils, les nombreuses pauses-café, réunions et e-mails échangés au cours de ces dernières années. Je remercie également Dr. François Exertier et M. Frédéric Soinne pour m'avoir (re-)accueilli chez Bull et pour leur encadrement,

À Pierre Salkazanov, ainsi qu'aux professeurs Philippe Lalanda et Noël de Palma de m'avoir ouvert les portes des équipes BPS Innovation, ADELE et ERODS, respectivement,

J'aimerais remercier particulièrement le Dr. Walter Rudametkin et le Prof. Diogo Souza, qui sont à l'origine de toutes mes expériences professionnelles et scientifiques en France et au Brésil,

Ces remerciements ne serait pas complets sans une pensée pour plusieurs amis de longue et moyenne date, qui se trouvent au Brésil. Merci de m'avoir encouragé. Merci encore à la Xepa, à la Tropicolloc', à Gabriel, à Janine et à Cécile, qui ont été ma famille en France.

Mes dernières, mais pas les moins méritantes, pensées iront vers ma famille, surtout mes parents, qui m'ont toujours motivé et permis de poursuivre mes études jusqu'à aujourd'hui.

Merci beaucoup, thank you very much e muito obrigado.

Abstract

The increase of software complexity along the years has led researchers in the software engineering field to look for approaches for conceiving and designing new systems. For instance, the service-oriented architectures and component-based approaches are considered nowadays as some of the most advanced ways to develop and integrate modular and flexible applications. One of the software engineering solutions principles is re-usability. Consequently, generality is important, which complicates its application in systems where optimizations are often used, like real-time systems. Thus, creating real-time systems is expensive, because they must be conceived from scratch. In addition, most real-time systems do not benefit from the advantages which come from software engineering approaches, such as modularity and flexibility. Recently, some isolated solutions have been proposed to overcome this problem, such as real-time service-oriented architectures and real-time component models.

This thesis proposes a unified metamodel to design real-time service-oriented component-based applications. This metamodel allows dynamic architecture reconfiguration in real-time systems, making them flexible and modular. Services, components, connections and platforms can be modelled with real-time attributes, which are taken into account in offline and online verifications. Moreover, a methodology is proposed, covering the development process from design to execution.

In order to exemplify its generality, the metamodel is instantiated in extended versions of other model standards used to conceive both service-oriented and real-time applications, namely AADL, SCA and UML-MARTE. A Real-time Java-based framework for an extended version of the SCA component model has been implemented, and evaluations show that the expected quality of service is achieved for soft real-time applications.

Keywords: MDE, Metamodel, Real-time, SOA, Component-Based Design, Dynamic Adaptation

Résumé

La croissante complexité du logiciel a mené les chercheurs en génie logiciel à chercher des approches pour concevoir et projeter des nouveaux systèmes. Par exemple, les architectures orientées services (SOA) et l'approche orientée composants sont considérées actuellement comme deux des moyens le plus avancés pour réaliser et intégrer rapidement des applications modulaires et flexibles. Une des principales préoccuppations des solutions en génie logiciel et la réutilisation, et par conséquent, la généralité de la solution, ce qui peut empêcher son application dans des systèmes où des optimisation sont souvent utilisées, tels que les systèmes temps réel. Ainsi, créer un système temps réel est devenu très couteux, dû au fait qu'ils sont souvent construits en partant de rien. De plus, la plupart des systèmes temps réel ne bénéficient pas des facilités apportées par le génie logiciel, tels que la modularité et la flexibilité. Ces dernières années, quelques travaux isolés ont proposé des solutions à ce problème, telles que la SOA temps réel et les modèles de composants temps réel.

Cette thèse propose un méta-modèle unifié pour la conception d'applications orientées service et composant temps réel. Ce métamodèle permet la reconfiguration architecturale dynamique dans les applications temps réel, ce qui les rend plus flexibles et modulaires. Services, composants, connexions et plate-formes peuvent être modélisés avec des attributs temps réel, qui sont pris en compte dans les phases de vérifications avant et durant l'exécution de l'applications. En outre, cette thèse propose aussi une méthodologie qui couvre tout le processus de développement, dès la modélisation jusqu'à son exécution.

Pour démontrer la généralité de la solution, le metamodèle est projeté sur des versions étendues de modèles standards AADL, SCA et UML-MARTE. Une plate-forme de déploiement et exécution, basé sur le modèle SCA, a été implémentée, et les évaluations montrent que la qualité de service attendue a été obtenue pour des applications temps réel mou.

Mots-clé: IDM, Métamodèle, Temps-réel, Architectures orientée service, Modèle de composant, Adaptation dynamique

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Challenges . . . 2
1.3 Contribution . . . 3
1.4 Document Structure . . . 4

2 State of the art **5**
2.1 Introduction . . . 6
2.2 Service-Oriented Component Models . . . 7
2.2.1 Dynamic Software Architectures . . . 7
2.2.2 Component-Based Design . . . 10
2.2.3 Service-Oriented Architectures . . . 13
2.2.4 Service-Oriented Component Models . . . 16
2.3 Real-time Systems . . . 17
2.3.1 Principles . . . 17
2.3.2 Infrastructure of Real-time Systems . . . 19
2.3.3 Design and Development Techniques for Real-time Systems . . . 21
2.4 Adaptiveness and Dynamism in Real-time Systems . . . 26
2.4.1 Real-time Component Models . . . 26
2.4.2 Real-time Service-Oriented Architectures . . . 30
2.5 Summary . . . 32

3 Unified Real-time Service-Oriented Component Metamodel **35**
3.1 Motivations . . . 36
3.1.1 Problems in the development process of real-time systems . . . 37
3.1.2 Requirements for a real-time service-oriented component model and framework . . . 38
3.1.3 URSO main goals . . . 39
3.2 URSO Concepts . . . 40
3.2.1 Deployment Concern . . . 41
3.2.2 Assembly Concern . . . 49
3.2.3 Behaviour Concern . . . 60
3.2.4 Revisiting the Deployment Concern . . . 70
3.2.5 Other aspects that must be taken into account . . . 78
3.3 Example: Dynamic Collision Detection Application . . . 81
3.3.1 Overview of the CD_x Benchmark . . . 81
3.3.2 DCD_x: A Dynamic and Service-Oriented CD_x benchmark . . . 82
3.3.3 URSO Description of DCD_x . . . 83
3.4 Summary and Discussion . . . 89
3.4.1 URSO Overview . . . 89

3.4.2 Discussion . . . 92
3.4.3 Summary . . . 93

4 Mapping URSO Onto Existing Component Models 95
4.1 Motivations . . . 96
4.2 URSO and SCA . . . 96
4.2.1 Overview of SCA . . . 96
4.2.2 Extending SCA . . . 97
4.2.3 Mapping extended SCA to URSO . . . 104
4.3 URSO and AADL . . . 105
4.3.1 Overview of AADL . . . 105
4.3.2 Extending AADL . . . 108
4.3.3 Mapping extended AADL to URSO . . . 110
4.4 URSO and MARTE . . . 113
4.4.1 Overview of UML-MARTE . . . 113
4.4.2 Extending UML-MARTE . . . 117
4.4.3 Mapping extended UML-MARTE to URSO . . . 120
4.5 Summary and Discussion . . . 120
4.5.1 Overview on the extensions to SCA, AADL and MARTE . . . 120
4.5.2 Discussion . . . 122
4.5.3 Summary . . . 124

5 Implementation and Validation 127
5.1 Implementation . . . 128
5.1.1 SCA:PlatformDesc Command . . . 129
5.1.2 SCA:PlatformInfo Command . . . 130
5.1.3 SCA:ChangeMode Command . . . 131
5.1.4 SCA:List Command . . . 131
5.1.5 SCA:Deploy Command . . . 131
5.1.6 SCA:Undeploy Command . . . 132
5.1.7 URSO+NaSCAr framework architecture . . . 132
5.2 Usecase: Revisiting DCD_x . . . 135
5.2.1 Platform Description . . . 135
5.2.2 Service Compositions, Instances and Tasks: the Detector example . 138
5.2.3 DCD_j, a Java-based implementation of DCD_x . . . 141
5.3 Validation . . . 144
5.3.1 Methodology overview . . . 144
5.3.2 Platform description validation analysis . . . 145
5.3.3 Platform information analysis . . . 146
5.3.4 Contribution deployment analysis . . . 146
5.3.5 Contribution undeployment analysis . . . 149
5.3.6 Mode change analysis . . . 150
5.3.7 Execution timeliness analysis . . . 152
5.4 Summary and Discussion . . . 154
5.4.1 Overview . . . 154
5.4.2 Discussion . . . 155

6 Conclusions and Perspectives 157
6.1 Conclusions . . . 157
6.1.1 Summary . . . 157

6.1.2 Conclusions . . . 159
6.2 Perspectives . . . 161

A Appendix **165**
A.1 URSO name and logo . . . 165
A.2 Example: URSO Potential Use Cases Description . . . 166
A.3 Example: Mapping URSO Composite to AADL . . . 170
A.4 UML MARTE Simplified Metamodel . . . 174
A.4.1 Core Elements . . . 174
A.4.2 Non-Functional Properties . . . 174
A.4.3 Time . . . 175
A.4.4 Generic Resource Model . . . 177
A.4.5 Allocation . . . 177
A.4.6 Generic Component Model . . . 177
A.4.7 High Level Application Model . . . 179
A.4.8 Generic Quantitative Analysis Model . . . 179

List of figures **182**

List of tables **183**

List of algorithms **184**

List of Listings **185**

Bibliography **187**

Chapter 1

Introduction

"Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop."

Alice in Wonderland
Lewis Carroll

Contents

1.1 Context 1
1.2 Challenges 2
1.3 Contribution 3
1.4 Document Structure 4

1.1 Context

As the complexity of software systems increases, new techniques have been conceived to mitigate its impact on several steps of the application development process and improve efficiency in the management of important software properties, such as adaptability, portability and maintainability [Crnkovic 2005]. *Separation of concerns* is one of the main design principles applied by these techniques: it puts forward the separation of an application into distinct parts, which address different concerns (that is, aspects that affect an application in a particular way). Separation of concerns results in *modularity*, which is an important application capability in the structured programming paradigm. Modular applications are composed by independent and interchangeable *modules*, that communicate with each other by means of well-defined *interfaces*. Independence among modules makes development faster, since several programmers can work on smaller individual modules at the same time. In case of failure or maintenance, only specific modules can be replaced, easing debugging, recovery and update. In addition, modules can be reused in other projects; the most generic modules can actually be reused without change [Mitchell 1990].
Separation of concerns is the foundation of the component-based and service-oriented approaches in software engineering. In the component-based approach, the system processes are encapsulated into separate modules (called *components*), in a way that the internal data of each component are semantically related (*high cohesion*). Components interfaces are used to specify functions provided by the component and required for its functioning. Only the interface of a component can be seen by other components; it means that internal details related to its implementation (*i.e.*, the code that truly implements the component and its inner workings) can not be seen from the outside (*low coupling*). Components can be substituted by other components, as long as they implement the same interface [Szyperski 2002]. The Service-oriented architecture (SOA) design extends the component-based

architecture by adding a network in the components communication. The functionality provided and required by components becomes individual entities called *services*; these services can be published and discovered over a network, often by means of a *service registry* or *catalogue*. Common and industry-specific standards have been established for SOA applications, the most popular being related to the Web Services approach [Papazoglou *et al.* 2007]. Services and components have been extensively used for several years in many distributed, flexible and modular applications; still, there are some application domains which use these techniques in a lesser degree. One of these domains is that of real-time applications [Crnkovic 2005].
Real-time applications are applications whose correctness depends on both logical and temporal aspects. These temporal aspects are expressed by means of execution timing constraints, called *deadlines*. In order to satisfy those timing constraints, it is important to know the timing properties of all involved parts, from hardware to software components [Stankovic 1992]. Consequently, design approaches favouring dynamic (*i.e.* at run-time) adaptation are difficult to use. At the same time, they are often necessary, due to the interaction of real-time systems and real world entities, whose availability may be dynamically changed. In this context, multiple works have been consecrated to the adaptation of popular or the development of brand new techniques for the run-time adaptation of real-time systems [Bihari & Schwan 1991].
This thesis focuses on the adaptation of the service- and component-based approaches for real-time systems. It aims to deeply understand such techniques, find the common points between existing solutions and requirements for a service-oriented component-based real-time system and to propose a metamodel for such applications. This metamodel must take into account existing models and standards, so that these technologies can be reused in development of conform applications.

1.2 Challenges

The dynamism offered by service-oriented component models is intrinsically not compatible with the predictability required by real-time applications. Whereas the former offers dynamic availability support through transparent dynamic reconfiguration without downtime, the latter expects knowledge about the worst case execution time of all components so that the satisfaction of all timing constraints is guaranteed by means of formal and schedulability analysis. Indeed, there are very few works concerning the use of service-oriented components in real-time applications; most real-time SOA solutions do not include component models, and, similarly, component models for real-time do not support the service-oriented paradigm.
To address this limitation, this thesis considers the main existing models for both real-time and service-oriented applications and provides a metamodel and a methodology to design service-oriented component-based real-time systems. In order to do so, many issues had to be accounted for:

- We need to define what dynamism in real-time applications means. It is important to find a degree of dynamism which brings enough flexibility and does not compromise the determinism of the application;

- It is necessary to define what is a real-time service, where it is published, which service properties are published along with the service interface, and how this service can be retrieved by other components;

- Service properties should depend the least possible of the execution platform. However, time-related properties are necessary at the service selection. That means that it is up to the platform to qualify the service in regards to the platform where the service is being executed, taking into account communication channels that are used to link components;
- Metamodels must represent models, which in turn characterize systems. It is important to choose which models can be represented by the proposed metamodel, based on criteria such as generality, industry adoption, and expressiveness. In addition, if modifications or extensions must be brought to the target model, which corresponding changes must be made in its metamodel and how can they be done without disrespecting conformity rules;
- We need to identify which mechanisms must be implemented by the framework implementing the conform model, so that all constraints are respected at execution. Moreover, we must detail how these mechanisms must function, how they interact with the metamodel and how they can interfere in the application execution;
- It is important to find extension and implementation-specific points in the metamodel, so that it can be easily extended to support more functionality;
- It is desirable to provide tools for model transformation based on the metamodel, and to prove the consistency of the model transformation function;
- And finally, it is important to implement the proposed ideas and validate their feasibility.

This thesis aims to cover all these aspects and to present its application in the context of an aircraft collision detection benchmark.

1.3 Contribution

The contributions of this thesis cover three main points. The first is the principal contribution; the last two can be seen as consequences of the application and implementation, respectively, of the main contribution.

Proposition of a metamodel for real-time service-oriented component-based applications: We introduce the URSO (Unified Real-time Service-Oriented) component metamodel. URSO treats three concerns: *Deployment*, in which software entities are mapped onto physical platform components and may reserve resources for their execution; *Assembly*, where software components are described and composed, and services can be defined and qualified; and *Behaviour*, in which the internal behaviour of services is generated, and semantic rules and policies can be associated to services. We also suggest a methodology to apply in the design of URSO-compliant service-oriented real-time components.

Mapping of the proposed metamodel onto standard models: We define the mapping relations between URSO and three standard models for real-time and service-oriented component-based models: AADL, a standard component-based architecture description language used in the avionics domain; UML-MARTE, a UML profile which enables time-related descriptions in object-oriented real-time applications; and SCA, an extensible OASIS standard developed by major IT vendors for designing and implementing of service-oriented and

component-based applications. These mappings allow transforming applications developed in a URSO-compliant model to applications developed in another model. For instance, URSO-compliant SCA applications may benefit from AADL analysis tools. It is worth mentioning that in order to be conforming with the URSO metamodel, it was necessary to extend the target models. Consequently, we also present the extensions that were used in this work.

Implementation of a framework supporting the proposed metamodel: In order to demonstrate the feasibility and applicability of the proposed ideas, a framework prototype was developed to support a URSO-compliant extended version of the SCA component model. In order to validate our proposition, we have developed a dynamic version of the CD_x benchmark proposed in [Kalibera *et al.* 2009]. The framework has demonstrated to respect timing constraints established for soft real-time applications in Real-time Java.

1.4 Document Structure

The remainder of this document can be divided into four parts.
The first part comprises the state of the art, and includes basic concepts for understanding the challenges and the contribution of this thesis. It also presents some of the definitions and the terminology applied throughout this document. It corresponds to **Chapter 2**.
The second part presents our approach. It is divided into two chapters: **Chapter 3** introduces URSO, the core of our proposition, and its different concerns. It also exemplifies the application of URSO in a dynamic aircraft collision detector application. Following that, **Chapter 4** presents mappings between URSO meta-elements and AADL, SCA, and UML-MARTE elements. Moreover, it shows the extensions that were proposed for these models so that they are conform to the URSO metamodel.
The third part, which corresponds to **Chapter 5**, presents a framework prototype, implementing real-time and dynamic extensions presented for the SCA component metamodel. We also depict a use case implemented on this framework: the dynamic aircraft collision detection application. Evaluation metrics for the framework are presented in the end of the chapter.
Finally, the fourth part contains our conclusions and perspectives of future work. It is composed by **Chapter 6**.

Chapter 2

State of the art

"No one wants to learn from mistakes, but we cannot learn enough from successes to go beyond the state of the art."

To Engineer Is Human: The Role of Failure in Successful Design
Henry Petroski

Contents

2.1 Introduction **6**
2.2 Service-Oriented Component Models **7**
2.2.1 Dynamic Software Architectures 7
2.2.1.1 Software Architectures 7
2.2.1.2 Dynamic Software Architectures 8
2.2.1.3 Architecture Description Languages and Dynamism . . . 8
2.2.1.4 Dynamic Software Adaptation 9
2.2.2 Component-Based Design 10
2.2.2.1 Principles 10
2.2.2.2 Component Models 11
2.2.3 Service-Oriented Architectures 13
2.2.3.1 Principles 13
2.2.3.2 Dynamic Service-Oriented Architectures 14
2.2.4 Service-Oriented Component Models 16
2.2.4.1 Principles and Motivations 16
2.2.4.2 Existing Service-Oriented Component Models 16
2.3 Real-time Systems **17**
2.3.1 Principles 17
2.3.1.1 Predictability in real-time systems 18
2.3.1.2 Determinism in real-time systems 19
2.3.2 Infrastructure of Real-time Systems 19
2.3.2.1 Real-time Operating Systems 19
2.3.2.2 Real-time Middleware 20
2.3.2.3 Real-time Programming Languages 21
2.3.3 Design and Development Techniques for Real-time Systems 21
2.3.3.1 Formal Methods 22
2.3.3.2 Structured Methods 23
2.3.3.3 Object-oriented Analysis and Design 24
2.4 Adaptiveness and Dynamism in Real-time Systems **26**
2.4.1 Real-time Component Models 26

2.4.2 Real-time Service-Oriented Architectures 30
2.5 Summary **32**

2.1 Introduction

A few decades ago, software development basically consisted of programming. The industry maturation has significantly changed this scenario in the late 60s, when software engineering was first recognized as a computer science discipline in itself, and the software development started to be seen as an engineering discipline rather than an art.
Since then, increasing software complexity has led software engineers to look for new methods of designing, constructing and maintaining applications. The industry pressure forced software development methods to focus on non-functional requirements as well as functional ones [Cooling 2003]:

- Developing complex applications became expensive. Thus, software engineering techniques must reduce development **costs** as much as possible;
- At the same time, due to code complexity, non-functional properties such as **safety**, **quality** and **reliability** must be improved and ensured;
- The **time** dimension has also become very important: time taken for designing and developing applications, also known as *time to market*, must be minimal.

The complexity and constant evolution of software originated many of desirable properties for systems (which may be present or not depending on the software purpose and the environment in which it is executed) such as:

- Flexibility, that is, the ease with which a system can be adapted for use in environments other than those for which it was initially designed for [IEEE Standards Comittee 1990];
- Dependability, defined as the ability to deliver service that can justifiably be trusted, that is, the system reliance can justifiably be placed on the service it delivers [Avizienis *et al.* 2004];
- Resilience, *i.e.*, the persistence of dependability when facing unpredicted environmental changes [Avizienis *et al.* 2004];
- Robustness, *i.e.*, whether a system can function correctly or not in the presence of faulty inputs and environmental conditions [IEEE Standards Comittee 1990];
- Recoverability, *i.e.*, the ability of a system to restore itself to the point at which a failure occurred;
- Configurability, that is, whether an application can be fit to one's needs by adjusting its parameters[1] [Sun *et al.* 2008].

[1] Configurability can be measured by an application's configuration and customization capabilities. Configuration can be distinguished from customization by the fact that the former refers to adjusting an application simply by changing parameters without touching its code (for instance, by means of a *configuration* file), whereas the latter involves coding, which is riskier and more costly.

Therefore, the ability to adapt software in response to changes in the system's environment and in the system itself - called *software adaptation* - has become a very important research topic in software engineering. This adaptive behaviour must be designed and expressed in all software dimensions, from its requirements to its implementation [Cheng *et al.* 2009]. These aspects are mapped on top of software architectures and frameworks supporting software adaptation statically or dynamically. Most of the approaches used to implement software adaptation are based on software modularity, framework containers to automatically manage adaptations and, when performed automatically, calls and method interceptors [Oreizy *et al.* 2008].
However, when it comes to real-time systems things are different. Many of the methods proposed by the software engineering discipline are not suitable for real-time systems. The engineering of real-time adaptive software is a challenge, due to its need of predictability. But at the same time, with the emergence of ubiquitous [Weiser 1993] and autonomic [Kephart & Chess 2003] computing, real-time applications may indeed have the need to take environment changes into account in a cost-effective way.
The aim of this chapter is to discuss these background notions needed to understand the problems underneath designing and implementing of dynamic real-time service-oriented and component-based applications and provide the state-of-the-art in research of both service-oriented computing and real-time systems fields.

2.2 Service-Oriented Component Models

In this section, we will present the state-of-the-art in service-oriented component models (SOCMs). But first, we are going to introduce more basic notions, such as dynamic software architectures and the main approaches used to represent them: architecture description languages (ADLs), component-based software engineering and service-oriented architectures (SOA). The SOCM approach, as the name suggests, reunites the benefits of both service-oriented and component-based approaches, and is the basis for the main contribution of this thesis.

2.2.1 Dynamic Software Architectures

2.2.1.1 Software Architectures

Complexity issues have been present in the computer science domain since its early days. The first problems related to programming complexity were solved through the use of data structures, the development of algorithms and scope separation. Most of what we know now as software engineering comes as results of the increasing complexity problem. The first works about the importance of structuring software systems were developed by Dijkstra [Dijkstra 1968] and Parnas [Parnas 1972], in the late 1960s and early 1970s, respectively. These works were the basis for the software engineering discipline called *Software Architecture*.
A *Software architecture* is an abstraction used for structuring software systems, by representing their **software components**, their **interconnections**, **properties** over both components and interconnections, and the rules concerning their design and **evolution** over time.
Garlan and Shaw's definition [Garlan & Shaw 1994, Garlan & Perry 1995, Shaw *et al.* 1995] considers that a software architecture is defined by its components and their connections. Perry and Wolf's definition [Perry & Wolf 1992] considers that the data transmitted between components by means of their connections are also part of a software architecture. This definition is more adapted to network and data-based architectures, in which data is

important to determine the system's behaviour.
We consider that a software system may be represented by several different software architectures during its execution. Many aspects of a system can be addressed in its architectural descriptions, such as functional and non-functional requirements and different configurations. Software architectures help dealing with software scalability, evolution and complexity issues in software. Jointly with compositional techniques, their higher level of abstraction may ease systems' design, analysis and implementation.

2.2.1.2 Dynamic Software Architectures

Dynamic software architectures add the dynamism dimension to software architectures. They are architectures in which the interconnections between components may change during the system's execution. This behaviour is known as *runtime evolution or adaptation* [Taylor *et al.* 2009]. The main motivations for this type of adaptation are risks, costs and inconveniences incurred by downtime of computation-intensive systems in order to be adapted (by an administrator or by the software itself) due to changes in its environment [Oreizy *et al.* 2008].
The ability to dynamically reconfigure an application can be useful in many application domains. For instance, some applications require dynamic software updates, in order to fix bugs or to add new features, without stopping and restarting the full application. Some examples are critical and non-stop applications, such as financial and air-traffic control systems, which must provide a continuous service [Magee & Kramer 1996a].
This flexibility does not come without a cost. Although we can perform modifications which were not foreseen during the application design phase, it is hard to anticipate the effects of a dynamic modification. It can affect predictability and safety, which is inadmissible for safety-critical systems; the updated modules may require more disk space and more resources, which may be unacceptable for constraint and embedded systems; updates can add unsafe third-party modules, shutting down the whole platform; or it can bring the application to a wrong and inconsistent state, after applying the changes required for the adaptation.
Figure 2.1 exemplifies the safety issues on dynamic software adaptation. At design time, we specify that component B is connected to component A by means of Interface I1. At run-time, this structure is preserved; however, after an update, the new module may have dependencies towards other modules, unknown at design time, which are incorporated with the architecture.
Frameworks supporting run-time adaptation must ensure that changes will remain transparent to the applications. In order to do that, they usually perform modifications in the binary code, injecting code or intercepting calls.

2.2.1.3 Architecture Description Languages and Dynamism

Architecture Description Languages (ADLs) are languages and/or models used to describe and to represent software architectures. The ISO/IEC/IEEE 42010 specification [ISO/IEC/(IEEE) 2007] has defined and standardized the minimum requirements for ADLs. The use of ADLs helps to take and to communicate early design decisions and to have a system abstraction on which architects, developers and clients can reason together. ADLs may have both graphic and textual syntaxes, being able to represent key properties of complex architectural styles. ADLs are usually human and machine readable, and the latter often enables automatic code prototype generation and formal analysis and verification [Allen & Garlan 1997, Perry & Wolf 1992].

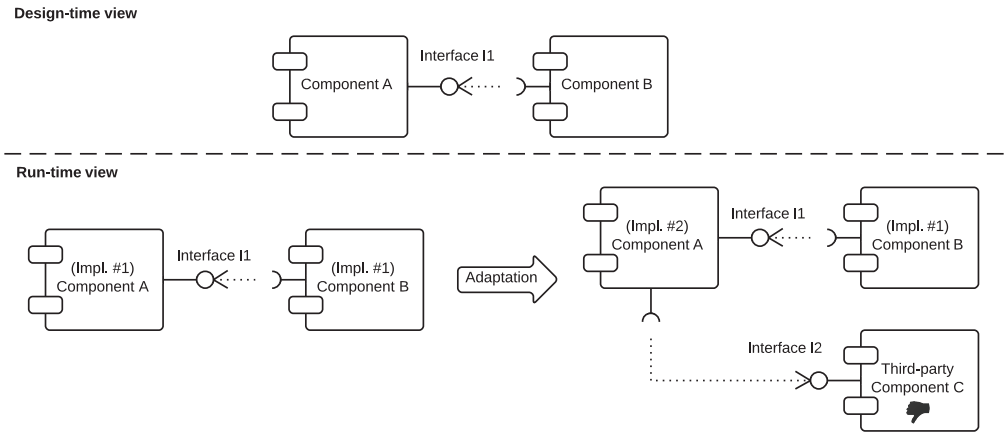

Figure 2.1: Dynamic Software Evolution

Among the many existing ADL used in the industry and in the academy, only few support the expression of dynamic reconfiguration. ACME [Batista *et al.* 2008], Dynamic Wright [Allen *et al.* 1998], Darwin [Magee & Kramer 1996b], Rapide [Vera *et al.* 1999] and C2ADL [Medvidovic 1996] are typical examples. Most ADLs do not support both predicted and unpredicted reconfigurations. A detailed analysis of problems on handling dynamic adaptation in ADLs is provided in [Minora *et al.* 2012].

2.2.1.4 Dynamic Software Adaptation

Two main approaches are used to implement software adaptation: *parametric adaptation*, in which modification of system variables is enough to change the system behaviour; and *compositional adaptation*, where software components are added or replaced in order to adapt a system to its environment. Parametric adaptation allows tuning application parameters, however its adaptation mechanism is limited, since it is not possible to adopt new strategies. On the other hand, compositional adaptation permits an application to be recomposed during its execution. This dynamic re-composition/adaptation differs from static (or design time) adaptation, where modifications are made before the system is running (*e.g.*, in source code or in requirements) [Brogi *et al.* 2006].
Most approaches implementing dynamic compositional adaptation are based on dynamically linking/injecting component objects or indirectly intercepting and redirecting interactions among software entities [Fox & Clarke 2009]. Several techniques may be used to implement such functionality, like pointer manipulation, aspect weaving, proxies or interception by means of middleware [McKinley *et al.* 2004]. In the next sections, we are going to detail two software engineering and architecture approaches commonly used in dynamically adaptive software: Component-based design and Service-oriented architectures. Both approaches are based on the *separation of concerns* principle. This principle emphasizes the separation of the application logic from cross-cutting concerns (such as quality of service, synchronization, security and fault tolerance) at conceptual and implementation levels. It allows for simplifying software development and maintenance, and making software easier to be reused [Hürsch & Lopes 1995].

2.2.2 Component-Based Design

2.2.2.1 Principles

Component-based design uses components as its basic software abstraction. Software components are software units, which can be composed in order to build a complete complex system. Components have contractually specified interfaces and contain explicit context dependencies. These units can be independently developed and deployed. Component composition can be static or dynamic, depending on when the developer is able to add, remove or reconfigure components (compile time or run time, respectively). Dynamic adaptation in component-based systems can be performed using late binding mechanisms, which allows coupling components at run-time, just before its invocation, through well-defined interfaces. This architectural style also promotes software reuse, reduces production cost (because software systems can be built from existing code) and shortens time to market [Clements 2001]. By decoupling the system into distinct components with low coupling and high cohesion, developing high-quality and consistent complex systems becomes more effective, since developers may focus on one component at a time.
Thus, three basic characteristics can be distinguished in the component-based software engineering [Szyperski 2002]:

- **Isolation**: Components can be deployed and developed independently as an isolated part of the system. A Component should be an atomic unit of deployment (*i.e.* can not be deployed partially);
- **Composability**: Components should be able to be assembled with other components in order to create a composed and more complex component (often called *composite*). This assembly must be made by means of the component's well specified interface, which corresponds to the list of self-contained operations implemented by this component; and
- **Opaqueness**: Internal and implementation details from a component should not be accessible by other components or by the environment. Opaque components are also called **black-box** components.

Using a component-based design however may bring some disadvantages and risks to a software project [Crnkovic 2001]:

- *Increased time and effort required for developing reusable components*, due to the need to correctly design and document it. In addition, reusable parts of a software tend to be changed more often than non-reusable parts until they reach a stable state;
- *Ambiguity and incompatibility in components requirements*, which come from the fact that components are supposed to be general and adaptable enough to be used in different applications. Consequently, component requirements (functional and non-functional) tend to be unclear or incompatible with certain applications;
- *Increased component maintenance cost*, which is another consequence of component generality. Despite the fact that application maintenance is decreased, components must still be adapted to the application's context, environment and requirements before being executed;
- *Reliability issues*, which again come from the component generality principle. The interaction between components created with different requirements, life-cycles, and potentially, technologies may introduce failures and a non-negligible risk that the final

application's requirements might not be completely satisfied. Hence the need of an approach to develop components from design to deployment, considering different technologies and development processes.

2.2.2.2 Component Models

The foundation of a component-based methodology lies on its *component model*. A component model defines what components are, how they can be constructed, implemented, assembled, and deployed. Component models also specify how interfaces between components should be defined and what these interfaces must contain. Most component models use objects or architectural elements as components. Thus, ADLs can be seen as component models as well.
In [Lau & Wang 2007], the authors define *software component model* as the specification of the *semantics*, the *syntax* and the *composition* of components.
Semantically, components are seen as software units with provided services (operations performed by component implementations) and required services (services needed by component implementations during the execution of their provided services). A component interface is defined as the specifications of both required and provided services. Components may sometimes present multiple interfaces with different sets of provided and required services or one single interface, consisting of the fusion of all component services' specifications. Provided services often correspond to the methods contained in the component implementations in object-based component models (also called operation-based component interfaces). Examples of models containing operation-based interfaces are COM [Box 1997], Java-based component models (such as EJB [Burke & Monson-Haefel 2006] and JavaBeans [Morrison 1997]) and Fractal [Bruneton *et al.* 2006]. In ADLs, provided and required services are ports, which are linked by means of connectors. These ports may either be distinguished as input (required service) or output (provided service) ports, or not (the same port is used by both service provision and consumption). Port-based interfaces are supported by BIP [Basu *et al.* 2006], PECOS [Gen*et al.* 2002], ROBOCOP [Muskens *et al.* 2005] and SaveCCM [Hansson *et al.* 2004], among others. Interfaces in the OMG standard CCM support both operation-based and port-based approaches.
Concerning the syntax of component models, most models contain a component and/or interface definition language. After its definition, a component can be realized in different implementation languages. However, in some component models, the implementation and component definition language happen to be the same. For instance, in the Java-based component models JavaBeans and EJB, components correspond to a Java class. Other models, like COM and CCM [Object Management Group 2006a], define a technology-independent interface definition language (IDL) for specifying components interfaces.
In component models where the main elements correspond to architectural concepts, ADLs are used to define both components and interfaces. That is the case in the component models defined by Koala [van Ommering *et al.* 2000], PECOS, Fractal and SOFA [Plásil *et al.* 1998]. The component and interface definition languages may describe different aspects of components: for instance, in BIP, developers are able to describe components' behaviour, interaction and interactions' priorities.
Composition is a fundamental issue in component-based approach, since complex systems are supposed to be built by the assembly of simpler components. System composition can be defined by means of a composition language, which must be compatible with components as they were defined in the component model. Some component models (such as JavaBeans, COM and CCM) do not contain a composition language, whereas other define the compositions by means of the architecture connectors (like Koala), ADLs (like PECOS and

SCA [Beisiegel *et al.* 2005]) or UML (like KobrA [Atkinson *et al.* 2008] and UML2.0 [Object Management Group 2006c]). In addition, the composition process may be classified as horizontal (*i.e.* when the interface of a service consumer is bound to an interface of the same type of a service provider, resulting in a set of components cooperating to realize a given functionality) and/or vertical (also known as *hierarchical composition*, that is, connecting a component required (or provided) service interface to another inner component required (or provided) service interface, so that calls originated by the inner component (or calls towards the inner component) are *delegated* by the outer component) [Crnkovic *et al.* 2011]. Composition is based on a theory, which allows developers and architects to predict the composition result before execution. Depending on the component model, the composition may take place at different stages (for instance, design, deployment or execution time).
Some other aspects that may be defined by component models are:

- Components life cycle, that is, an abstraction of the states in which the components are allowed to be during their deployment, execution and reconfiguration processes, the meaning of each state and its impact on the whole application and on other components, the transitions between these states and the conditions that must be respected in order to switch from a state to another;

- Components' communication type, *i.e.* the interaction style between service providers and consumers (*e.g.* broadcast, request-response, event-driven, etc.), the synchronism of this communication, how binding both entities is performed and which part of the system is responsible for doing so;

- Hierarchical composition, that is, if sub-components can be hidden inside the component that includes them;

- Non-functional properties, that means, properties which are not specific to the functionality of a system and that may be used to judge its operation. Contrarily to the system functional properties, which are usually defined during the system design phase, non-functional properties are related to system architecture. Depending on the component model, non-functional properties may be managed inside the component itself, or through an external mechanism (*e.g. configuration files*). After the definition of non-functional properties, the component model decides as well what should be done with this information: for instance, in real-time systems, it is common to use non-functional properties to perform analysis and to verify the correctness of the system composition;

- Components' packaging, *i.e.* how can we package components and their descriptors in order to install, remove, and manage them, which package formats are supported and what information must be present in such packages;

- And components' deployment, that is, how components are deployed in the system, whether they are retrieved from a file system placement or a repository, how components are instantiated from the executable files in the system, and then how components are assembled in order to be executed.

Detailed surveys and classifications on component models are presented in [Feljan *et al.* 2009] and [Crnkovic *et al.* 2011].

2.2.3 Service-Oriented Architectures

2.2.3.1 Principles

Service-oriented computing is a paradigm in which autonomous and and platform-independent entities called *services* are used to develop low-cost, interoperable, adaptable and distributed applications. Service-oriented computing principles are implemented by means of Service-oriented architectures (SOA). Service-oriented architecture is an architectural style and a programming model that may be used for implementing separation of concerns and potentially dynamic and distributed adaptive applications [Papazoglou *et al.* 2007].
In the service-oriented approach, a service is defined as a software unit which performs a function and whose operations and properties are described in a *service descriptor* or *service contract*. It is by means of these descriptors that different pieces of software are going to interact with each other. Service descriptors contain the operations provided by a given service and how these operations can be invoked. Service and service descriptor concepts are very similar to the concept of (operation-based) component interfaces in the component-based software engineering. Indeed, these concepts derived from the design-by-contract approach conceived for software modularization in object-oriented systems [Meyer 1997].
After their deployment, services advertise their arrival at a *service directory* or *service registry*. Service registries can be centralized or distributed. In some service-based technologies, this mechanism, called *service publication*, is performed by means of a broadcasted message; in some others, the service registers itself directly at the service directory. Another possible publication mechanism is the automatic detection of services during the deployment of new software units. Applications can look up services by querying their names and properties. This query may return details about how to interact with an implementation of a desired service or even an instance of such implementation. This querying mechanism is known in the service literature as *service discovery* or *service lookup*. Service consumers may also filter service implementations based on the service properties published in a service registry. This mechanism is called *service selection*. The binding between service consumers and providers can be performed after these negotiation and agreement of service usage terms [Raibulet & Massarelli 2008]. These terms can be monitored at run-time and the violation of the terms by one of the parts may incur the application of special policies by the underlying platform. The whole interaction mechanism among the different actors in a service-oriented architecture is depicted in Figure 2.2.
The service-based approach was popularized by the Web-services technology [Weerawarana

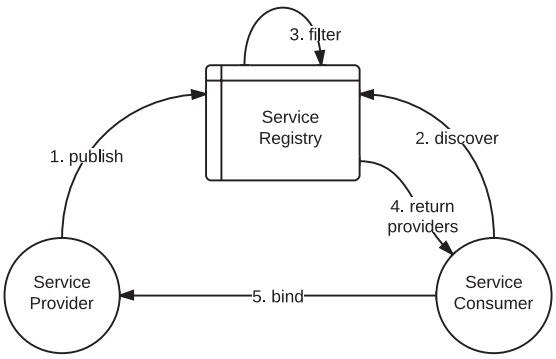

Figure 2.2: Service-Oriented Architecture: Actors and Interactions

et al. 2005]. Web services are Internet-based services, identified by a Uniform Resource Identifier (URI). Web-Services use open Internet standards, such as Simple Object Access Protocol (SOAP) for data exchange, Business Process Execution Language (BPEL) for orchestrating services in business processes and Web Services Description Language (WSDL) for describing service contracts. Discovery in Web-services may be addressed by two different protocols: UDDI (Universal Description, Discovery and Integration), which provides a central Service Registry, known by all participant applications; and WS-Discovery, in which service consumers broadcast a service request to locate candidate services and their properties. Due to their interoperability capabilities, Web Services have been widely used to implement cross-enterprise software communication.

2.2.3.2 Dynamic Service-Oriented Architectures

Dynamic SOA adds the dynamism dimension to the SOA. Two different forms of dynamism can be added to SOAs:

- *Dynamic availability* of services [Cervantes & Hall 2004], which means that services can change their availability status at any moment at run-time, without the control of the application;
- And *dynamic reconfiguration*, that designates services whose properties (thus, whose service description) can be modified at run-time.

Dynamic availability allows a system to evolve without downtime by means of dynamic service publication, update and removal. Figure 2.3 extends the diagram depicted in Figure 2.2 for the dynamic case.

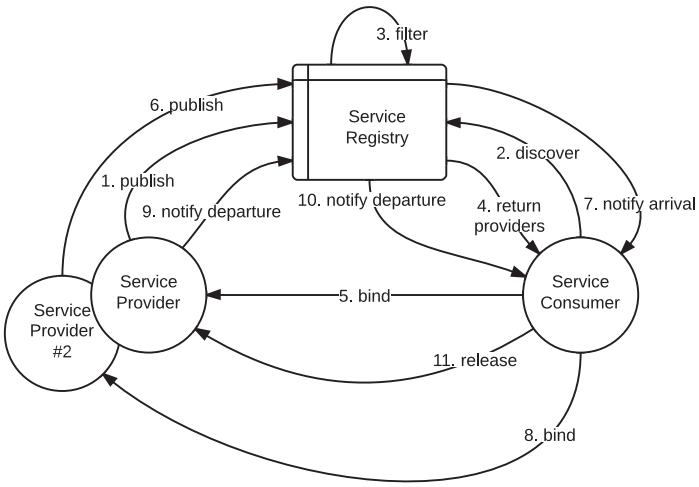

Figure 2.3: Dynamic Service-Oriented Architecture: Actors and Interactions

The research in services dynamic availability was stimulated by two main factors. First, the booming of highly extensible applications, in which new functionality and content support may be dynamically incorporated by the user. In this case, applications are constantly adding new modules to their architecture to integrate new services. Second, the use of unreliable and remote entities in applications, which can be unexpectedly disconnected from

a network (*e.g.* services provided by plug and play devices, or web services provided by a third-party provider through an unreliable network) or whose battery power may end during an execution (*e.g.* services provided by physical devices in a smart home or by smart communication devices).
Consequently, most protocols and technologies developed to support dynamic service availability were inspired by smart homes and devices. **UPnP** (Universal Plug and Play) [UPnP Forum 2000] is a platform-agnostic device-level standard that abstracts communication protocols but relies on specific languages for describing devices and service contracts. UPnP devices can be connected by means of an ad-hoc network in which services are announced through broadcasted messages and can be discovered and consumed. In addition, the UPnP standard supports event-based communication and remote operation of devices through control commands.
DPWS (Devices Profile for Web Services) [Chan *et al.* 2005] is profile for Web Services that supports dynamic discovery and event capabilities. Contrarily to UPnP, DPWS discovery and interoperability can be extended beyond local networks. Since it is aligned with the Web Services standard, it arranges several of the standards used for Web Services, and it is technology-agnostic. Just like UPnP, DPWS is an industrial standard protocol. Other industrial service-oriented protocols targeting devices include **IGRS** [Group 2003] and **Echonet** [Matsumoto 2010].
Another framework that targets devices is the TAco Next Generation Objects (**TANGO**)[2] [Chaize *et al.* 1999]. TANGO is a CORBA-based framework whose communication layer uses both CORBA (for synchronous and asynchronous communication) and ZeroMQ[3] (for event-based communication) libraries. This communication is performed through a common bus (that can be seen as a service registry) in which devices can be added. In addition, the bus provides central standard services such as logging and security. TANGO applications can be written in different languages, such as Python, Java and C++.
Jini [Arnold 1999] was a Java-based service platform developed by Sun Microsystems in which services could be dynamically deployed in a network. Jini services were defined by Java classes and interfaces. When entering a Jini architecture, service providers and consumers broadcasted announcements which were received by search services that were specially provided by the platform. These announcements were replied, so that the new members could have the registry reference. In order to support dynamic availability, consumers were notified about the availability of the services they were using. The project Apache River[4] and Rio Project[5] are both based on Jini's source code.
Similarly, the **OSGi** Service Platform [The OSGi Alliance 2012] is a platform where Java services can be dynamically deployed. Although OSGi's main objective is to provide a modularization solution for Java applications, it also provides a service layer, by which modules can interact and communicate with each other. OSGi modules (also called *bundles*) can publish and discover services in a non-centralized shared service catalogue, which is accessible through a special object used to access OSGi framework facilities. Services can be accessed through service implementation objects which are published along with the service interface and the service properties in the OSGi service registry. In addition, OSGi provides an event-based canvas for notifying service consumers about the arrival and withdraw of new and existing services, respectively.

[2] More information available at `http://www.tango-controls.org/`.
[3] ZeroMQ is a socket library that provides a fast and concurrent framework. It supports several communication protocols, programming languages and OSs. More information available at `http://www.zeromq.org`.
[4] Available at `http://river.apache.org`
[5] Available at `http://www.rio-project.org/`

2.2.4 Service-Oriented Component Models

2.2.4.1 Principles and Motivations

Despite the similarities between the component and the service-based approaches, their main difference remains the fact that they are focused on different actors: while the former focuses on the provider's view, easing the deployment of new functionality, the latter focuses on the consumer's view, to supply functions to consumers independently of service implementations [Rouvoy *et al.* 2008]. In addition, in the component-based approach, applications are built from blocks which are integrated at assembly time, whereas in the service-oriented approach only service descriptions are accessible at assembly time. Other differences are the explicit separation between component and instances and the importance of the packaging activity in the component-based approach; none of these are considered in the service-based approach, which focuses on the activities happening after service registration and discovery [Cervantes & Hall 2004].
Service-oriented component models (SOCMs) combine advantages from both service-oriented and component-based approaches. They were born from the need to add explicit dynamism and dynamic availability support in component models. In this approach, components are bound by means of services; bindings are thus non-explicit and performed as late as possible (late-binding), enabling component (and consequently, service provider) substitutability. An application is formed by a composition whose architecture may evolve and be adapted at run-time depending on service availability. Dynamic service adaptation may occur automatically, if it runs on a supporting framework, contrarily to the component-based approach, in which dynamism support usually requires more code development.
Dynamic availability is also the origin of the main problem of service-oriented component models: unpredictability. The availability changes must be informed and taken into account by other compositions, their internal components and the instances of these components. Component departure implies a query for alternative service implementations, and if no implementations can be found for replacement, both composition and framework must decide what to do with the unsatisfied service dependency.
Next section presents some existing service-oriented component models and their main characteristics.

2.2.4.2 Existing Service-Oriented Component Models

Most existing SOCMs are OSGi (and consequently Java)-based. They were created to ease the development and management of components in the OSGi Service platform.
For instance, in the Apache Felix **iPOJO** [Escoffier *et al.* 2007] component model, the framework injects Java objects at execution time and manages service provision, dependency and other aspects by adding a container around each component. iPOJO is an extensible component model, which means that other aspects (called *handlers*) may be added to the container, such as persistence, security and other non-functional concerns. iPOJO's container also manages components instances life cycle. Instances may be valid or invalid depending on the state of all its handlers. Meta-data for handler configuration can be provided by means of XML files, Java annotations or an API provided by the framework. iPOJO requires a build step in order to add the code that corresponds to the container to the service implementation classes. iPOJO (and OSGi as well) has been adapted for Python [Calmant *et al.* 2012].
Similarly, the OSGi **Declarative Services (DS)** model also allows the specification of dependencies between OSGi components under the form of required and provided services. This information, which is found in a XML file, is used by the Service Component Runtime

container that automatically manages dependencies, service registration and component binding. Services are "lazily loaded", that is, they are only loaded if they are used by another component. This behaviour saves start-up time and memory space. Both iPOJO and DS are based on the Service Binder component model for OSGi [Cervantes & Hall 2004].
Spring Dynamic Modules (**Spring DM**)[6] is an adaptation of the very popular dependency injection framework Spring for OSGi platforms. Recently, the Spring DM model has originated the **Blueprint Service** specification which is present in the OSGi official specification, just as the DS model [The OSGi Alliance 2012]. The Blueprint Service model uses the extender pattern, in which an external entity (in this case, an external bundle) monitors the bundles present in the framework and, based on their state, performs actions on these bundles, if they are Blueprint bundles. A bundle is a Blueprint bundle if it contains a Blueprint XML file describing its services, dependencies and properties. For each Blueprint bundle, a Blueprint container is created. This container is responsible for parsing, instantiating, binding and managing services of the Blueprint component (called *Bean*).
Service Component Architecture (SCA) [Beisiegel *et al.* 2005] is currently the only service-oriented component model not based on the OSGi Service Platform. SCA is technology-independent and its assembly model can be extended to support new client implementation, binding and interface types. In SCA, components provide and require services, which in turn are described by interfaces. Components can be organized into larger structures, called composites. Inside a composite, components can be bound dynamically or statically by means of pre-defined links called wires. Components' services and dependencies can be promoted, so that they can be seen by other composites in the system. The SCA specification does not inform however how the links between different composites can be established inside a SCA framework. An extension and a framework to support dynamism and dynamic availability at both component and composite levels have been proposed in [Américo & Donsez 2012]. Seinturier *et al.* have also taken into account reconfiguration issues in the Fractal-based SCA runtime FraSCAti [Seinturier *et al.* 2009]. This reconfiguration is performed by means of an API and requires the specification of a target and a source component for each binding. The CEVICHE framework [Hermosillo *et al.* 2010] extends those capabilities to the context of Web Services and business processes.

2.3 Real-time Systems

In computer science, the term "*real-time systems*" designs systems whose correctness depends on both logical and temporal aspects [Stankovic 1992]. While the former is required by all software systems, the latter comes from the impact these systems have on the real world.
Although the basis and theory of real-time systems are very well established and permanent, they remain a developing and very promising area for research. That is due to the evolution in computer architectures and the application of real-time technologies in new application domains.
The next sections present an overview on real-time systems and technologies.

2.3.1 Principles

Real-time systems can be defined as "systems whose logical correctness depends on both the correctness of the outputs and their timeliness" [Laplante & Ovaska 2011, Stankovic 1992]. The timeliness in real-time systems may be expressed as **timing constraints** for the set

[6] Available at http://www.springsource.org/osgi

of tasks which composes the system. Depending on the arrival pattern of these tasks, they can be classified as **periodic**, **sporadic** or **aperiodic**. Periodic tasks must be executed within regular time intervals, while the arrival time pattern in sporadic and aperiodic tasks is unknown; however, in the former, time interval between two releases[7] of the task is known to be greater than or equal to a constant, whereas in the latter, time interval between two releases is completely unknown (greater than or equal to zero). Thus, in order to satisfy the different timing constraints present in an application, it is important that the algorithms and activities performed by the system are executed in **bounded** time.
Timing constraints, also known as **deadlines**, can be relative to an event or absolute, precising a point in time for the termination of a task execution. Depending on task deadline enforcement, real-time systems can be divided into **soft, hard and firm real-time systems**. In *soft real-time systems*, the violation of a deadline leads the system to a state where its performance is degraded but not destroyed. In hard real-time systems, the violation of a single deadline may lead to a complete system failure and potential safety issues. The definition of *firm real-time systems* is unclear: for some authors [Kopetz 2011, Liu *et al.* 2006, Mok 1996], the results produced after the deadline violation are useless but consequences are not severe; for some other authors [Shin & Ramanathan 1994, Laplante & Ovaska 2011], they designate systems which tolerate the violation of a few deadlines, but missing more than that may lead the system to a complete failure.
Thus, real-time systems do not need to be fast: they must be **predictable** and **deterministic**. These two concepts will be discussed in the sections below.

2.3.1.1 Predictability in real-time systems

Predictability means that it is possible to guarantee (very often mathematically), for a given set of real-time tasks, that all timing constraints will be met. This demonstration is usually performed at design time, when the decoupling of the system into tasks is performed and their deadlines are associated. Sometimes, however, the response time of a task[8] depends on factors that cannot be completely foreseen at design time. In this case, probabilistic guarantees can be assumed [Stankovic & Ramamritham 1990]. Two techniques are commonly used to ensure predictability:

- **Schedulability analysis**, in which given a set of tasks, their priorities, their timing constraints and their worst case execution time (WCET), it is possible to find a schedule which satisfies all the constraints for all present tasks. Many scheduling analysis algorithms can be applied for real-time systems. Some examples are the Rate-monotonic analysis (RMA), Earliest Deadline First analysis (EDFA) and static order scheduling [Liu & Layland 1973, Davis & Burns 2011];

- And **formal verification**, in which the whole system and its properties are formalized into logical statements and timing constraints are verified through model checking or theorem proving techniques [Wang *et al.* 2010, Bernstein & Harter 1981].

However, both solutions may not be enough to design a predictable real-time systems [Huang *et al.* 2005].

[7] *Release time* is the time at which the instance of a scheduled task is ready to be executed [Laplante & Ovaska 2011]. A release is thus the execution of a given scheduled task.
[8] *Response time* is the time between the presentation of a set of inputs to the system and the realization of the required functionality [Laplante & Ovaska 2011]

2.3.1.2 Determinism in real-time systems

Determinism in real-time systems is the ability of ensuring the execution of an application despite presence of external factors that could unpredictably cause disturbance. Determinism is closely related to predictability, since perturbations may alter the functionality and response time of a real-time application, invalidating predictability analysis. Thus, determinism is necessary ,so that all application deadlines can be met and predictability can be achieved.
Two metrics are related to determinism:

- **Latency**, that is, the time between an event and a system response to that event (*e.g.* the time between the instant a task was supposed to be released and the time it is released in fact). Developers usually focus on minimizing latency, but in real-time systems it is more important to normalize it, in a such way that system latencyis known and predictable. In order to minimize latency, it is important to find all sources of latency of a system and measure the impact of each source in final latency. For known event processing times, latency can be found by measuring response time (which will be the sum of the known processing time and the latency itself);

- and **Jitter**, which defines the unsteadiness of system latency. Jitter can be measured through the distribution and standard deviation of response times. A common way to define jitter is as the difference between the highest and the lowest response time of a system. In real-time systems, reducing jitter is sometimes more important than deadline enforcement. Jitter may also be known as real-time punctuality [Laplante & Ovaska 2011].

2.3.2 Infrastructure of Real-time Systems

As said before, in order to minimize latency and achieve predictability, all aspects of a system must be taken into account and mastered during the design of a real-time system. Initially, real-time systems were designed for and implemented in dedicated hardware. Today, due to advances in computer hardware, even general purpose hardware can be used to execute real-time systems. Most real-time concerns are concentrated in the software layer of the system. The next subsections present an overview of the software technologies available for real-time systems, regarding operating systems, middleware and programming languages.

2.3.2.1 Real-time Operating Systems

Real-time operating systems (RTOS) are operating systems that support applications with timing constraints [Stankovic & Rajkumar 2004]. Thus, besides providing access to low level and basic functions, they must provide determinism. Some features found in traditional operating systems are not important for RTOSs. However, features related to predictability and dependability, such as predictable interrupt handling and scheduling, are required. For firm and soft real-time applications, enhancing conventional operating systems (just as real-time and low latency kernel versions for Linux distributions) may be enough for achieving real-time behaviour; however, for safety-critical and other hard real-time applications, certified RTOS is mandatory [Kopetz 2011, Stallings 2011] . Domain-specific and higher-level functionality should be implemented by the middleware running on top of the RTOS.
The IEEE Portable Operating System Interface for Computer Environments (POSIX) 1.b standard specifies a list of basic services required by a RTOS [IEEE & Electronics Engineers 1993], such as asynchronous and synchronous input/output, memory lock-

ing, semaphores, shared memory, execution scheduling, timers, interprocess communication (IPC) and real-time threads. It is also common to find deterministic synchronization mechanisms, real-time priority levels, dynamic deadline identification and predefined latencies for task switching and interrupt mechanisms [Baskiyar & Meghanathan 2005].

2.3.2.2 Real-time Middleware

Middleware is a software layer that lays between applications and operating systems or hardware, isolating the application from domain or platform specific differences. Middleware provides several types of abstraction and transparency to applications, such as host infrastructure, distribution, common services and domain specific services [Emmerich 2000].
Today's real-time systems are increasingly being connected via networks to create distributed real-time systems [Schmidt 2002]. Along with the increasing software complexity, this factor has led to the development of the first middleware for real-time systems. By using middleware, the development of applications can be simplified, resources can be managed effortlessly and interoperability of heterogeneous environments and technologies can be achieved more easily, reducing system life-cycle costs and code errors [Wang *et al.* 2004]. In addition, in the particular case of real-time systems, it is important for middleware to support and respect non-functional properties, such as performance and feasibility.
A popular framework for real-time systems is **RT-CORBA** [Schmidt & Kuhns 2000], which adds real-time support for CORBA specification[9]. CORBA in its standard form provides distribution transparency, automating tasks like connection management, object marshalling, unmarshalling and demultiplexing, technology independence and fault-tolerance. RT-CORBA adds quality of service (QoS) control, resource management and priority inversion control in order to improve applications' predictability. TAO [ISIS a] is an implementation of the RT-CORBA specification with component and deployment/configuration support extensions (CIAO [ISIS b] and DAnCE [Deng *et al.* 2005], respectively). Other real-time ORBs are MC-ORB^2 [Zhang *et al.* 2009], nORB [Subramonian *et al.* 2004] and the already extinct ROFES [Lankes *et al.* 2001]. Another implementation, RT-Zen [Raman *et al.* 2005], integrates Real-time Specification for Java (RTSJ) [Bollella & Gosling 2000] features.
The **Data Distribution Services** (DDS) [Object Management Group 2007] middleware specification offers a real-time message-based communication. Just as CORBA and RT-CORBA, DDS is a specification published and maintained by the Object Management Group (OMG), but instead of focusing on objects and following a RPC-based paradigm, DDS follows a data-centric publish/subscribe paradigm[10]. DDS is technology-independent, flexible, robust, scalable and supports the control of QoS properties that affect predictability, overhead and resource usage. There are several implementations for the DDS standard, both commercial (developed by vendors such as Sparx [11], PrismTech [12], Twin Oaks [13] and MilSOFT [14]) and open source (namely OpenSplice and OpenDDS).
The **Quality Objects** (QuO) [15] framework is an adaptive layer which supports quality

[9]CORBA stands for "Common Object Request Broker Architecture", and the RT- prefix means "Real-Time". CORBA is a standard for Object Request Brokers, also known as ORBs. An ORB is a middleware enabling remote procedure calls (RPC) and interoperability in object-oriented distributed systems.
[10]In the publish/subscribe message pattern [Birman & Joseph 1987], there are two main roles: publishers and subscribers. Publishers may publish categorized messages to a broker, whereas receivers receive messages of a particular category from this broker.
[11]Available at `http://www.sparxsystems.com.au/dds`.
[12]Available at `http://www.prismtech.com/opensplice`.
[13]Available at `http://www.twinoakscomputing.com/coredx`.
[14]Available at `http://dds.milsoft.com.tr/`.
[15]Available at `http://quo.bbn.com/`.

of service attributes on the standard CORBA platform and Java RMI. QoS attributes are defined separately from the objects they are used on, enabling interoperability and their reuse in several objects at the same time. These attributes can be dynamically adapted as well. QuO uses aspect-oriented software development techniques and provides a complete framework for developing applications.

2.3.2.3 Real-time Programming Languages

In order to enable the use of a programming language to develop real-time applications, it is important to guarantee predictable, reliable and timely operation. Every activity must be expressed in a language by means of time-bounded constructs, enabling the analysis of the system timing constraints. Therefore, three approaches can be used to add support to the expression of timing constraints and deterministic behaviour in programming languages:

- Eliminating constructs which have indeterminate execution times;
- Extending existing languages with constructs to express real-time behaviours; and
- Constructing a language jointly with an operating system, allowing direct access to low-level (and more predictable) facilities.

Other desirable features for real-time programming languages are process definition and synchronization, interfaces to hardware access, real-time interrupt handling, error handling, strong typing and modularity [Stoyenko 1992].
Three types of general-purpose languages are commonly used to develop real-time systems:

- Assembly languages, which provide direct access to hardware and low overhead compared to higher-level languages, in exchange of lack of portability, modularity and other higher-level abstractions;
- Procedural languages, such as C, FORTRAN 95, and Ada 95, or subsets of these languages targeting safety-critical systems (like Spark-Ada and MISRA-C), offering strong typing, abstract data typing, exception handling and modularity features; and
- Object-oriented languages, like C++ (and its subset EC++, for 'Embedded C++') and real-time extensions for Java (based or not in the RTSJ [Bollella & Gosling 2000] or SCJ [Schoeberl *et al.* 2007] specifications)[16] which benefit from procedural languages' advantages and add higher-level programming abstractions, increasing developers' efficiency and code reuse, but potentially introducing sources of unpredictability and overhead.

In addition, many highly-specialized or research-only languages for real-time applications were also created along the last 40 years. These include Eiffel, Pearl, LUSTRE, MACH, MARUTI and ESTEREEL, among others [Laplante & Ovaska 2011, Kaisler 2002, Cooling 2003].

2.3.3 Design and Development Techniques for Real-time Systems

Just as in general-purpose software systems, the development process of real-time systems consists of the following steps:

[16] Real-time Java extensions and technologies are extensively described in [Higuera-Toledano & Wellings 2012].

- *Requirements analysis and specification*, that is, to identify and describe what the software is supposed to do in order to satisfy the customer's requirements;
- *Architectural design*, in which the software structure is modelled in terms of essential software components and how these components communicate with each other;
- *Physical design*, that is, the mapping of the architectural design onto the hardware platform. This is a critical activity in real-time systems, since they tend to be very complex and a mismapping may lead the system to violate its timing constraints;
- *Implementation*, when the system and program structures are translated to source code; and
- *Test, integration and debug*, in which we try to show that the system works as expected by means of testing individual and combined system modules.

Different development techniques may cover different steps of this development process. They can be combined in order to obtain a more efficient development process. The next sections describe four types of development techniques: Formal methods, Structured methods, Object-oriented methods and Model-Based Engineering methods.

2.3.3.1 Formal Methods

Formal methods is the term used to designate the use of mathematical and logical techniques to express and analyse properties in one of the computer system (hardware, software or both) development steps. Formal methods are most applied then to the specification and analysis of requirements (formal specification in a language with precise syntax and semantics), architectural design, physical design (formal modelling of the software and hardware properties) and test phases (formal analysis and verification based on discrete mathematics and logic techniques, like propositional logic, set theory and predicate calculus). Formal methods can be applied as well to the implementation step, by using a formally defined programming language [Kopetz 2011].
By means of formal methods, one may prove formally that a software module implements correctly a given information. Formal specification and modelling require human involvement, whereas formal analysis can be mechanized. But since analysis uses results from specification and modelling, using formal methods is sometimes error-prone. In addition, they are considered difficult to use and often believed to increase project costs and delays. Due to their correctness and safety guarantees, use of formal methods is highly esteemed in the certification of safety-critical applications. It is not sure though that the correctness of a software according to a specification will lead to correctness in the real world, because proving a specification does not 'prove' a software product itself. It is known as well that, just as software testing, formal verification can not prove the absence of bugs, just their presence [Laplante & Ovaska 2011]. Techniques such as model checking and consistency checking may be used to perform analysis on formal specifications.
Finite state machines (FSM) [Hopcroft 2007] are commonly used to formally model systems. Basically, this technique relies on the fact that a system can be represented by a finite number of unique states. Transitions between states may depend on factors such as time or the occurrence of specific events. FSMs may be represented by diagrams, sets and matrix. Since it uses a formal mathematical description, FSMs are unambiguous. Concurrency may be expressed by using several state machines or by using non-deterministic transitions. Other benefits of using FSM include easy code generation, optimization through state reduction techniques and a rich theory that can be exploited during the development

of systems. It is not possible though to describe the internal aspects of each module/state and to factorize the system onto functions and sub-functions.
Statecharts [Harel 1987] improve FSMs by combining it with data-flow diagrams and enabling the representation of synchronous and asynchronous communication. In statecharts, states may have their own FSM. In addition, concurrency can be more easily depicted by the explicit description of concurrent states, called AND-states. Communication between states is refined and can be achieved by means of two mechanisms: global memory (asynchronous) and broadcast communication (synchronous). The use of Statecharts can be incorporated to other types of methods, such as structured or object-oriented methods.
Petri nets [Murata 1989] are another common formal method to specify operations in real-time software systems [Bucci & Vicario 1995]. In Petri nets, circular boxes (called *places*) represent data stores or processes. Rectangular boxes correspond to transitions or operations over data. These entities are connected by unidirectional arrows. Places may be labelled with data counts, whereas transitions can be labelled with transition functions. A transition is fired when it has as many inputs as required for its output. Petri nets can model flowchart constructs and constitute a very powerful formal method for representing dynamic multiprocessing systems. Extended types of Petri nets were proposed to particularly deal with timing issues, such as timed Petri nets [Zuberek 1991], time Petri nets [Merlin & Farber 1976] and stochastic timed Petri nets [Florin *et al.* 1991]. Formal proving techniques can be applied to Petri nets. However, the task of decomposing a complete application into Petri nets may be very costly [Laplante & Ovaska 2011].

2.3.3.2 Structured Methods

Structured analysis and design methods are widely used in real-time systems. This technique has appeared along with the evolution of the procedural programming languages. Consequently, they are based on the notion of a data flow in which successive transformations are performed, so that the structure of the problem matches the high-level structure of the software solution. Its analysis phase focuses on structuring the system's context (interfaces between the system and other entities), processes (functional specification) and content (data used by the system). During the design phase, it is common to use procedural-based functional decomposition techniques, such as Parnas partitioning [Parnas 1978].
The *de-facto* standard for structured methods is the **Yourdon's Structured Method (YSM)** [Yourdon 1989]. This method was proposed in the mid 1970s and ever since several extensions and derived methods have been proposed, the most famous being the Ward-Mellor [Ward & Mellor 1991] methodology, widely used in the design of real-time embedded systems. In the Yourdon model, the information produced in the analysis step (called *essential model*) is used as output for the design step. This information is composed of two parts: an *environmental model* (*i.e.* the external view of the system, containing a description of the complete system, its decoupling into communicating parts, the relationship between different parts and how parts communicate) and a *behavioural model* (that is, how the system is supposed to behave, including system functions, how the behaviour may change over time and the information exchanged within the system). Both models are used in the design step to create the *implementation model*, which is going to define how the system can be implemented in a given technology. The implementation model is composed of three distinct parts: the *processor environment model*, in which system partitions are allocated to processors; the *software environment model*, which defines tasks, data sections and communications for each process; and the *code organization model*, which, as the name suggests, defines the code structure of each task.
Yourdon model has been updated in 2003 by Wieringa [Wieringa 2003] as the **Post-modern**

Structured Analysis, which incorporates elements from both Jackson System Development[17] [Jackson 1983] and object-oriented methods. In summary, it refines the environmental and behavioural models, enabling a more refined description of the context and of the desired system behaviour.
Statemate [Harel & Politi 1998] is a tool created in order to help in the requirement specification of real-time systems. The method implemented by this tool combines statecharts and activity charts (which are syntactic variations of data flow diagrams), thus being considered as a formal and structured method. The combination of both models enabled the expression of parallelism, timed and event-based transitions between activities and activity hierarchy. The tool could also execute the designed program following a semantically well-defined execution algorithm.
Many other structured design methods target real-time systems: MASCOT, MOON, DARTS, MCSE, EPOS, MARS, RTSA, PAMELA, etc. [Burns & Wellings 1994,Burns & Wellings 2001].
According to Gomaa in [Gomaa 1994], there are four important objectives that must be achieved in a real-time design method:

- To support structuring the system as a set of concurrent tasks;
- To support components reuse through information hiding;
- To support the use of finite state machines for defining behavioural aspects;
- To support the analysis of the system design to validate the respect of its real-time properties.

Finally, **CoRE** [Faulk *et al.* 1992] is a viewpoint-based methodology which is used only for requirement modelling and analysis. Viewpoints help to see the main problems from particular points in order to describe its nature and content. Each viewpoint describes input information, actions necessary to process this information and generated results. Information necessary to identify and build viewpoints is collected by an analyst; then, after building the viewpoint-based model of the problem, the analyst can identify conflicts and inconsistencies in both software and hardware requirements by combining information. CoRE has been applied to the analysis of European avionic and defence projects.

2.3.3.3 Object-oriented Analysis and Design

While structured methods favour the Von Neumann architecture, in which there is a separation between data storage and processing, the object-oriented design, just as the whole object-oriented paradigm, favour the encapsulation of both data storage and data processing in entities called *objects*. Objects communicate with each other, processing inputs and generating outputs to their neighbours. Using objects may increase maintainability, understandability, modularity, reuse and extensibility in real-time systems.
The object-oriented design method is based on the production of abstract models of the system under design. It is often composed of the following steps [Deacon 2005, Laplante & Ovaska 2011, Williams 2005]:

[17] The Jackson System Development (JSD) is a software development methodology developed by Michael Jackson in the 1980s. This method consists of three phases: *modelling* of real world events, entities, roles and their properties; design of the system processes as a process *network*, in which data collection, process and generation processes are added to the basic model processes; and the *implementation* phase, in which implementation-related aspects such as timing, scheduling and database are considered. JSD can be seen as an extension to the Jackson Structured Programming (JSP) method [Jackson 1975], handling more complex process compositions and including system specification and implementation stages.

- **Analysis phase**: It includes the *requirements analysis*, the *systems analysis* and a high level version of the *structural and behavioural object model* of the system. The requirements analysis is performed by defining the system usage through use cases and real execution scenarios. In the systems analysis, outline aspects of both system hardware and software are defined, such as the overall architecture of the system. The preliminary object models are used to represent each subsystem as a set of collaborating objects and may be used to evaluate aspects related to their timing constraints, concurrency and external interfaces;

- **Design phase**: In the design phase, the architecture, the communication and the object types (called *classes*) identified in the preliminary object model are refined. In the *detailed architecture design*, it is necessary to model concurrency in the system by identifying the potential active objects[18] (which might correspond to real-time tasks), mapping the software components onto the available hardware (deployment model) and refining the component decoupling. In the *detailed mechanistic design*, the designer must model the objects' collaboration and the sequence of messages that are exchanged. Classes are then detailed, by adding fields (called *attributes*) and internal data processing procedures (called *methods*). New classes can be defined if necessary, and they can be organized onto modules (called *packages*) according to their category or functionality;

- **Source code translation phase**: The model information is translated (manually or automatically) to source code which is then compiled to generate an executable application. Generating code from models is an important part of the Model-driven Engineering (MDE) approach [Bézivin 2005];

- **Testing, Integration and Validation phase**: In this phase, corrections are made to the code generated by the model. Defects can be found in the design models and tests are performed in separate and combined objects so that errors can be detected and corrected.

A special attention must be given to the affectation of objects to tasks. Too many tasks may lead to many context switching and scheduling problems, whereas too few may lead to timing constraints violations due to the excess of operations performed by the present tasks [Williams 2005].
UML[19] diagrams are often used to model object-oriented systems. A given type of UML diagram can be used in different stages of the development process: the difference lies in the detail level of the information that is present in the diagram.
There are several *design patterns* available for the development of object-oriented systems. These patterns provide proven solutions to common design problems. There are patterns for mapping and creating objects, for building and interacting with complex structures, for managing communication between objects and other systems, etc. [Gamma *et al.* 1995, Schmidt *et al.* 2000]. Some design patterns were developed specifically for real-time systems [Douglass 2002].
Time and timing constraints were introduced in UML by an extension profile called MARTE[20].

[18] The object-oriented paradigm follows a client-server architecture style. We call an *active object* the object which performs calls (or *execute a method*, as said in the object-oriented approach) on other objects - called *passive objects*.

[19] UML [Object Management Group 2006c] stands for Unified Modelling Language. It is a standardized modelling language with graphic notations used to create object-oriented software systems. UML is currently in its 2.4.1 version and is managed by the Object Management Group (OMG).

[20] MARTE stands for "Modelling and Analysis of Real-Time and Embedded systems".

Both software and hardware timing properties can be modelled with UML-MARTE. The fundamental unit of behaviour in the UML-MARTE specification is the *action*, that transforms set of inputs into a set of outputs and takes a given duration. These actions can be initiated by *triggers*. Three types of time can be modelled in UML-MARTE: *logical time*, in which time is measured by the temporal order of causal events, without a specific metric; *discrete time*, where time is partitioned into a set of ordered granules, during which actions may be performed; and *real time*, where physical time progression is precisely modelled [Object Management Group 2008, Laplante & Ovaska 2011, Mallet & de Simone 2008]. The UML-MARTE specification will be further detailed in the Chapter 4.

2.4 Adaptiveness and Dynamism in Real-time Systems

The timing constraints in real-time systems come from the interaction and the impact those systems have with entities in the real world. However, the real world may be extremely dynamic. Entities in the real world may appear and disappear, combine and separate, replace and be replaced, definitely or temporarily. Thus, it is important for real-time systems to be able to adapt themselves to these real world changes, which may not be possible to specify during the system design phase. At the same time, real-time systems must ensure predictable behaviour under no matter what operation conditions. The need for adaptive real-time systems comes from two types of applications: adaptive systems whose interactions must be time-bounded, and real-time systems needing flexibility at run-time [Bihari & Schwan 1991]. In this section, we will discuss some software architectural styles developed to overcome the conflict between predictability and dynamism in adaptive real-time systems. Here, we will focus on two of the main styles used to develop flexible real-time systems: real-time component- and real-time service-based approaches.

2.4.1 Real-time Component Models

Re-usability and modularity were the keys to success of the component-based approach. With the use of component frameworks, developers can focus on business logic parts of the system. Non-functional concerns of the code can be delegated to the framework, which can automatically generate a code, shortening time-to-market, decreasing development costs and improving developers' efficiency [Szyperski 2002]. It is also important to mention its support to compose potentially heterogeneous and distributed components, which is a relevant advantage for modern applications.
The popularity of the component-based software engineering has led to its adoption in different application domains. As mentioned before, in the beginning, real-time systems were developed to a specific hardware and platform, compromising portability, integration with other systems, increasing development and maintenance costs due to the use of low-level and platform-specific statements. These difficulties have encouraged the adoption of the component-based approach in the design and the development of real-time systems. Consequently, in recent years, several component models targeting real-time and embedded systems were developed.
It is expected for a real-time component model to satisfy the following requirements:

- These component models should be compatible with the expression and satisfaction of real-time requirements. This means that the services provided by components must have a predictable behaviour, in order to assure determinism in their execution time. To achieve a predictable behaviour, it is not enough to characterize the execution time

of components services; features such as resource constraints, resource reservation, dependability, concurrency and synchronization play an important role.

- In the component-based approach, all application entities are modelled as opaque components or components of components. On the other hand, in real-time applications, applications are seen as a set of tasks which may be scheduled according to their real-time (*e.g.* deadline, latency) and/or scheduling (*e.g.* priority, period) parameters. It is important first, to provide a mapping from the code contained within components to schedulable real-time threads and second, to be able to establish this mapping without any information concerning the internal details of components.
- Real-time component models should be able to support distribution, since both real-time and component-based approaches are used to implement distributed systems.
- Real-time components should be able to be composed to provide a more complex functionality; this composition should not interfere in the real-time behaviour of the resulting component and that of the whole application.

The main modelling languages for real-time systems support the expression of components. Most of them offer a pure modelling approach though, without an underlying execution framework. **AADL**[21] (Architecture Analysis and Design Language) [Feiler *et al.* 2006] is a standard language for modelling real-time embedded systems in avionics domain. AADL allows modelling software and hardware aspects of the application and the early analysis of their functional and non-functional properties. Semantics of AADL are objects of several studies in the computer science literature [Yang *et al.* 2009, Ölveczky *et al.* 2010]. Components can be specified by means of AADL component assembly model, in which components can be connected with each other and their implementation instances can be referenced.
SysML (System Modelling Language) [Friedenthal *et al.* 2008] is a result of an effort from OMG in unifying modelling languages. Just as MARTE, it is a profile of UML2.0. It enables the description of behavioural structures and the assignment of these structures to the system architecture. Components can be described in MARTE and SysML by means of UML2.0 component diagrams.
BIP [Basu *et al.* 2006] is a modelling language and component framework with formal semantics and mathematically proven properties. The name BIP is derived from the name of the three layers that compose the model: the Behaviour layer, where behaviour of individual components can be represented by transition systems; the Interaction layer, where connectors between behaviour transitions can be defined; and the Priority layer, which includes policies (*e.g.* scheduling policies) that will be applied to transitions interactions. Composing components in BIP means composing their three layers using well-defined rules and parameters. BIP contains a set of tools which can be used for code generation and analysis.
As discussed previously, the CORBA Component Model (CCM) is a platform-independent component model developed for the CORBA framework. Several works have extended this component model specification to target safety-critical and real-time applications. The Lightweight CORBA Component Model (**LwCCM**) [Object Management Group 2006a] offers a simplified version of CCM based on the RT-CORBA specification, eliminating dynamic mechanisms (such as introspection) and other features that may lead to performance issues. López Martínez *et al.* [Martínez *et al.* 2013] extended the programming model and the interface specification of LwCCM, removed the dependency on CORBA from the communication layer and created the Real-time Container Component Model (**RT-CCM**). As

[21] AADL will be further discussed in the Chapter 4.

most of the models, RT-CCM composes the real-time model of each participating component in order to define the real-time model of the complete application. RT-CCM components may be passive or active and interact by means of required and provided ports. Active components are components that can react to external or timed events. These reactions are triggered internally. RT-CCM concentrates all real-time design, composition and configuration concerns at a container level, so that these mechanisms are invisible to the business code developer. A container-based approach also improves code reuse. RT-CCM relies on CBS-MAST [Lopez *et al.* 2006], an extension of the MAST methodology [Harbour *et al.* 2001], which is used to formulate components' real-time model. In addition, RT-CCM integrates RT-D&C [Martinez *et al.* 2010], an extension to OMG's '*Deployment and Configuration of Component-based Distributed Applications*' specification (D&C) for real-time systems.
UM-RTCOM [Díaz *et al.* 2008] is another component model based on the RT-CORBA specification. Its approach is similar to RT-CCM: individual components' behaviour can be formulated in SDL-RT (a real-time extension of the Specification and Description Language - SDL) [Alvarez *et al.* 2003] and then schedulability analysis is applied to the final model resulting from the composition. **MyCCM - High Integrity** [22] is a project based on LwCCM in which the application model is mapped onto AADL for analysis and code generation.
ACM (ARINC-653 Component Model) [Dubey *et al.* 2011] is a component framework combining CCM with the static memory allocation and isolation mechanisms specified in the ARINC-653[23] platform standard (*i.e.* temporal and spatial isolation) [Aeronautical Radio, INC 1997]. Like RT-CCM, ACM allows the definition of component triggers that are executed in order to activate components. Deadlines are applied to both end-to-end compositions and individual component services.
Despite the name similarity, **SaveCCM** [Hansson *et al.* 2004] (which means 'SAVEComp Component Model') is not based on CCM. SaveCCM (and its implementation for vehicular systems, **SaveCCT** [Åkerholm *et al.* 2007]) was developed in the context of a project whose goal was to establish a methodology for the development of component-based applications for safety-critical and embedded systems. With SaveCCM it is possible to model the behaviour of each component through timed automata. The real-time model is analysed by means of the UPPAAL-PORT [Håkansson *et al.* 2008] tool. After the analysis, executable entities are generated and deployed in the execution environment. SaveCCM focuses on control systems; thus its components were conceived in a way to only implement passive functions. Applications are constituted by triggering calls towards sequential component functions. This low-level granularity approach was also adopted by the **PIN** [Hissam 2005], **PECOS** and **COMDES** [Ke *et al.* 2007] component models. It is often associated with port-based interface component models. **ProCom** [Sentilles *et al.* 2008] is an extension of the SaveCCM model supporting higher-level components and hierarchical composition. SaveCCM was inspired by **Rubus-CM** [Hanninen *et al.* 2008], a component model developed by Arcticus Systems AB and industrially used in hard real-time systems.
Koala [van Ommering *et al.* 2000] is another industrial component model. It was conceived by Philips. The introduction of real-time features into Koala led to the creation of the component model **ROBOCOP** [Muskens *et al.* 2005]. ROBOCOP supports predictability by the same means as RT-CCM and UM-RTCOM: composition of isolated component behaviours into a final application real-time model. Bosch has developed **BlueArX** [Kim *et al.* 2009], a hierarchical view-based component model targeting embedded applications

[22] Available at `http://sourceforge.net/apps/trac/myccm-hi/wiki`.
[23] ARINC-653 is a software specification created in 1996 which defines an API and principles for space and time partitioning in safety-critical avionics real-time operating systems.

in the automotive domain. **AUTOSAR** [24] is another component model used in the automotive domain; in fact, it is considered as the standard in this domain, backed by a consortium of companies. Real-time properties support is not clearly specified, although it runs on real-time operating systems [Hošek *et al.* 2010]. This gap has been filled by works on the extension of the AUTOSAR technology [Becker *et al.* 2010]. AUTOSAR defines a complete development process.

In the context of Java technology, some component models have been developed by incorporating features of the Real-time Specification for Java (RTSJ) [Etienne *et al.* 2006, Hu *et al.* 2007, Plšek *et al.* 2012]. This was motivated by the complexity incurred by the use of some RTSJ constructs (such as the different types of memory areas) and the lack of modularity between business code and RTSJ concerns in real-time Java applications. These models offer access to RTSJ facilities, such as memory management, real-time threading, asynchronous event handling and asynchronous transfer of control. A component model combining RTSJ and EJB over a real-time RMI communication has also been proposed by [Wang *et al.* 2005].

The most popular approach to implement run-time adaptation in real-time component-based systems (and in real-time systems in general) is the use of *modes*. A mode is a pre-configured architecture which can be switched at well-defined points, called *mode transitions*. A *mode transition protocol* is used to specify how this change occurs (*i.e.* which parts of the architecture remain, which ones are replaced and which parts are used to replace them).

Another possible approach is the use of *components freezing*, that is, stopping the whole activity of the components before realizing run-time adaptation. This approach was first suggested by Kramer and Magee in [Kramer & Magee 1990]. Wermelinger [Wermelinger 1997] has improved the algorithm to consider only the involved components in the freezing, minimizing interruption time. Rasche and Polsche [Rasche & Polze 2005] presented a related technique in which the whole application is blocked during a bound time, disabling dynamic reconfiguration during this time interval. Américo *et al.* [Américo *et al.* 2012] used a similar technique to enable the use of service-oriented component models in real-time applications: real-time tasks demand the platform to block dynamic reconfigurations during the execution of critical tasks, and indicate the end of these tasks. If several tasks make a request at the same time, the platform is only unblocked after receiving termination signals for all tasks. A time-out can be established in order to avoid the platform to remain indefinitely blocked.

The use of incremental reconfiguration protocol has been proven as being robust by Boyer *et al.* in [Boyer *et al.* 2013]. This protocol, which is based on the concept of a contractual reconfiguration based on an incremental sequences of operations, was mathematically formalized and proven correct. In addition, the algorithm is fault tolerant and its complexity is linear with respect to that of the target reconfiguration.

Bozga *et al.* developed in [Bozga *et al.* 2002] a toolbox to validate component-based real-time applications. In their work, components were seen as processes that could be created and deleted dynamically. These processes were described in a dynamic version of extended timed automata. Their semantics were similar to those of timed automata and the system analysis was based on state-space exploration techniques.

Regarding Java applications, solutions for run-time adaptation also include the use of mechanisms such as concurrent class-loaders [Pfeffer & Ungerer 2004] and agents [Brennan *et al.* 2002].

[24] AUTOSAR is an acronym for for '*Automotive System Open Architecture*'. Available at `http://www.autosar.org/`.

2.4.2 Real-time Service-Oriented Architectures

Real-time Service-Oriented Architectures (RT-SOA) is an extension to the Service-Oriented Architecture (SOA) approach aiming to include timing properties in aspects such as the modelling, composition, orchestration, deployment, policy, enforcement and management of services [Tsai *et al.* 2006]. Research works in RT-SOA have originated from the need of enterprises who have adopted the service-based approach in their systems, but suffer from the lack of predictability in the current solutions. The RT-SOA approach has already been applied to the medical, military, financial and industrial automation domains in several projects [Bohn *et al.* 2006, McGregor & Eklund 2008, Cucinotta *et al.* 2009, Panahi *et al.* 2010, Moreland 2013].

RT-SOA systems inherit the need for time-bound requests from real-time systems. Thus, it is necessary to know the timing behaviours of each service implementation; the composition of these behaviours will result in the characterization of the composite services. This information, along with service consumer timing requirements, must be taken into account during service selection and to perform end-to-end timing analysis.

Several works have proposed and investigated techniques for specifying timing behaviours in **service interfaces**. Using ontologies, such as OWL-S, may ease semantic analysis and automatize service composition, despite its lack of mechanisms to specify timing properties and constraints [Papaioannou *et al.* 2006, Moussa *et al.* 2010]. Another alternative is the use of languages dedicated to the specification of QoS attributes, such as **QML**[25] [Jin & Nahrstedt 2004, Becker 2008]. Other propositions focus on the use of new service definition languages based on formal methods, such as SCC [Boreale *et al.* 2006], CaSPiS [Bruni 2009], COWS [Lapadula *et al.* 2007] and SOCK [Guidi *et al.* 2006].

However, in the industry the most used approach remains the XML-like languages. This approach includes QoS extensions to service description standards (such as **WSDL**[26] [Al-Ali *et al.* 2002, D'Ambrogio 2006, Dai & Wang 2010] and **USDL**[27] [Marienfeld *et al.* 2012, Aniketos Project Consortium 2011]) and SLA[28] languages [Cucinotta *et al.* 2009, Kübert *et al.* 2011, Américo *et al.* 2012]. The design of QoS languages and the use of distinct QoS description and negotiation terms result from the fact that inserting QoS data directly in the service interface may limit the reuse of both QoS attributes and service interface.

Despite the recentness of the subject (the first openly considering real-time aspects in service-oriented architectures dates back from early 2000s), many other aspects of real-time service-oriented applications have been studied in the last years:

- Two main approaches have been proposed to validate and verify **service compositions**. First, the creation of new service composition languages, whose semantics is based on formal methods, like process calculus [Benghazi *et al.* 2010]. These languages may be combined with other languages for design (*e.g.* graphical languages) and execution (*e.g.* executable languages). Second, mapping composition languages (such as BPEL and WS-CDL) to formal methods, such as timed automata and rewriting logic [Kazhamiakin *et al.* 2006, Dong *et al.* 2006, Bruni *et al.* 2006, Al-Turki & Meseguer 2007, Wehrman *et al.* 2008, Martínez *et al.* 2009a, Tan *et al.* 2011, Stachtiari

[25] QML (*QoS Modelling Language*) [Frølund & Koistinen 1998] is a QoS specification language designed by HP Labs in late 90s.

[26] WSDL (Web Services Description Language) [Christensen *et al.* 2001] is a XML-based language used for defining web-services and their functionality.

[27] USDL (Unified Service Description Language) [Oberle *et al.* 2013] is an extensible service description language designed to describe technical and business services. USDL is being developed and incubated at W3C (more information available at `http://www.w3.org/2005/Incubator/usdl/`.

[28] SLA (*Service Level Agreement*) [Verma 1999] is a contract document defining the QoS terms and requirements between a service provider and a service consumer.

et al. 2012];

- **Service selection** was deeply studied by Jaeger and Mühl in [Jaeger & Mühl 2006]. Since service selection with multiple criteria can be seen as a combinatorial problem [Jaeger *et al.* 2005], they describe heuristics that can be used in the dynamic selection of services to service compositions. Anastasi *et al.* proposed in [Anastasi *et al.* 2011] a **service registry** for service-oriented real-time systems, coping with the issues raised by the dynamism incurred by this architectural style. Their registry enables dynamic QoS data gathering, future prediction based on collected data and permanent data storage. Christos *et al.* [Christos *et al.* 2009] worked on a service registry for dynamic Web Services, where it is possible to specify a QoS execution policy and to automatically handle exceptions according to the QoS policy activated in the system. Similarly, Heam *et al.* proposed a set of techniques for QoS aware verification of web services substitutability [Heam *et al.* 2007]. Dynamic reconfiguration and service re-selection were also studied by Zhai *et al.* in [Zhai *et al.* 2009]. In their work, they have used an iterative inspection algorithm to reconfigure failed services without violating QoS constraints and applied it to a research service-oriented framework called *Llama*;

- There are some works exclusively consecrated to the **timing analysis** of dynamic real-time service-based systems. For instance, Calinescu *et al.* applied the quantitative verification approach [Kwiatkowska 2007] to service-based systems in [Calinescu *et al.* 2012]. They assume that if the specification and domain assumption statements (which in their work are modelled as discrete time Markov chains) are satisfied, then the system requirements (modelled as a probabilistic computation logic tree) are met. In their turn, Aminpour *et al.* have described in [Aminpour *et al.* 2011] how to model service-oriented architecture systems along with their non-functional properties in AADL, so that the complete application architecture may be analysed and simulated with AADL tools.

Several works describe complete real-time service middleware solutions. The framework developed in the context of the European Union-funded project IRMOS[29] [Kyriazis *et al.* 2010] targeted soft real-time applications hosted in the Cloud[30]. In their framework, components and services can be modelled in UML2. The developer is responsible for providing application, component and work-flow descriptors. From these data, an SLA is automatically produced. The descriptors of different components and the hardware configurations are used as input for a service that outputs the resulting QoS of the system. The resulting QoS in turn is used to estimate the application's behaviour by means of formal methods such as finite state machines. At run-time, the SLA terms are automatically negotiated and the virtual and physical resources are reserved by a deployment manager. The execution of the different work-flows is monitored, and in case of SLA terms' violation, a resource adaptation and SLA re-negotiation processes are triggered.
The framework developed by Samaras *et al.* [Samaras *et al.* 2010] targets Wireless Sensor Network (WSN) applications. It aimed to decrease the overhead incurred from connection establishments and continuous message exchanges. The solution adopted was mapping WSN to IP networks and compressing messages' contents, keeping SOA standards and using SOA technologies to enable interoperability. Similarly, the SIRENA project [Bohn *et al.* 2006] was also focused on embedded devices and relied on Web Services standard

[29] More information at `http://www.irmosproject.eu`.

[30] Cloud computing is a term that designates distributed systems with a large number of computers over a network. It enables ubiquitous and on-demand access to shared resources (*resource pooling*) that are provisioned without delays (*elasticity*), with minimal interaction from the service provider (*self-service*) and billed accordingly to resource consumption (*measured service*) [Hayes 2008].

protocols to create real-time service-oriented applications. SIRENA was the base of the SODA project[31], and the results of both were used in the SOCRADES project. Besides timing constraints, their platform had additional requirements such as energy constraints and plug and play reconfiguration. Service interfaces were described in DPWS. Service logic was modelled as Petri nets and verified by the platform. The different aspects of the SOCRADES framework are managed by the different *bots* available in the platform [Mendes *et al.* 2009].

The RT-Llama framework [Panahi *et al.* 2010, Panahi *et al.* 2011] addressed the execution support of real-time work-flows in SOA platforms. This is achieved through an execution reservation mechanism based on the use of virtual CPUs. In addition, the framework is able to perform a deadline-based work-flow composition to provide end-to-end predictability. A prototype was developed in real-time Java.

With regard to real-time Java applications, some works have proposed real-time extensions for the OSGi Service Platform. A descriptive approach was described in [Gui *et al.* 2008], in which contracts were specified in OSGi component's meta-data. Then, a special service in the platform is responsible for the constraint resolution and component binding at run-time. Management and adaptation logic execute separately in a non-real-time partition of the platform, whereas real-time tasks, coded in native code, run directly in the RTOS layer. In turn, Américo *et al.* [Américo *et al.* 2012] proposed a dynamism-centric solution based on the freezing of the components in the platform and the use of real-time SLAs. Richardson and Wellings [Richardson & Wellings 2012] proposed a more comprehensive solution, which used RTSJ features to provide temporal isolation, admission control, WCET analysis and garbage collection pace configuration mechanisms. Similarly, Basanta-Val *et al.* [Basanta-Val *et al.* 2013] analysed the integration of RTSJ features on the OSGi platform, as new services for schedulability analysis, mode definition, recover, service characterization and access to the classes in the RTSJ API (more particularly, real-time threads).

2.5 Summary

Real-time systems are systems whose correctness depends on both logical and temporal aspects. Two properties must be present in a real-time system: **determinism**, which is the ability to ensure the execution of the application despite the presence of external and unforeseen factors; and **predictability**, which means the possibility to guarantee that for a set of real-time tasks, all their deadlines will be met. These properties are very related: predictability assumes determinism, because disturbances in the execution of an application may alter both functionality and performance aspects. Therefore, it is important to master the latency sources at all system layers, from the operating system until the applicative features.

As a result, several technologies have been developed to create a complete real-time infrastructure. Real-time operating systems, for instance, are operating systems which incorporate important features for achieving predictability, such as predictable scheduling policies and interruption handling. Real-time middleware has been created as well to isolate real-time applications from platform specific differences and provide infrastructure, distribution, resource management and heterogeneity transparency. Likewise, programming languages have also eliminated time-unbounded constructs, offered new extensions to express timing behaviours or even been created anew to allow the access to more predictable facilities.

Design methods have also been developed or adapted for real-time applications. It is very common in this application domain to use mathematics-based methods to specify the ap-

[31] More information available at `http://www.soda-itea.org`.

plication behaviour and composition, so that predictability may be verified by means of techniques like model checking and theorem proof. However, due to the complexity of the use of formal methods, in the industry it was preferred to adapt popular software engineering methods, like the structured and object-based approaches.
Due to their interaction with real world devices, some real-time systems must present dynamic adaptation features, that is, to be capable to perform unforeseen modifications on their properties and architecture at run-time, potentially without restarting. Many techniques have been developed to support dynamic adaptation, such as computational reflection and call interception. Service-oriented architectures and component-based design are two development approaches used to implement dynamically adaptive systems that have become very popular. This can be proven by the large number of component models and service-based technologies available in the industry. Service-oriented component models (*i.e.* a component model where components communicate in a SOA-like manner) inherits the benefits of both approaches. In addition, it enables separating SOA-related mechanisms from functional code, and adds the support for dynamic availability.
However, despite the flexibility provided by run-time adaptation and evolution features, these can not be directly applied to real-time systems. Indeed, the predictability that characterizes real-time systems requires a mastery of the impact of the dynamism on the application, so that determinism is guaranteed and timing constraints are satisfied. This conflict between predictability and dynamism is the subject of research works, what motivated the development of real-time variations of dynamism enabler approaches, such as real-time component models and real-time service-oriented architectures.
Among the existing approaches some can be identified as potential standards: AADL, a modelling language used as standard in avionics, with several existing tools for analysis and simulation; UML-MARTE, a UML profile for designing real-time systems, which besides having the support of UML editors and an expressive language for describing timing properties, also has analysis tools; and SCA, an OASIS standard designating a service-oriented component model, supported by various major IT vendors and service-based frameworks. These three models have the advantage of being extensible: SCA may support real-time properties and dynamism by means of extensions, just as AADL and UML MARTE components may be extended to interact by means of services.
This thesis aims to introduce a methodology for building real-time service-oriented component-based applications. With the modularity of the component-based approach and the dynamism of the service-oriented architectures, real-time systems may benefit from flexibility, interoperability, low maintenance and development costs, low time-to-market, and all other benefits resulting from the use of both approaches, which were listed in the previous sections of this chapter. This methodology includes a component metamodel, called **URSO**; the concepts present in this metamodel can be mapped onto extensible component models and modelling languages, so that applications developed with one real-time service-oriented and component-based technology may easily interact with and benefit from tools developed for another technologies.

Chapter 3

Unified Real-time Service-Oriented Component Metamodel

"He who moves not forward, goes backward."

Johann Wolfgang von Goethe

Contents

3.1 Motivations **36**
3.1.1 Problems in the development process of real-time systems 37
3.1.2 Requirements for a real-time service-oriented component model and framework 38
3.1.3 URSO main goals 39
3.2 URSO Concepts **40**
3.2.1 Deployment Concern 41
3.2.1.1 Resources 41
3.2.1.2 Machines, Nodes and Interconnections 43
3.2.1.3 Platform and Platform Modes 43
3.2.1.4 Definitions 43
3.2.1.5 Deployment Concern: UML Diagrams 47
3.2.2 Assembly Concern 49
3.2.2.1 Services, Dependencies and Bindings 49
3.2.2.2 Service Descriptions 50
3.2.2.3 Components, Composites and Compositions of Services . 50
3.2.2.4 Definitions 50
3.2.2.5 Assembly concern: UML Diagrams 58
3.2.3 Behaviour Concern 60
3.2.3.1 Service Operations, Parameters, Restrictions and Instructions 61
3.2.3.2 More about Policies 61
3.2.3.3 Definitions 61
3.2.3.4 Behaviour Layer: UML Diagram 69
3.2.4 Revisiting the Deployment Concern 70
3.2.4.1 Resource Requirements and Capabilities 70
3.2.4.2 Implementation, Instances and Nodes 71
3.2.4.3 Tasks and Mode Transitions 71

3.2.4.4 Definitions 71
3.2.4.5 Deployment Concern: UML Diagrams 73
3.2.5 Other aspects that must be taken into account 78
3.2.5.1 Technical Components 78
3.2.5.2 Run-time Monitoring 79
3.2.5.3 Applicative WCET vs. Technical WCET 80
3.3 Example: Dynamic Collision Detection Application **81**
3.3.1 Overview of the CD_x Benchmark 81
3.3.2 DCD_x: A Dynamic and Service-Oriented CD_x benchmark 82
3.3.2.1 A Service-Oriented version of CD_x 82
3.3.2.2 Adding Dynamism to CD_x 82
3.3.3 URSO Description of DCD_x 83
3.3.3.1 Application Design Overview 83
3.3.3.2 Platform Description: the Deployment concern 83
3.3.3.3 Services and Components: the Assembly concern 85
3.3.3.4 Mapping onto the Deployment Concern 88
3.4 Summary and Discussion **89**
3.4.1 URSO Overview 89
3.4.1.1 URSO Metamodel 89
3.4.1.2 Methodology 90
3.4.2 Discussion 92
3.4.3 Summary 93

This thesis' main contribution is to establish a metamodel for developing real-time service-oriented component-based applications. The metamodel, named Unified Real-time Service-Oriented (URSO) Component Metamodel, describes concepts in three different concerns: Behaviour, Assembly and Deployment.
In the Deployment concern, we will find platform infrastructure-related concepts, such as hardware component descriptions, components instantiation, resource reservation mechanisms and associations between concepts from service-based applications to the traditional real-time task-based execution model.
The Assembly concern includes concepts related to application architecture, its decoupling into components and composites and their interaction by means of explicitly declared services and dependencies.
The Behaviour concern contains concepts used for describing the services internal content. This information is used afterwards by the URSO framework to estimate worst case execution times and to monitor input sent to service implementations and corresponding output generated by them.
The next sections describe what motivated us to design URSO, as well as its different concerns and the content of each one of them. Metamodels are here presented by means of UML Class diagrams. They are also formalized in order to give them a more precise semantic. We also present UML Activity diagrams in order to show the methodology that must be applied in order to correctly use URSO concepts.

3.1 Motivations

As shown in the previous chapter, there are several solutions for designing dynamically adaptive and real-time applications separately. Some efforts have been made to join both

worlds as well: mapping SOA concepts onto real-time component models, developing real-time extensions to service platforms and creating new service frameworks supporting the expression and the guarantees needed for real-time applications. However, there are very few works on the development of service-oriented component models natively supporting real-time constraints. URSO aims to fill this gap by providing a metamodel for developing real-time service-oriented components.
In this section, we detail the factors that motivated us to develop URSO.

3.1.1 Problems in the development process of real-time systems

As stated before, initially real-time systems were designed and were implemented in dedicated hardware. With the advances in computer hardware, real-time concerns were concentrated in the system software layer. Since hardware facilities offer smaller latencies, the need of predictability has led real-time developers to code targeting specific platforms, benefiting from optimizations and tuning settings. However, despite increasing development costs due to the fact that it is necessary to train developers to code for a given hardware, this tuning decreases portability and makes software maintenance more difficult. Improvements in the software engineer discipline, have led to the conception of technologies to ease the development of real-time, which were detailed in the last chapter.
The introduction of service-oriented and component-based technologies in the development of real-time systems came naturally with the increased complexity of the latter. The combination of modularity and use of open standards with published interfaces enabled reusing real-time components among several applications, facilitating new developers' technology insertion and the integrating of heterogeneous components into applications and reduced maintenance constraints [Moreland 2013]. These properties are very important in today's real-time systems for several reasons.
The decomposition of applications into smaller pieces is not a new concern for real-time applications. However, today these multiple pieces are often developed by different companies, with different development cycles and different technologies. It is important for real-time applications to take into account during all the development process the different levels of trust in quality of service, and to foresee actions in case of failure of these untrusted components. This implies a more costly development and potential evolutions in real-time software. Hence the need of flexible real-time software.
For instance, in a satellite after its launch, only its embedded software elements can be modified. These modifications are used to repair malfunctioning software elements, to compensate satellite mistrusts and to adapt or evolve the satellite mission. The software must often be maintained during the whole satellite life-cycle, which may vary between 10 and 25 years, and so must the means to maintain the software: it is important to ease the integration of new technologies through interoperability or to ensure the competence of developers in specific technologies for a long time. In addition, the software may be updated several times due to requirements modification or context adaptation.
The use of models (particularly service-oriented component models) and their integration in the different phases of the development process, just as the automation of the process through the use of tools and middleware may help to solve these recurrent issues and offer a generic solution allowing development and characterization of reusable and modular real-time software.

3.1.2 Requirements for a real-time service-oriented component model and framework

A real-time service-oriented component model is in the intersection of three application domains: service-oriented computing, real-time computing and component-based design. It must thus satisfy requirements concerning all of these domains. The most important requirements in such a model are:

(Real-time)

- Provide determinism and real-time performance;
- Provide real-time QoS guarantees;
- Provide compatibility with real-time requirements;

(Service-Oriented)

- Provide continuous and dynamic interaction between services;
- Provide support for service modelling and orchestration;
- Provided bounded response time for SOA framework operations;
- Perform rebinding if performance is not satisfactory or if faults are detected;
- Provide network interconnection transparency;

(Component-based)

- Provide support for horizontal and vertical component composition;
- Enable unit and element-level testing and debugging;
- Enable multi-element integration testing and debugging;
- Support multi-element interoperability;
- Support the scheduling of opaque and distributed elements;
- Provide distribution transparency;

(General-purpose)

- Provide Fault tolerance, recovery and isolation;
- Minimize manual configuration;
- Allow cost-effective functionality enhancement;
- Provide deployment support;
- Support automatic monitoring and management;
- Provide policy enforcement;
- Perform data collection;

The three first requirements come from real-time applications. Deterministic real-time performance can be measured in terms of latency (internal, when it references a single service, or end-to-end, when it refers to service assembly), periodicity and predictability (in this case, to ensure that a service invocation will be completed within a specific timing constraint). The provision of real-time QoS guarantees requires performance estimation methods, such as logical and quantitative model checking (verification), simulation, testing (validation) and service scheduling analysis. It requires the enhancement of services with real-time attributes, the expression of timing constraints and the characterization of workloads.
The next five requirements issue are from the service-oriented computing domain. The model must support dynamic service deployment and reconfiguration, without downtime and in bounded time. Other operations must react within timing constraints as well, such as service discovery, management, assembly and validation. Through the model, developers must be able to model real-time services and service orchestration. In addition, the model must foresee automatic dynamic reconfiguration in case of fault detection or unsatisfactory performance. Finally, services must be able to communicate with each other locally and remotely, so abstractions to networks (and related aspects such as topology, protocol and routing) must be provided [Tsai *et al.* 2006].
The six requirements that come next emerge from the component-based approach. Besides supporting component composition, it is important to provide support for testing and debugging components and components of components (hereafter called *composites*) with different granularities. It is also important to take into account component dynamic availability and distribution aspects in those tests. The model must be able to support components developed with different technologies and provide abstract interfaces for the communication between those components. Furthermore, it is important to consider that components are opaque elements, that is, they must be deployed along with their services without requiring further implementation details.
The last seven requirements are applicable to all applications. Fault tolerance and recovery are important so that the availability, correctness and determinism of the system are not compromised upon the detection of a fault or the violation of QoS requirements in a given component. Fault isolation is important as well to avoid fault propagation to other components. Frameworks supporting the model must be able to minimize human interference and configuration, to avoid error-prone tasks, toreduce design time and to reduce costs. In addition, it is desirable that the model (and implementing frameworks) may be extended to support new technologies or protocols, for instance. For the same reason, it is important to enable the automatic monitoring and management of components and resources, based on policies that are defined by the developers and enforced by frameworks. A model must also support deployment tasks and, if possible, help in the mapping of software resources towards hardware resources respecting the components' QoS requirements. In the real-time applications context, deployment also means resource and network reservation, so that temporal isolation is achieved. Finally, data collection may help to evaluate system performance and, if necessary, modify properties on the model itself to improve predictability.

3.1.3 URSO main goals

URSO aims to provide a *metamodel* for developing real-time service-oriented components. Metamodels are central to the Model-Driven Engineering (MDE). MDE principles are based on two entities (system and model) and two relations (conformance and representation). A *model* is a structure *representing* some aspects of a certain *system* and *conforming* to the definition of another structure called **metamodel**. For instance, component models are system models that focus on modularity aspects. Thus, metamodels are used to define and

express one or more models. *Meta-relations* are the relations between model elements and metamodel elements (which are called *meta-elements*). Metamodels can be used to define aspects of several models at once [Bézivin 2005].
Thereby, we have decided to reason at the metamodel level. This metamodel can be mapped onto standard and extensible models. Moreover, developers can use several different component models (and the tools available for those models) with some extra extensions in order to address the real-time and/or service-oriented paradigms. URSO proposes as well a methodology, indicating the actions that must be taken by all the actors (runtime, administrators and developers) for both logical and timing correctness of real-time service-oriented application. Some framework implementation details must be provided too, in order to parse the information, and deploy and manage the application accordingly to the metamodel.
In short, these are the main objectives of the URSO component metamodel:

- Provide a technology-independent metamodel for flexible and modular real-time applications;
- And define a methodology to design, implement, deploy and execute real-time service-oriented component-based applications.

In addition, it must be generic enough so that real-time and adaptive industry standard component models may be easily conform to it by adding some extensions.
To illustrate the application of URSO, we have extended the AADL, UML-MARTE and SCA models so that they are conform to our metamodel. As result, applications modelled in one component model may be transformed into another to benefit, for instance, from development and analysis tools support. They may also be developed in URSO and mapped onto one or more component models to profit from its execution support.

3.2 URSO Concepts

In this section, we introduce the URSO metamodel. Each of its concerns is described in detail in the following subsections. It is important to mention that the concerns are not completely independent; some concepts are present in more than one concern (cross-cutting concerns[1]). Since each concern corresponds to a different color, these cross-cutting concerns are easily identified in the UML Class diagram due to the fact their colors correspond to the different concerns they are present on.
First, we introduce basic aspects of the Deployment concern related to the platform infrastructure and hardware, such as resource and nodes definitions. Then, we present the Assembly concern. It contains concepts related to application development, its decoupling into components and services and the declaration of dependencies towards these services. The Behaviour concern is presented afterwards, adding semantic restrictions and information about the Service Operations that will be used to estimate their Worst Case Execution Times (WCET). Then, we revisit the Deployment concern in order to add concepts that allow mapping instances of the components presented by the Assembly concern, defining service compositions and real-time tasks. Finally, we present some other aspects which are more implementation-related but that must be taken into account as well when modelling an application and conclude the chapter. Figure 3.1 depicts a taxonomy of URSO potential

[1]The term 'cross-cutting concern' is more often used in Aspect-oriented programming to represent an aspect that affects more than one concern, like logging or authentication concerns. Here it is just used to show a concept that is present in different concerns of the metamodel.

users. Six main categories can be identified: Modellers, which develop models for both hardware and software entities; Developers, which produce artefact and code implementation for the modelled entities; Deployers, which are responsible for grouping software implementations into deployable units and mapping them onto hardware entities; Analysts, which may analyse the model created for different entities against their real implementation and validate compositions against their constraints; Administrators, which can monitor both application and platform at execution time and perform actions accordingly; and Executioners, which are going to instantiate the software entities on the target hardware and execute them.

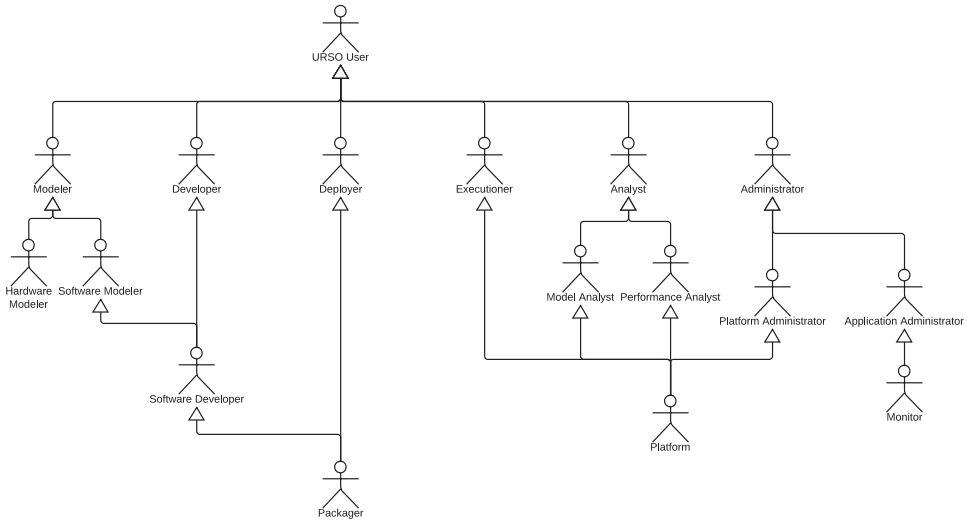

Figure 3.1: A taxonomy of potential URSO users.

Figure 3.2 summarizes the potential interactions of the actors with the model and modelled entities. Although it is possible to infer the description of each use case through its name, the reader may find more details in Appendix section A.2.

3.2.1 Deployment Concern

The most fundamental concept in URSO's deployment concern is that of *platform*. However, before formally defining what a platform is, we must explain what it represents and we must define some more basic concepts which are part of a platform's definition.
A platform represents the infrastructure on which the framework, and consequently the real-time service-oriented applications are going to be executed. A platform description must include then the machines that are going to be used, the resources provided by these machines, connections between these machines and information about platform configurations.

3.2.1.1 Resources

We define **resource** as anything that can be provided or required by a model entity. Examples of resources are native libraries, the underlying operating system, available RAM memory, available disk space and CPUs. Entities may be bound to resources in two different ways: if the entity provides a resource, we consider this as a *capability*; however, if it

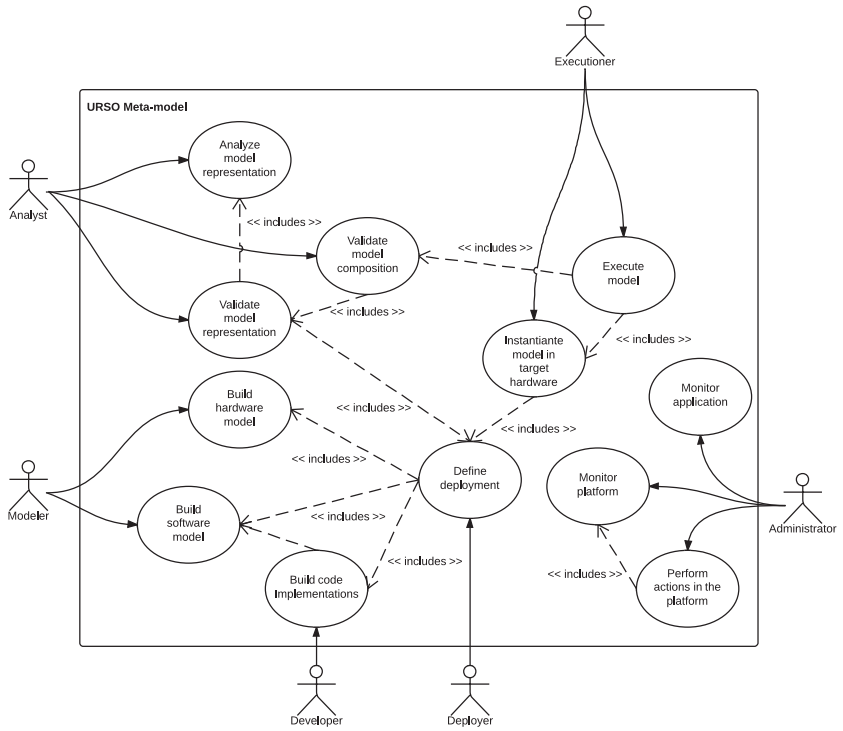

Figure 3.2: Potential use cases for the URSO metamodel

requires a resource, it is considers as a *requirement*. A resource is qualified by its name, its type and a set of resource properties, such as 'Version', 'Vendor', and 'Quantity'.
Some resource properties are required for certain resource types: for instance, *quantity* is required for the *processor* resource type. In addition, some other properties are automatically managed for some resource types; for instance, the property *available* is automatically updated by the platform as components reserve them during their deployment. Furthermore, some properties may make sense for one type of resource but not for another; *e.g.* since a machine can only run a single operating system at a time, it is pointless to have a property containing the number of available operating systems.
Here is a list of commonly used resource types and properties:

- *Processor*[2]: Processor's usual properties are *capacity* (the total number of Processors), *available* (number of currently available Processors) and *minimum* (the minimum quantity of available Processors required for the system to work). Additional information can be added, like processor architecture and endianness;

- *Memory and disk storage*: Memory[3] and disk use similar properties as those of Processors. The main difference is the unit used to describe the resource (CPUs in Processors, bytes in memory and storage);

- *Operating system and native libraries*: An operating system can be described by means of its name and properties such as its *Version* or *Memory Addressing Word Size*. Just like processors, extra information can be added, like target architecture.

[2]By processor, we mean a processor unit. In multi-core architectures, it corresponds to a processor core
[3]By RAM Memory, we refer to a memory space which is available to all processes in a machine. Thus it does not include processor's cache memory and other type of memories with limited visibility.

While some resources are unique to a machine, some others can be shared between machines (e.g. shared memory, shared disk file system or network connections). The access to shared resources is controlled according to an *access protocol*. This access protocol may define the access as exclusive or not.
Our resource mechanism is based on OMG Deployment and Configuration specification [Object Management Group 2006b] and it is considered mainly as a defensive strategy for resource reservation. We do not consider in this thesis the time it takes for accessing a resource (either local or remotely). Since we are focusing on service-oriented systems, it would be possible to consider the access to resources as calls towards a platform service and allow the platform to bound the execution time of this operation.

3.2.1.2 Machines, Nodes and Interconnections

A **machine** corresponds to the physical concept of computer: a device consisting of at least one central processing unit and some sort of storage system, both used to execute a finite set of operations. A **node** is a logical abstraction which refers to a machine or a part of it (a subset of resources, for example). The main characteristics of a node is that its resources are isolated (except if it contains a platform shared resource) from those of other nodes. That means that node resource failures can not be propagated to neighbour nodes.
Machines communicate with each other by means of *Interconnections*. An **interconnection** links two or more machines and is bound to one or more **communication protocols**. Since we are dealing with real-time systems, communication delay and response time must be bound and known by the platform. Thus, the platform can take them into account to estimate execution times and perform feasibility analysis. Nodes connect with each other by means of the interconnections of the machines that contain them. If a component in a node wants to communicate with a component in another node through a specific protocol, there must exist at least one interconnection between the machines containing these nodes which supports that specific protocol.

3.2.1.3 Platform and Platform Modes

A **platform** corresponds to the physical infrastructure on which the URSO framework will be deployed. A platform is formed by one or more machines and may contain several resources. The runtime verifies that resources declared by machines and nodes are also declared in the platform resources, and that their quantities do not exceed the amount of available resources.
Platforms can declare **policies**, which can be applied to applicative components (for instance, assigning a fail-stop policy to the non-respect of a service method input restriction) and some technical components (e.g. scheduling policy). A policy is related to one or more executed **actions** that can interfere on applicative components and in the platform itself.
A platform can have several predefined configurations: these configurations are called **modes**. Platform modes have their own set of resources capabilities, policies and nodes, which are subsets of those of the complete platform. Only one mode can be active at a time. Transitions between nodes can entail: shutting down several nodes, halting their tasks, stopping their components, and removing those components' services from service registries.

3.2.1.4 Definitions

We present here the definitions of the concepts presented above.

URSO Definition 1 (Resource)**.** A *Resource* is a 3-uple

$$\langle rName\ ,\ rType\ ,\ rProp\rangle,$$

where *rName* is a String-typed identifier, *rType* specifies its type and *rProp* is a set of 3-uples

$$\langle name\ ,\ value\ ,\ type\rangle,$$

where *name* specifies a property name, *value* specifies a property value and *type* is a *ResourcePropertyType.*

If we would like to describe a Linux operating system on a server, along with its Kernel version, the Resource description would look like this:

$$\langle \text{`}Linux'\ ,\ \text{`}OS'\ ,\ \{\langle \text{`}Version'\ ,\ \text{`}Version'\ ,\ \text{`}3.2.0-33-lowlatency'\rangle\}\rangle.$$

A Resource r: $\langle rName\ ,\ rType\ ,\ rProp\rangle$ is *quantifiable* if there are properties named *'Quantity'* and *'Available'* in its property set *rProp*[4]. For quantifiable Resources, it is imposed that the quantity of available Resources of a given type must always be lower than the declared total quantity, that is, the value of the *'Available'* property must remain smaller than the value of the *'Quantity'* property.
There is a special type of Resource called *Shared Resource.* Its description follows the same structure as that of normal Resources, but upon the requirement of a Shared Resource, a given entity must specify an *Access Protocol*, which will establish rules for accessing this Resource. Shared Resources and Access Protocols will be better detailed and explained further, in the section 'Revisiting the Deployment Concern'. Examples of Shared Resources are files and shared memories. For instance, upon declaring a Virtual Machine, which is a Resource shared by many applications and that can only be accessed as a reader, the application deployment administrator may specify that Component Implementations access the JVM Shared Resource as readers according to a "Concurrent access" Access Protocol.

URSO Definition 2 (Communication Protocol)**.** A *Communication Protocol* is a 2-uple

$$\langle pname\ ,\ protocolProperties\rangle$$

where *pname* is a String and *protocolProperties* is a set of 3-uples

$$\langle name\ ,\ value\ ,\ type\rangle,$$

where *name* specifies a property name, *value* specifies a property value and *type* is a *ProtocolPropertyType.*

The protocolProperties field corresponds to the protocol QoS description. Among the property types, we may find 'Worst case throughput', 'Worst case delay' and other non functional aspects.

URSO Definition 3 (Interconnection)**.** An *Interconnection* is a 2-uple

$$\langle participants\ ,\ supportedProtocols\rangle,$$

where *participants* is a set of two *Machine* names (String) and *supportedProtocols* is a non-empty set of *Communication Protocols* supported by this Interconnection.

[4] *Quantity* and *Available* properties are comparable because they have the same Type (*e.g.* both must have the type *MemoryInMB*, referring to Memory measured in MBytes or *ProcessingUnit*, referring to processor cores. This will be clearer in the example depicted in the Section 3.3).

Interconnections represent a point-to-point connection between two machines in the platform. In order to use a given protocol *prot* to bind two Components deployed in two Machines m_1 and m_2, at least Interconnection must exist between both Machines (that is, there must exist an Interconnection which has n_1 and n_2 in its set of Machine names), and the protocol *prot* must be supported by this Interconnection (*prot* must be present in the set of supported Communication Protocols *supportedProtocol*).
Since our restriction does not impose $n_1 = n_2$, this is also valid for binding two Components in the same Machine. A platform must inform a default Communication Protocol, which will be used when no Communication Protocol is informed. In addition, each machine must have a local Interconnection (that is, an Interconnection where the two participants are actually the same machine).
For instance, if the system administrator wants to describe that two machines A and B are linked by an InfiniBand bus that supports IPoIB[5] and SDP[6], the corresponding Interconnection would look like this:

$$\langle\{\langle machineA\ ,\ \textbf{machine description}\rangle\ ;\ \langle machineB\ ,\ \textbf{machine description}\rangle\}\ ,$$
$$\{\langle IPoIB\ ,\ \textbf{IPoIB timing description}\rangle\ ;\ \langle SDP\ ,\ \textbf{SDP timing description}\rangle\}\rangle.$$

An Interconnection can uniquely identified by the elements of its *participants* set.

URSO Definition 4 (Node)**.** A *node* is a 2-uple

$$\langle name\ ,\ capabilities\rangle,$$

where *name* refers to the node's name (a String-typed identifier), *and capabilities* is a set of Resources.

As mentioned before, a Node corresponds to a Machine or part of a Machine (*e.g.* a set of processor cores or other resources). The set of Resource capabilities of a Node is a subset of the Resource capabilities set of the Machine that contains it; in the case where a Node corresponds to a Machine, both *capabilities* set (that of the node and that of the machine) are equal. When a node does not contain any processing unit among its Resource capabilities, it can not be used to deploy tasks.

URSO Definition 5 (Machine)**.** A *machine* is a 3-uple

$$\langle name\ ,\ nodes\ ,\ capabilities\rangle$$

where *name* is a String-typed identifier, *nodes* is a set of Nodes, and *capabilities* is a set of Resources.

A Machine corresponds to a physical computer. Thus, its interconnections, which are listed on the platform, are real links towards other machines with their supported protocols and their timing properties (which are going to depend on real world physical factors such as machines' real position and the type of the wire used to link them)[7].

URSO Definition 6 (Rule)**.** A *rule* is a 2-uple,

$$\langle name\ ,\ expression\rangle,$$

where *name* is a String-typed identifier, and *expression* is a Boolean expression.

[5] IPoIB stands for IP over InfiniBand, it adds an Ethernet layer atop an InfiniBand bus
[6] SDP stands for Sockets Direct Protocol, it adds a socket layer atop an InfiniBand bus
[7] Clusters may emulated in the metamodel by adding an "abstract" machine which connects to all machines belonging to the cluster. Consequently, two clusters can be connected with each other by connecting their corresponding "abstract" machines.

URSO Definition 7 (Action)**.** An *action* is a 2-uple,

$$\langle name\ ,\ command \rangle,$$

where *name* is a String-typed identifier, and *command* is a machine-executable command.

The commands defined by an Action may refer to the variables used in the Rule expression and to environment variables.

URSO Definition 8 (Policy)**.** A *policy* is a 3-uple,

$$\langle name\ ,\ rules\ ,\ actions \rangle,$$

where *name* is a unique String-typed identifier, *rules* is a set of Rules, and *actions* is a set of Actions.

Policies are used to indicate actions which must be taken automatically in case of certain events. For instance, a fail-stop policy would look like this:

$$\langle \text{“}Fail - Stop\text{”}\ ,\ \langle \text{“}Fail\text{”}\ ,\ platform.errorState = \mathbf{true} \rangle\ ,\ \langle \text{“}Stop\text{”}\ ,\ platform.stop() \rangle \rangle$$

where the first element of the policy corresponds to a Rule that queries whether the platform is in an error state and the second element presents a command which can be used to stop the platform. Policies are similar to the Event Condition Action (ECA) structure used to express active rules in active databases and event-driven architectures [Dittrich *et al.* 1995].

URSO Definition 9 (Platform)**.** A *Platform* is a 6-uple

$$\langle machines\ ,\ modes\ ,\ policies\ ,\ capabilities\ ,\ \underline{currentMode}\ ,\ interconnections \rangle,$$

where *machines* is a set of machines, *modes* is a set of Platform Modes, *policies* is a set of Policies, *capabilities* is a set of Resources, *currentMode* is a Platform Mode and *interconnections* is a set of Interconnections.

We consider that the following properties must be met in a Platform:

- The platform currentMode field underline means that this field will be automatically updated by the framework. The Platform administrator can use this field to indicate an initial Platform Mode.
- The current Platform mode is always an element of the set of Platform Modes *modes*. This predicate must be true despite platform dynamic mode changes;
- The Resources referenced by the Platform Machines (*i.e.* the Machines in the set *machines*) must be present as well among the Platform Resources listed in the set *capabilities* (just as the Nodes Resources must be a subset of its containing Machine Resource).
- The total quantity of quantifiable Resources declared for all the Platform Machines is not greater than the total quantity of Resources declared by the Platform that contains them. In addition, at execution time, the total amount of available Resources at the Machines of a Platform must correspond to the amount of available Resources of the Platform and the current Platform Mode itself.

URSO Definition 10 (Platform Mode)**.** A *Platform Mode* is a 6-uple,

$$\langle name\ ,\ \underline{isActive}\ ,\ machines\ ,\ policies\ ,\ capabilities\ ,\ interconnections \rangle$$

where *name* is a unique String-typed identifier, *isActive* is a Boolean variable, *machines* is a set of Machines, *policies* is a set of Policies, *capabilities* is a set of Capabilities and *interconnections* is a set of Interconnections.

A Platform Mode must present the same properties present in Platforms concerning Machines and Resources. In addition, only one Platform Mode can be active at the time, that is, only the Platform Mode indicated by the Platform's *currentMode* field must have its *isActive* flag set to true; all the other Platform Modes must have this flag set to false. This field is automatically updated by the platform.
An example of Platform Mode is the *Static mode*. In the Static mode, dynamic deployment and dynamic adaptation are disabled. Thus, new Components can not be installed and current Components can not be uninstalled or updated (just as the Architectural freezing approach described in [Américo *et al.* 2012]). In addition, Service Implementations can not be substituted in Service Compositions. This can be implemented by disabling the technical component responsible for performing operations related to dynamic adaptations and activating policies which fire errors in case of dynamic operations.
Another example of Platform Mode is an *Energy saving Mode*, in which only a subset of Machines are activated. Energy saving can also be executed at hardware level, by decreasing the processor frequencies and voltage and disabling only a subset of processor cores, but this is feature not available in all processors and since URSO intends to be generic enough to be executed on any hardware infrastructure, is not covered in this work.
The transition from one mode to another implies a reconfiguration phase, where some tasks, nodes and components may be disabled, others can be included and many verifications can be performed. Thus, this can not be done during the execution of critical/real-time tasks. This transition/intermediary mode is started after the execution of all current tasks[8].

3.2.1.5 Deployment Concern: UML Diagrams

A Deployment UML Class diagram depicting the concepts presented in this section can be found in Figure 3.3.
As stated before, the main element of this diagram is that of Platform. A Platform is composed by Machines, Platform Modes (which are themselves platform descriptions), Resource capabilities and Policies. Machines are composed by at least one Node. Interconnections link Machines and Nodes and may offer the support of several Communication Protocols. The communication in each protocol must be qualified, so that the platform may take communication delay into account in worst case execution time calculations. Given an active Platform Mode, its provided Resources can be scattered through its different Machines, and consequently, Nodes. These Resources can be qualified as well, allowing the framework to reserve them in advance and avoid the introduction of non-deterministic waiting in applicative components' execution. The Platform and its Platform Modes also provide Policies, which can be associated to other concerns' entities (which will be further detailed), enabling the Platform to properly respond to unexpected events.
An Activity diagram showing a methodology to create the platform description as well as the actions executed by the framework to process this description is presented in Figure 3.4. First, platform administrator must describe the platform in terms of Resources, Communication Protocol support and policy Actions and Rules. The Resource description is then used to describe the Platform Modes and its Machines and Nodes. Interconnections must be used to link Machines in a Platform Mode, having in mind that they must have at least one Interconnection (with itself). Policies can then be created. Communication protocols

[8] In the case of periodic tasks, its periodic behaviour is disabled.

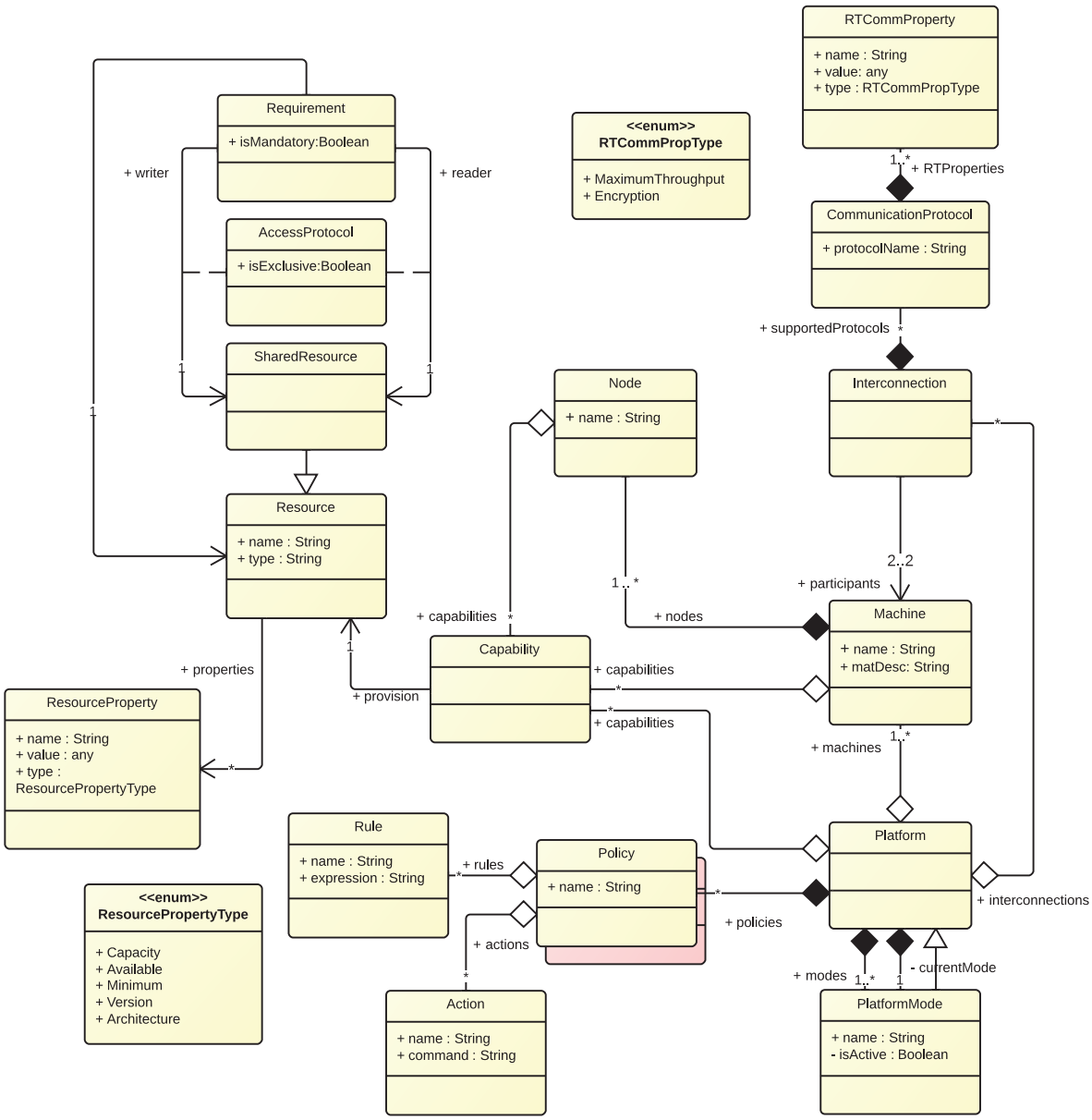

Figure 3.3: URSO Component metamodel - Deployment concern concepts

are associated to Interconnections and in the end we have a Platform description, which is used as input for the framework start. Before starting, the framework performs validations on Platform Modes, Machines, Nodes and declared Resources, according to the properties listed in their respective definitions. It is important to mention that in this initialization phase, the framework is already executing in the described Platform. Special attention must be paid to the fact that the framework itself does not perform any verification on the Resources to ensure, for instance, that the declared number of processors matches with the total number of cores in the machines. However, modifications can be performed in the framework in order to perform such tasks. In addition, technical components responsible for

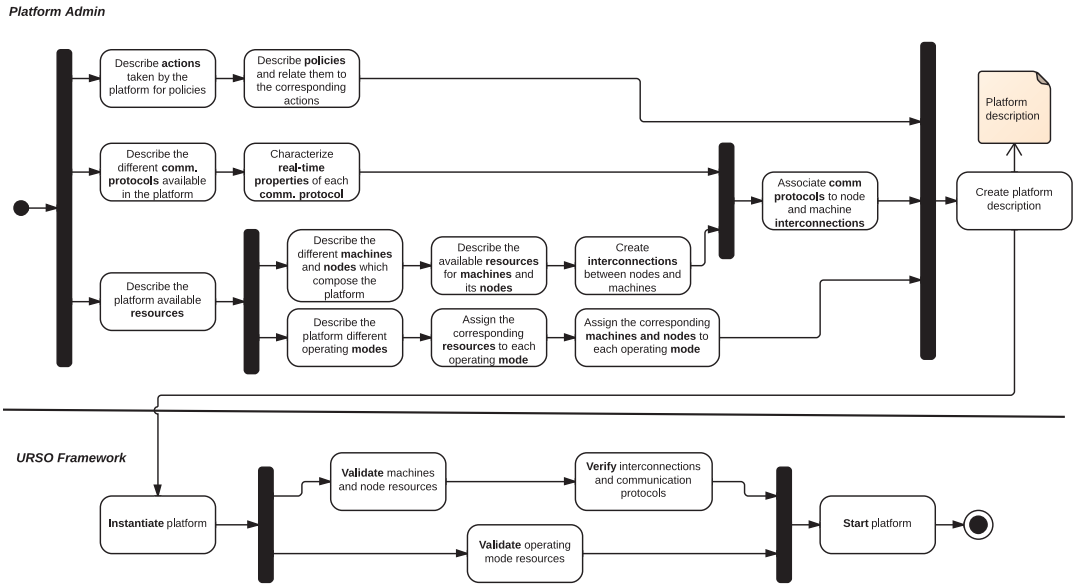

Figure 3.4: URSO Component metamodel - Platform definition

parsing and verifying the platform description must already be installed before applicative components[9].

3.2.2 Assembly Concern

While the Deployment concern contains concepts related to infrastructure lower levels, the Assembly concern concepts are in the highest abstraction level. These concepts are used to model the application architecture in terms of components, composites, service provision and service dependencies. Services and dependencies are bound to a service description, which can be seen as a contract, containing a list of operations that are or must be provided by the service provider Implementations.

3.2.2.1 Services, Dependencies and Bindings

URSO is a service-oriented component metamodel, thus URSO components interact by means of **Services**. Services are identified by a name and are associated to a Service Description and a list of Service Properties. Both Service Description and Service Properties are published in Service registries upon the instantiation of their providing Component in the Platform.

A **Dependency** towards a given Service Description means that a service provider Implementation for this Service is required by the Component. Just as Services, Dependencies are identified by a name. They also have a Multiplicity and may declare a Service Ranking function to filter and rank the candidate service provider Implementations. Depending on its Multiplicity, a dependency may be optional/simple ([0, 1]), optional/multiple ([0, ∞)), mandatory/simple ([1, 1]) or mandatory/multiple ([1, ∞)).

The link between a Service and a Dependency is called **Binding**. A Binding may be static or dynamic. A static Binding means that once a Service is linked to a dependency, this link

[9]Technical components are discussed in the Section 3.2.5

will not be removed, unless the Components responsible for providing or consuming the Service are stopped. In dynamic Bindings, the service provider may be modified during the application execution. In the Platform Modes that enable application reconfiguration, the framework will try to satisfy dynamic Dependencies with the best ranked available Service Implementations for them. As Components can be dynamically started in the Platform, the publication of Services in the Service registries changes the set of available Services, potentially adding Implementations which are best ranked than the available ones for a given Dependency. Bindings are associated to Communication Protocols, which are also present in the Deployment concern and were presented in a previous section. Cross-concern concepts like Communication Protocols are very important for putting all concepts together without mixing different concerns.

3.2.2.2 Service Descriptions

Service Descriptions are defined as a list of Operations, along with its Parameters and Restrictions[10]. They can be used for both publishing and retrieving Services; when the list of Operations of a Dependency Service Description is contained in a published Service Description (*i.e.*, there is a Service Description which contains at least the same set of operations), the framework considers that the published Service matches the Service Dependency.
An Operation contains a list of Parameters. These parameters can be classified as input or output. Inputs and outputs are associated to types, which are described in the Platform. This type qualification contains information such as the size of objects of this type, which is important for estimating worst case execution times. Operations will be further discussed in the 'Behaviour concern' section.

3.2.2.3 Components, Composites and Compositions of Services

Components are the software units that provide and require Services. They can be grouped into Composites. In turn, these Composites can be used as Components inside other Composites, making URSO a hierarchical component metamodel. Each Composite has an internal service registry, which lists the Services published by its Components. In order to make an internal Service visible to other same-level Composites (or to search Services in same-level Composites to satisfy internal Dependencies), a Service (or Dependency) must be *exported* (*imported*, in the Dependency case).
Components may be qualified by means of properties and they can have several Implementations in different technologies (Java classes, BPEL processes, C applications, Python scripts, and so forth).
Service Compositions reunite Services in order to mediate them and perform a business function upon some data. They are going to be assigned to tasks and executed. Depending on the language used to describe a Service Composition, the corresponding executable code can be automatically generated by the platform. Despite its similarity, Composites and Service Compositions can be developed independently. Service Compositions represent a horizontal service composition (a composition over time, just as business process), whereas Composites are vertical component compositions (a hierarchical composition and subcontracting of modules, as present in component models such as SCA or AADL).

3.2.2.4 Definitions

This section presents some formal definitions for the concepts introduced above.

[10]Restrictions correspond to Pre-conditions, Post-conditions and Invariant predicates defined over Operation Parameters. They will be explained in detail in the Behaviour concern section.

URSO Definition 11 (Service)**.** A *Service* is a 5-uple

$$\langle name\ ,\ description\ ,\ properties\ ,\ bindings\ ,\ dependencies\rangle$$

where *name* is a unique String-typed identifier, *description* is a Service Description, *properties* is a set of Service Properties, *bindings* is a set of Bindings and *dependencies* is a set of Dependencies.

A Service corresponds to a provided functionality. It is identified by a name and contains a Service Description with its lists of operations, a set of Service Properties which qualifies the Service and helps other Components to filter Implementations, and a set of Bindings to inform the supported protocols for a given Service. Depending on the protocols supported by a Service, its publication can be done in different ways (for instance, a Web Service-typed Service will be published using UDDI or WS-Discovery, whereas a Java Service can use OSGi-like service registries). The set of Dependencies is used to inform Service-level Dependencies, that is, which Dependencies must be satisfied before the Service is published. That allows a Component to only stop a subset of Services if one of its optional Dependencies is not satisfied.

URSO Definition 12 (Service Property)**.** A *Service Property* is a 2-uple

$$\langle spName\ ,\ spValue\rangle$$

where *spName* is a unique String-typed identifier and *spValue* is a String.

The Service Property name is an unique ID due to the fact that a Service cannot be declared with two properties having the same name and different values. Despite the fact that Service Property values are String, they can be converted to other value type in order to perform comparisons and to assign a rank for number-type Service Ranking functions.

URSO Definition 13 (Dependency)**.** A *Dependency* is a 5-uple

$$\langle name\ ,\ description\ ,\ multiplicity\ ,\ ranking\ ,\ bindings\rangle$$

where *name* is a unique String-typed identifier, *description* is a Service Description, *multiplicity* is a Multiplicity, *ranking* is a Service Ranking function and *bindings* is a set of Bindings.

A Dependency is identified by its name and contains a Service Description listing the Service Operations required by the Component, the Multiplicity of this Dependency, the Service Ranking function that will filter the available Service Implementations, and a set of Bindings that will inform the set of Communication Protocols supported by the Dependency.

URSO Definition 14 (Service Ranking function)**.** A *Service Ranking Function* is a String-typed expression which, when parsed, can be considered at the same time as a *filter* and a *sorting function* for service provider Implementations.

A Service Ranking function is considered as a filter when it compares a property value against another value. For instance, in functions like `fps > 25` or `location = "underground"`, all services whose properties match these criteria are considered as equal. An example of sorting function could be `max(fps)`, which picks all the service provider Implementations which have a property called `fps` and sort them in a way that the highest `fps` value is the first of the list, the second highest value is the second and so forth.
Pre-defined operators for ranking Service Properties are *max*, which arrange data in a high to low sequence (descending sort), and *min*, which does the opposite (ascending sort). Comparison operators for filtering are the '*equal*' (=), '*not equal*' (! = or $\neq$), '*less than*' (<),

'*greater than*' (>), '*less than or equal to*' (<= or ≤) and '*greater than or equal to*' (>= or ≥) operators. Different expressions can be assembled by using the logical operators '*and*' (&) and '*or*' (|). Expressions can also be negated by using the operator '*not*' (! or ¬). Dynamic service ranking is discussed in details in [Bottaro & Hall 2007].

URSO Definition 15 (Multiplicity)**.** *Multiplicity* is a String-typed enumeration which can assume the following values*:*

- *'0..1';*
- *'1..1';*
- *'0..*';*
- and *'1..*'.*

Multiplicities starting by zero are called *optional*; those starting by one are called *mandatory*. Optional Dependencies do not need to be satisfied to validate the Component; mandatory Dependencies must be satisfied in order to enable the Component to start. If Services declare optional Dependencies as Service-level Dependencies, the fact that the latter are not satisfied will only impact the Services that depend on it; thus the whole Component does not need to stop.
Multiplicities ending by one are called *simple*; those which end by the star sign are called *multiple* or *aggregated*. A simple Dependency means that only the best ranked service will be used to satisfy the Dependency, while in multiple Dependencies, all the service provider Implementations which satisfy the criteria defined by the ranking and filter function will be injected.

URSO Definition 16 (Binding)**.** A *Binding* is a 3-uple

$$\langle \underline{services} \ , \ \underline{dependency} \ , \ isDynamic \ , \ cProtocol \rangle$$

where *services* is a set of Services, *dependency* is a Dependency, *isDynamic* is a Boolean variable and *cProtocol* is a set of Communication Protocols.

The service and dependency fields are assigned by the framework itself. Depending on the value of the isDynamic field, services may assume different values in the same execution. Communication Protocols can be assigned to a Binding; if none is assigned, the framework uses the Platform's default Communication Protocol.

URSO Definition 17 (Service Description)**.** A *Service Description* is a 2-uple

$$\langle name \ , \ operations \rangle$$

where *name* is a unique String-typed identifier and *operations* is a set of Service Operations.

A service provider Service Description SD_1 is considered as a match to a service consumer Service Description SD_2 if SD_2's set of operations is contained in SD_1's set of operations. A common Communication Protocol must also exist among both Service and Dependency. These matches are then filtered and ranked according to the Dependency Service Ranking function.

URSO Definition 18 (Service Operation)**.** *A Service Operation is a 4-uple*

$$\langle name \ , \ parameters \ , \ instructions \ , \ restrictions \rangle$$

where *name* is a unique String-typed identifier, *parameters* is a list of Parameters, *instructions* is a list of Instructions and *restrictions* a set of Restrictions.

Service Operations are the main elements of Service Descriptions. They usually represent the concept of subroutine in computer programming. In this work, we are going to focus on the imperative programming paradigm and consider that they will be mapped onto callable units, like Java methods, Python functions and Pascal procedures, but they can also represent data service operations, such as reading and writing data entities, and event-based operations, like event publication and handling.
Service Operations will be further discussed and detailed in the Behaviour concern section.

URSO Definition 19 (Component)**.** A *Component* is a 5-uple

$$\langle name\ ,\ services\ ,\ dependencies\ ,\ implementations\ ,\ properties\rangle$$

where *name* is a unique String-typed identifier, *services* is a set of Services, *dependencies* is a set of Dependencies, *implementations* is a set of Component Implementations and *properties* is a set of Component Properties. The element *implementations* is optional.

Components may be used to qualify a family of implementations. An application developer may create a component with no implementation to indicate a family of implementations. Then, during the description of the Component Implementation, only has to indicate the name of the Component it is implementing and the other Component elements (Services, Dependencies and Properties) are automatically inferred.
The Services provided by Component are, by default, only visible to other Components inside the same Composite. The same is valuable for Component Dependencies. In order to make the Services visible to other Composites, or to get Services published by other Composites, Services and Dependencies must be *exported* and *imported*, respectively.

URSO Definition 20 (Component Property)**.** A *Component Property* is a 2-uple

$$\langle cpName\ ,\ cpValue\rangle$$

where *cpName* is a unique String-typed identifier and *cpValue* may have any value.

Depending on the technology used to implement the component, these properties can be injected in the Component Implementation objects upon their instantiation.

URSO Definition 21 (Component Implementation)**.** A *Component Implementation* is a 3-uple

$$\langle language\ ,\ implementationPath\ ,\ componentName\rangle$$

where *language, implementationPath* and *componentName* are String-typed fields.

A Component Implementation corresponds to a real implementation artefact for a component. The implementationPath must point to a file containing the Implementation's object code. It can be a Java class for a Java implementation, a C object code for C implementations, Code objects for Python Implementations or even code in Assembly language. From this object code, the framework will be able to estimate the worst case execution time for the provided services. The componentName field makes reference to the implemented Component.

URSO Definition 22 (Exported Service)**.** An *Exported Service* is formed by a reference towards a child *Component Service* or a child Composite *Exported Service.*

The Service referred by an Exported Service becomes visible to other Composites in the same level of the application architecture. This is implemented by multiple levels of service registries: A Composite $C_{n,\ Composite0}$[11] has an internal service registry in which

[11] This notation exprimes a Composite named *Composite0* at the n^{th}level in the application architecture.

all non-exported Services provided by its Components are published. When $C_{n,\ Composite0}$ exports a Component Service, this Service is seen as a Service provided by $C_{n,\ Composite0}$ itself; it is then visible to other Composites $C_{n,\ *}$[12] that are children of the same Composite $C_{n-1,\ Composite1}$. This Composite $C_{n-1,\ Composite1}$ may, in turn, export Services from the Composites that form it, making it visible for all Composites $C_{n-1,\ *}$ that are children of the same Composite $C_{n-2,\ Composite2}$ and so forth.
Hierarchical and *scoped service registries* were introduced by Cervantes and Hall along with the concept of *service-oriented component models* [Cervantes & Hall 2004].

URSO Definition 23 (Imported Dependency)**.** An *Imported Dependency* is formed by a reference towards a child *Component Dependency* or a child Composite *Imported Dependency.*

Similarly to Exported Services, Composites may look for other Composites' Services to satisfy a given Dependency. In order to do so, an Imported Dependency referring to that Dependency must be created, meaning the given Dependency is somehow promoted to be a Dependency of the parent Composite. A Composite can then query for Service Implementations for that Dependency in the service registry in which the other child Composites of its parent Composite (what may be considered as its "brother" Composites) have published their Exported Services.
The act of importing a Dependency or exporting a Service is called *Promotion*, because it *promotes* the Service or the Dependency so that it becomes a part of a higher-level composite. Figure 3.5 exemplifies the Service and Dependency promotion mechanism. Provided services are represented by circles, and required services are represented by half-circles of the corresponding color[13]. Inside the Composite C0, the Services S2 and S3 provided by the components A and B respectively are not promoted. Thus, they are published in the local service registry, where the Dependencies D1 and D2, that were not promoted either, can get bound to them. The Service S1 and the Dependency D3 were promoted; so they become part of the Composite C0 and they can be seen outside. The Composites C0 and C1 are both included in a bigger Composite named C2. The Service S1 provided by C0 is promoted inside the Composite C2; however, the Service S2 is not, so it is published inside Composite C2's service registry and is used to satisfy C0's Dependency D1. The Composite C2 promotes the Composite C1's Dependency D2 too. The Composites C2 and C3 are not included inside other composites; they are directly installed in the URSO Framework. Thus, their provided Services are published in the global service registry.

URSO Definition 24 (Composite)**.** A *Composite* is a 5-uple

$$\langle name\ ,\ components\ ,\ composites\ ,\ impDependencies\ ,\ expServices\rangle$$

where *name* is a unique String-typed identifier, *components* is a list of Components, *composites* is a list of Composites, *impDependencies* is a set of Imported Dependencies and *expServices* is a set of Exported Services.

In URSO, applications must be modelled as non-empty Composites (that means, components or composites lists must be non-empty) with a set of Promoted Services and References, in case the developer wants the Composites to interact with each other. Composites introduce a notion of scope in URSO. Services or Dependencies that are not explicitly imported or exported can not be seen by other Composites. However, it is not necessary for

[12] This notation designs all composites at the n^{th}level of the application architecture with any ID.
[13] This notation is based on UML's notation for implemented interfaces and will be used in this document to represent the service-based aspect in applicative systems.

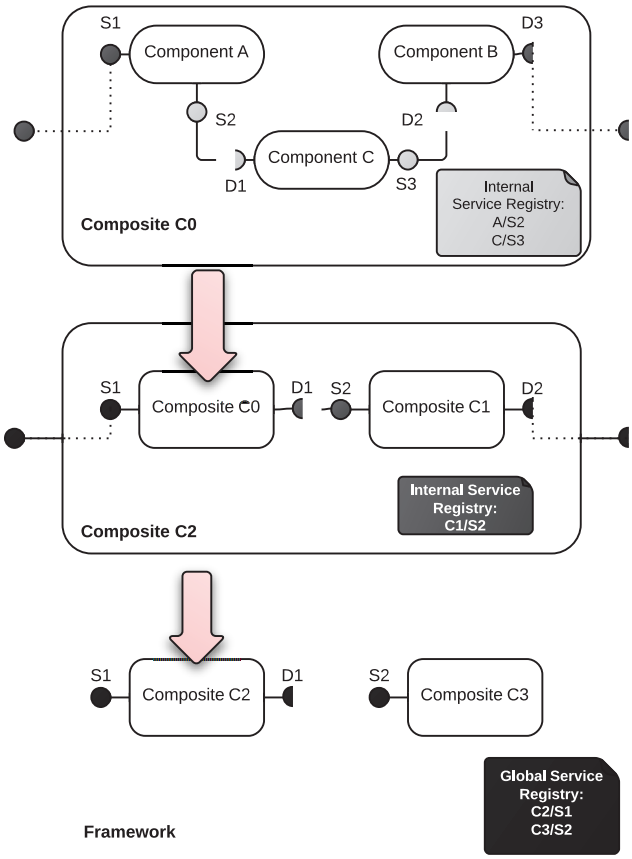

Figure 3.5: Service Registries in a Service-Oriented Hierarchical Component Model

a Composite to have an existence at run-time. Since URSO is a service-based model where Services are the only mean for entities to communicate, and Services follow the promotion and non-promotion rule for publishing and querying, there is no need to implement extra mechanisms to avoid Composites to interact with each other without the consent of the framework.

URSO Definition 25 (Service Composition)**.** A *Service Composition* is a 3-uple

$$\langle name\ ,\ composition\ ,\ usedServices \rangle$$

where *name* is a String-typed identifier, *composition* is a String and *usedServices* is a set of Imported Dependencies.

A Service Composition can be seen as a Composite which only has Dependencies towards other Services present in the global service registry. Service Compositions will be assigned to active (*i.e.* executable, with their own control flow) entities of our system.
A composition C can be created using the following grammar, which is based on a subset of the BPEL* language introduced and formalized by [Luo *et al.* 2008][14]

[14]BPEL was used as language reference due to the fact that it is was standardized and adopted by major IT industry vendors for describing business process as a service composition. Well before BPEL,

C ::= $\langle A\ ,\ F\rangle$
A ::= ε | A ; A | **if** b **then** A **else** A | $x := n$ | **invoke** $a\ x\ y$ | **throw** | A ‖ A | **wait** t_r | **waitUntil** t_a | **onEvent** e **do** A | **fireEvent** e | **while** b **do** A | **do** A n **times**
F ::= A

A Service Composition is composed by an Activity A and a Fault Handler F. In turn, Activities are composed by several sub-activities that can be grouped to express complex service composition behaviours. Whenever a **throw** activity is executed in a Service Composition, the control flow is asynchronously transferred to the group of activities indicated in the Fault Handler activity.
The ε is a NOP (No Operation) instruction - it corresponds to BPEL's *empty* activity. The "**;**" operator between two activities means that the referenced activities must be executed *sequentially*, that is, in A ; B, the activity B will be executed upon the termination of activity A. The **if** activity evaluates a boolean condition and, if its value is true, it executes the activity referenced after the *then* statement; else, it executes the activity referenced after the *else* statement. The "**:=**" operator is used to assign the value of the expression on its right side to the variables indicated on its left side. The **invoke** $a\ x\ y$ activity invokes the Service Operation indicated in a, passing x as input variable and receiving in y any possible return value. The "‖" operator is used to indicate parallelism between two activities - the framework is supposed to execute both concurrently and wait for the termination of both control flows. **Wait** and **waitUntil** allow developers to make the composition to pause its execution for a given duration, informed in relative time (*wait*) or until a given date, informed in absolute time (*waitUntil*). The **onEvent** e **do** A activity allows a service composition to wait until a given event identified by e occurs - then it will execute the activity informed after the **do** statement. **FireEvent** e is the activity used to fire the events that are handled by the *onEvent* activity. Finally, the **While** and **Do** n **times** are activities used to create indefinite and definite loops, respectively. All these activities correspond to real constructs in BPEL. A table showing the correspondences is presented in Table 3.1.
It is important for the framework to be able to bound the execution time of a service composition in order to perform scheduling analysis. Thus, it is important to estimate the worst case execution time for every service composition activity in our composition language.
The worst case execution time for a composition is bound by the sum of the execution time of both group of activities contained in a normal execution and in the fault handler. This situation can be seen in a composition whose last activity is a **throw**. Thus, for a Composition C = $\langle A\ ,\ F\rangle$, WCET(C) = WCET(A) + WCET(F).
The WCET for the ε activity is considered as negligible, since nothing is executed.
The WCET for the sequential activity composition A_1;A_2 is the sum of both WCET(A_1) and WCET(A_2), since both activities are executed.
The WCET from an **If** b **then** A_1 **else** A_2 activity is the sum of the time it takes to evaluate the boolean expression b plus the maximum WCET between both activities A_1 and A_2.
The WCET from an assignment "$x := n$" is bound by the time it takes to evaluate the expression n plus the WCET of the transfer of the number of bytes defined for the type of n to the position where x is stored in the memory.
The WCET from an **invoke** $a\ x\ y$ activity is defined by the sum of the transfer time for the in/out parameters x and y, and the WCET of the operation defined by a, which was estimated in the Behaviour concern and is present among the Service Operation parameters.

many other languages were already well known for describing processes, notably process algebras such as C.A.R. Hoare's Communicating Sequential Processes (CSP) [Hoare 1978] and Bergstra's Algebra for Communicating Processes (ACP) [Bergstra & Klop 1982].

Activity	BPEL-like Activity
ε	empty
$A_1;A_2$	<sequence><A_1><A_2></sequence>
if b then A_1 else A_2	<if><condition>b</condition><A_1><else><A_2></if>
$x := n$	<assign><copy><from variable="x"> <to variable="n"></copy></assign>
invoke a x y	<invoke operation="a" inputVariable="x" outputVariable="y"/>
throw	<throw />
$A_1 \parallel A_2$	<flow><A_1><A_2></flow>
wait t_r	<wait for="t_r>
waitUntil t_a	<wait until="t_a>
onEvent e do A	<onMessage partnerLink="e">><A></onMessage>
fireEvent e	<reply partnerLink="e"/>
while b do A	<while><condition>b</condition><A></while>
do A n times	<forEach><startCounter>1</startCounter> <finalCounter>n</finalCounter><A></forEach>

Table 3.1: Mapping towards BPEL activities

A **throw** activity just signalizes a control flow transfer thus its execution time is negligible. The WCET of the parallel composition activity $A_1 \parallel A_2$ actually depends on the platform thread scheduling policy. In the worst case, we may consider that both threads must share a processing unit and that the scheduler has decided to first execute a complete activity before executing the other (for instance, using a run-to-completion [Chiang *et al.* 1994] scheduling algorithm). In that case, the parallel composition works just as a sequential composition, and its WCET is defined by the some of the WCET of both activities A_1 and A_2.

The **wait** t_r activity establishes a relative time during which the framework must wait. The WCET of this activity is thus t_r. The **waitUntil** t_a activity provides an absolute wait time. If this absolute time is ahead of the current time, then the WCET is defined by the difference between t_a and the current time. Else, this instruction is ignored and its WCET is negligible.

Firing an event by means of a **fireEvent** e_1 activity requires the framework to look for all the subscribers of the event e_1 and to inform them all that their execution can be carried on (the event firing is not kept in memory, so it is cleared as soon as all its handlers are executed). That mechanism can be implemented by means of special partner links based on event names and asynchronous messages sending. So the WCET of this instruction depends on the number of subscribers for the event e. Calculating the WCET for the **onEvent** e_2 **do** A activity is even more complicated: the service composition is registered in memory along with the event it handles, and then its execution is suspended. Besides calculating the WCET of the activity A, it is not possible to foresee when the event e_2 will be fired. For both cases, a timeout must be stipulated. Thus we consider that the platform has an environment variable *activityTimeout* which informs how long the platform should wait for the completion of that activity. If the execution of the activity exceeds the timeout defined by the platform, its execution is interrupted and a throw activity is automatically executed, transferring the control flow to the fault handler. So for both activities, *activityTimeout* is the WCET. The same strategy is used to bound the WCET in the **While** b **do** A activity. Instead of defining a timeout, we use another environment variable called *activityMaxLoop* and we keep count of the number of times the loop is executed. The WCET in that situation is then the maximum value between *activityMaxLoop* * WCET(A) and *activityTimeout*.

A '**do** A n **times**' activity works just like a 'for' instruction in imperative programming languages: n is evaluated and the resulting value is used to bound the number of times the loop is executed. Thus, its WCET is the sum of the evaluation time for the expression n plus WCET(A) plus the WCET of the instruction used to update the counter variable multiplied by the maximum value between n and *activityMaxLoop*.
The WCET of the service composition activities is summarized in Table 3.2. [15,16,17]

Activity	**Estimated Activity WCET**
ε	0
A_1;A_2	WCET(A_1) + WCET(A_2)
if b then A_1 else A_2	eval(b) + max(WCET(A_1), WCET(A_2))
$x := n$	eval(n) + transferTime(n)
invoke a x y	transferTime(x) + WCET(a) + transferTime(y)
throw	0
$A_1 \parallel A_2$	WCET(A_1) + WCET(A_2)
wait t_r	t_r
waitUntil t_a	if $t_a >$ current time, t_a - current time; else, 0
fire e	*activityTimeout*
onEvent e do A	activityTimeout + WCET(A)
while **b** do A	activityMaxLoop * (eval(b) + WCET(A))
do A n times	eval(n) + max(n, *activityMaxLoop*) * (WCET(A) + update(counter))

Table 3.2: Estimated WCET for Service Composition Activities

3.2.2.5 Assembly concern: UML Diagrams

A UML Class diagram depicting the concepts of the Assembly concern is presented in Figure 3.6. As mentioned in the previous section, an application developer must design its application in terms of Services, Components, Composites and composites of Composites. These Services are going to be used in Service Compositions, which represent business functions. Since Implementation artefacts are linked to Components, only Components may implement Services and require Dependencies. Implementations are related to the Deployment concern due to the fact that the framework must know its requirements in terms of Resources (that is the reason why the Component Implementation class has a yellow projection behind it - yellow is the color used in the Deployment concern class diagram).
Composites can promote Components' Services and Dependencies in order to make it visible to other Composites; otherwise, the Services are only seen by other Components (or Composites) in the same Composite, and only these Services can be used to satisfy non-promoted Dependencies. Services and Dependencies are linked by Bindings, which are associated to Communication Protocols. These protocols are the same that were described in the Deployment concern. In order for a Service to match a Dependency three criteria must be met:

- The set of operations of the Dependency Service Description must be included in the service provider's Service Description.

[15] In the table, we call *eval(n)* the time needed to evaluate an expression. A timeout can be stipulated for the expression evaluation, just as for unbound loops.
[16] Where *transferTime(n)* = time to transfer a byte * number of bytes indicated as size in the type descriptor for the type of n
[17] *Update(n)* is the function that updates/increments the counter variable of a definite loop or iterator.

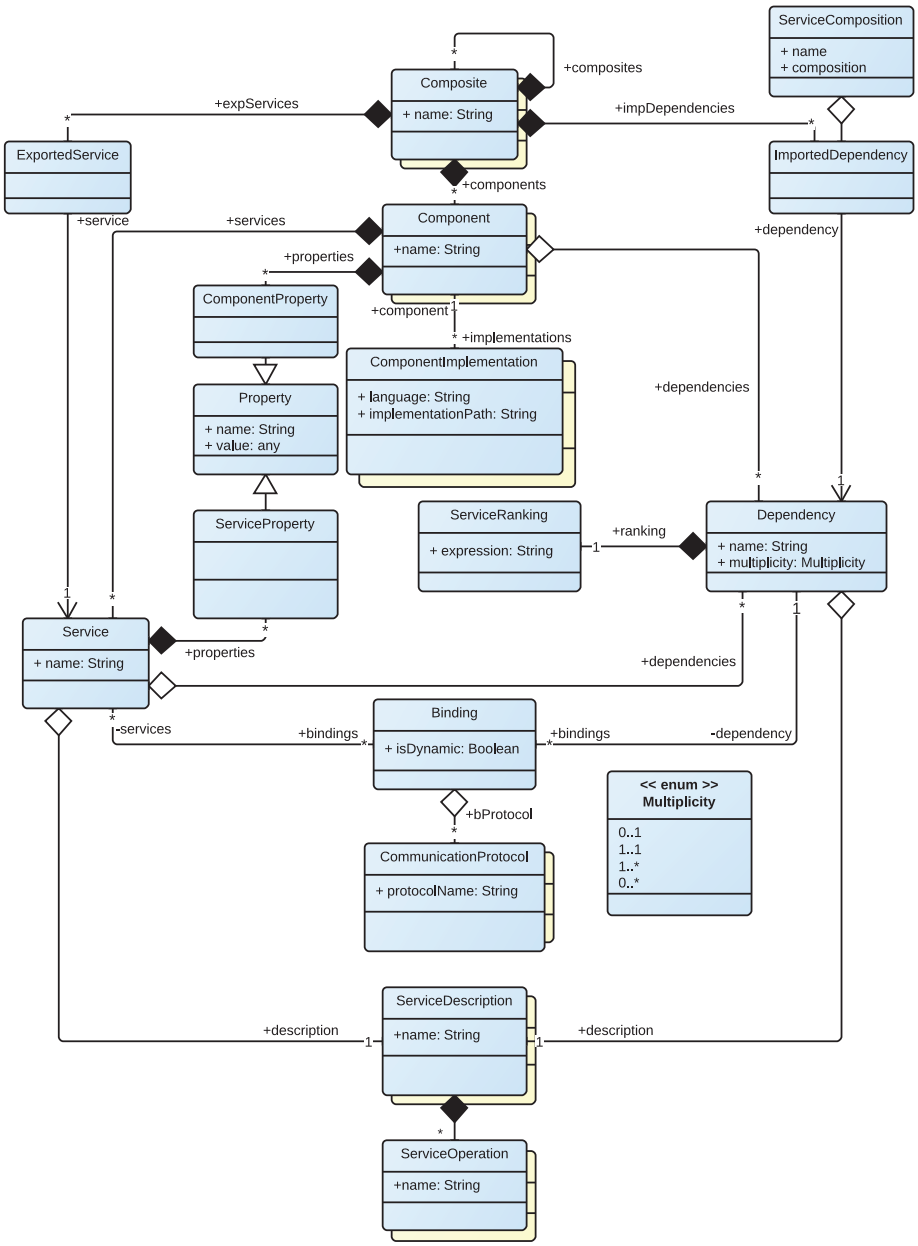

Figure 3.6: URSO Component Model - Assembly concern

- There must exist at least one common Communication Protocol between the Service and the Dependency Bindings.
- If the Dependency has a Service Ranking function, and this function uses a filter operator, the evaluation of the Service Ranking expression for the service provider properties must result **true**.

Service Descriptions are made of Service Operations. These operations have a name, a list of Instructions which is automatically retrieved by the framework, input and output

parameters and restrictions associated to them. Service Operations will be better discussed in the Behaviour concern section.
Service Compositions are used to aggregate Dependencies towards Services to perform a business function. Just as in its Component counterpart, Service Compositions are able to filter the Services Implementations used in its invoke activities.
The UML Activity diagram in Figure 3.7 shows a part of the process of creating applications using the URSO metamodel. The definition of real-time tasks was not detailed, as it is related to another concern. This will be further discussed when the complete Deployment concern is presented. As stated before, the developer may independently develop Service Descriptions, Component descriptions and Service Compositions. A Component Implementation depends on the Component contract it must respect and on the Service Descriptions of the Services it provides or requires. Services Descriptions are also inputs for defining Service Compositions, which can be associated to Business real-time Tasks. Business Tasks will be discussed when we revisit the Deployment concern later on.

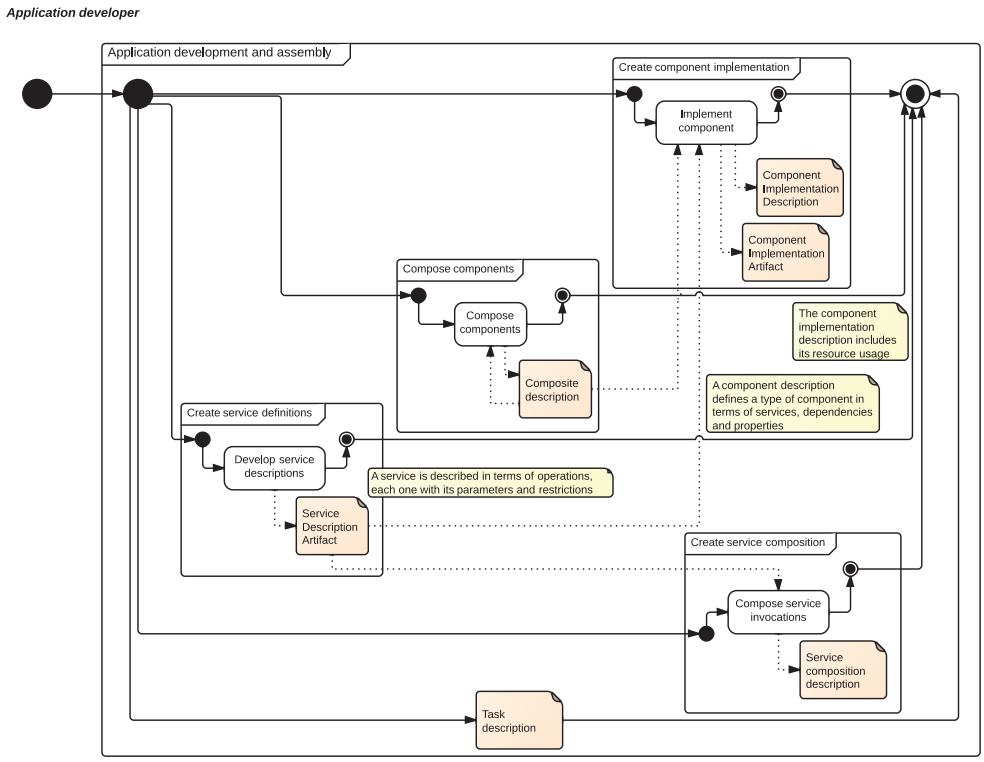

Figure 3.7: URSO Component Model - Component Development

3.2.3 Behaviour Concern

The Behaviour concern is composed of concepts related to the real-time characterization of Component Implementations and their Service Operations. This layer will also include concepts which are more platform specific, such as execution times and low-level instruction sets.

3.2.3.1 Service Operations, Parameters, Restrictions and Instructions

Service Operations are the main part of Services. They correspond to callable units, such as procedures, methods and functions, in different programming languages. Service Operations may contain parameters, that is, variables used to input and output values. These parameters have a type and may have restrictions associated to their value before, during or after the operation call. The list of Service Operation instructions corresponds to its implemented code in an assembly-like language. Each language has its different set of instructions, but some instruction types are present in all languages. The framework inspects the implementation object code and from that it constructs an oriented graph representing the possible control flows paths and their instructions. Then, if the platform knows the execution time of each instruction in a given infrastructure, it is possible for the framework to estimate a *partial* worst case execution time. This execution time is said to be partial because the Service Operation code may contain Service calls, and the framework does not know before the execution which Service Implementation is going to be used by this Operation. Once it has resolved all Component Dependencies, it may then estimate the total worst case execution time, which will be the path in the control flow graph that results in the highest execution time.

3.2.3.2 More about Policies

Policies are actions which are automatically taken by the platform when activated by a given event. This event can be specified by means of Rules, which are logical expressions over variable values. This variable may be a Service Operation Parameter or a framework environment variable. When Policies are linked to Restrictions, the Restriction expression is used as a rule for the Policy activation. Actions are defined by commands. The framework must be able to parse these commands and execute them. Thus, it is recommended for the framework developer to specify environment variables and standard commands (for stopping components, composites, or the whole platform, for instance). Policies were already defined and presented in the Deployment concern section.

3.2.3.3 Definitions

URSO Definition 26 (Parameter)**.** A *Parameter* is a 2-uple

$$\langle name\ ,\ type \rangle$$

where *name* is a unique String-typed identifier, and *type* is a Type.

A type must be described in the platform beforehand. A type description contains information such as the type name and the size in bytes of the objects in this type. There are two categories of Parameters: Inputs and Outputs.

URSO Definition 27 (Restriction)**.** *A Restriction is a 3-uple*

$$\langle name\ ,\ expression\ ,\ policies \rangle$$

where *name* is a unique String-typed identifier, *expression* is a String and *policies* is a set of Policies.

A Restriction is a special type of Rule. There are three different categories of Restrictions: Preconditions, Invariants and Postconditions. Preconditions predicates are evaluated before the execution of the Service Operation, whereas Postconditions are evaluated after its execution. If a Pre or postcondition predicate indicated by the field *expression* is evaluated

to false, the Actions expressed by the Policies contained in the set *policies* are executed. Invariants are predicates which are evaluated during the execution of a Service Operation. Like pre and postconditions, if the Invariant expression is evaluated to false during the execution of the Service Operation, the bound Policies' Actions are executed. For performance issues, the framework developer may decide to evaluate the Invariant expression before the Service Operation call and then periodically during its execution; he may also consider that if the condition is false in the beginning of the execution, then its value does not need to be monitored (like the evaluation of an implication statement - if *expression* was true then *expression* remains true).

URSO Definition 28 (Instruction)**.** An *Instruction* is a 2-uple

$$\langle \underline{name} \ , \ \underline{duration} \rangle$$

where *name* is a String and *duration* is an Integer. Both values are automatically assigned by the platform.

Instructions are abstractions for the low-level object code instructions. They can then be grouped into different categories, depending on their functionality. Despite the fact that the object code instructions set may vary a lot from technology to technology, three types of instructions are very often present:

1. Arithmetic instructions, *e.g.* mathematic and bitwise operators;
2. Jump type instructions, *e.g.* conditional and unconditional control flow transfer; and
3. Memory manipulation instructions, *e.g.* value storage and retrieval in the computer memory.

Figure 3.8 shows a taxonomy of Java byte-code instructions set. Many of the features present in the Java programming language are visible in its byte-code instruction set (*e.g.* exception handling, object-orientation and synchronization over attributes and methods). In addition, due to the fact that the Java byte-code computation model is stack-oriented, instructions to manipulate the stack are provided as well. The list of instructions from a Service Operation is automatically created by the framework. After the framework starts, but before the installation of business components, the platform administrator must use a sample program in order to allow the platform to estimate the worst case execution time for all low-level instructions in that hardware infrastructure. This value is then used to fill the field *duration* of Service Operation Instructions.

URSO Definition 29 (Service Operation)**.** A *Service Operation* is a 7-uple

$$\langle name \ , \ maxLoops \ , \ maxRecDepth \ , \ restrictions \ , \ parameters \ , \ \underline{instructions} \ , \ \underline{RTParams} \rangle$$

where *name* is a unique String-typed identifier, *maxLoops* and *maxRecDepth* are integers, *restrictions* is a set of Restrictions, *parameters* is a list of Parameters, *instructions* is a list of Instructions automatically filled by the framework and *RTParams* is a Real-Time Operation Parameter automatically managed by the framework as well.

MaxLoops and *maxRecDepth* are variables in which the application developer informs the maximum number of loops and maximum depth recursion in the Operation. These values help in the estimation of the worst case execution time (WCET). The *RTParams* attribute contains data used for the WCET of the whole Service Operation and is automatically updated upon Service provider changes.

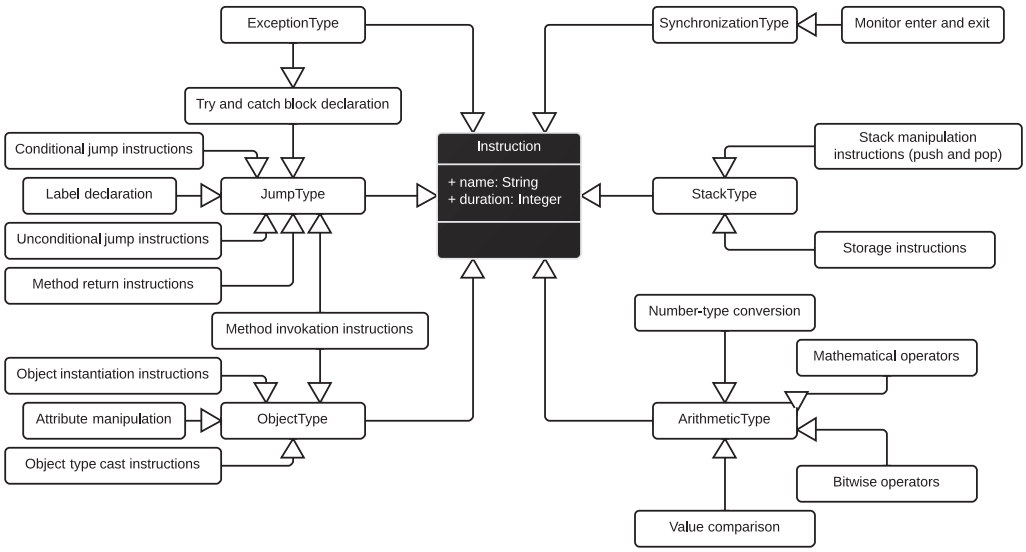

Figure 3.8: Taxonomy of Java byte-code instructions

URSO Definition 30 (Real-time Operation Parameter)**.** A *Real-time Operation Parameter* is a 2-uple

$$\langle \underline{partialWCETExpression} \ , \ \underline{totalWCET} \rangle$$

where *partialWCETExpression* is a String and *totalWCET* is an Integer. Both attributes are automatically managed by the framework.

Real-time Operation Parameters contain data used for the framework to recursively estimate the WCET of complete applications. The *partialWCETExpression* is calculated based on the Service Operation Instructions and the information about loops and recursion maximum depth. It represents a function depending on Service calls. Once the Component containing the Service Operation has its Dependencies resolved, this function is evaluated based on the *totalWCET* of the consumed Service Operations.
The calculation of the *partialWCETExpression* is performed in two steps. First the framework transforms the list of Instructions into a control flow graph. The edges of this graph are weighted with the sum of the duration of Instructions that are executed sequentially. These edges are also labelled with one of three possible values:

- *DIRECT*, if the sequence of Instructions that follows can be achieved without passing by a control transfer instruction;
- *IF*, if the sequence of Instructions that follows is achieved through a conditional control transfer instruction; and
- *GOTO*, if the sequence of Instructions that follows is achieved through an unconditional control transfer instruction.

An algorithm to obtain the control flow from a sequence of Java byte-code instructions is presented in pseudo-code in the next pages. The structures used to keep information in this algorithm are presented in the Algorithm 3.1.This algorithm targets Java byte-code and its stack-oriented programming model, but it can be easily modified to target object code with

less complicated programming models.
Each node in the control flow graph contains a name, which corresponds to the name of the label it represents in the object code. Nodes also contain a set of parent nodes' name, that is, the name of the nodes that can reach it in the direct graph. It is important to keep that list in a data structure, as it will help the algorithm to identify loops in the object code. It knows that a node is inside a loop when the set of parent nodes' names includes its own name. Nodes also contain a list of edges towards other nodes. In turn, an edge is characterized by the node it is pointing to, an edge type (that can be DIRECT, GOTO or IF) and a cost. Nodes also contain a flag that indicates whether it is inside a loop or not. If they are, another variable indicates the depth of the loop it is contained in (for instance, a node outside a loop would have its depth variable set to zero, whereas a node inside a 'for' instruction block would have it set to one). Nodes contain a list of instructions which are sequentially executed in the code. These instructions will be used to estimate the cost of the node's edges.

Algorithm 3.1. Data structures in Byte-code Control Flow Graph

```
Data Structure CFNode {
  String name {Node's name, based on the label name in the code}
  Set of Strings parents {Node's parent nodes names}
  List of Edges edges {Node's edges}
  Integer depth {The depth of the node in the code}
  Boolean insideLoop {Indicates whether the node is inside a loop or not}
  List of instructions instructions {A list of Instructions that is executed sequentially}
}

Data Structure Edge {
  CFNode node {Next node}
  Link type type {Edge label}
  Integer cost {Edge cost}
}

Enumeration Link Type {
  DIRECT, IF, GOTO
}
```

The algorithm assumes that the object code language uses labels to refer to program points where the control flow can be transferred. In languages whose jump instructions allow indirect jump instructions (like jumping to the instruction which lies in the address pointed by a variable) it is not possible to establish all the edges between nodes beforehand.
The algorithm snippet presented at the Algorithm 3.2 shows the structure of the graph construction algorithm. After an initialization phase, the algorithm wanders the list of instructions of a program. Three special types of instructions are considered by the algorithm for the graph construction: label declarations, which add a new node to the graph; labelled-jump instructions, that is, instructions that indicate a label to jump to, given a certain condition; and return instructions, which terminate the program execution, returning a value or not. Other types of instructions are added to the currentNode instructions list in order to calculate its cost. The only type of instruction that is not taken into account to estimate execution costs is the label declaration type, since their only function is to associate a name to a given place in the object code.

The algorithm snippet in the Algorithm 3.3 details the treatment of label declaration in-

Algorithm 3.2. Byte-code Control Flow Graph Construction Structure

Input: List of instructions *program* = $[i_0\ ,\ i_1\ ,\ \ldots\ ,\ i_n]$
Output: A CFNode *graph*

```
{Initialization}
graph ← ∅
index ← 0
currentLevel ← 0
initialNode ← ⟨"l_α" , ∅ , ∅ , currentLevel , false , ∅⟩ {CFNode tuple representation}
graph ← initialNode
currentNode ← initialNode
ifInstruction ← false

{The algorithm wanders the instruction list}
for all instruction i in program do
  if i is a label declaration then
    {A label in the program means the creation of a new node}
    * Treat the label declaration case *
  else
    {It is not a label declaration instruction}
    Add i to currentNode.instructions
    if i is a Jump Instruction pointing to label l_i then
      {It may be an IF, GOTO or JSR}
      if i is GOTO or i is JSR then
        * Treat the unconditional jump case *
      else
        {It is an IF instruction}
        * Treat the conditional jump case *
      end if
    else if It is a RETURN instruction then
      {Returns are unconditional jumps towards the final state}
      * Treat the return case *
    end if
  end if
end for
```

structions.
The processing considers two cases: whether there is a node currently being analysed or not. If existent, it means that the previously analysed node reaches the new node without a jump instruction: nodes are linked by what we called a DIRECT-type edge. The new node may already be present in the graph if it has already been referenced by another node's jump instructions; in this case, its edges list is updated. If it was not present, a new node is created with the DIRECT edge. If the instruction before this node declaration was an IF instruction, this new node is actually one of the branches of a conditional jump, thus its depth level (and the whole analysis depth level) is set to the current code depth level plus one. The new node's set of parents (which are now the anciently current node plus the current node's parents) and looping flag (after verifying if the node is reached by itself) are updated, before setting the new node as the analysis current node.
If there was not a current node being analysed, this means that the new node is either the

first node of the analysis or a label declared after an unconditional jump - like a GOTO instruction - that is used as a target for a jump instruction (else it would be considered as dead code). If the label is already present in the graph, the analysis updates its code depth level; if it is not, a new node with the name of the new label is added to the graph. Either way, the current node variable is updated to point towards the new node.

Algorithm 3.3. Byte-code Control Flow Graph Construction - Labels

```
if i is a label declaration then
  {A label in the program means the creation of a new node}
  if currentNode ≠ ∅ then
    currentParents ← currentNode.parents ∪{currentNode.name}
    {The parents of the new node will be the set of parents of the ancient current Node
    plus the current Node itself}
    if there is a CFNode whose name is i already in the graph then
      currentNode.edges ← currentNode.edges ∪ ⟨i , DIRECT⟩
      currentLevel ← i.level
    else
      {This is the other branch of an IF instruction.}
      if ifInstruction is set to true then
        currentLevel ← currentLevel + 1
        ifInstruction ← false
      end if
      graph ← graph ∪ ⟨i , ∅ , ∅ , currentLevel , false , ∅⟩
      Add ⟨i , DIRECT⟩ to currentNode.edges
    end if
    currentNode ← node in graph whose name is i
    currentNode.parents ← currentParents
    if Node whose name is i is in currentParents then
      Set the flag insideLoop of the node whose name is i to true
    end if
  else
    {Empty currentNode means that the algorithm is parsing its first node or an isolated
    node used by a jump instruction}
    if there is a CFNode whose name is i already in the graph then
      currentLevel ← i.level
    else
      graph ← graph ∪ ⟨i , ∅ , ∅ , currentLevel , false , ∅⟩
    end if
    currentNode ← node in graph whose name is i
  end if
else
  {It is not a label declaration instruction}
  * Other instruction types' treatment *
end if
```

Jump instructions treatment is detailed in the Algorithm 3.4. First the algorithm verifies if there is a node being analysed. Any instruction must belong to a node in the control flow graph - if it does not, this instruction is unreachable and the algorithm indicates an error in the analysed program.
The algorithm then differentiates the treatment for jump instructions whose label is indicated (like GOTOs or IFs) and for RETURN-like instructions, which can be seen as a

jump towards a final node/state. For Jump instructions, unconditional jumps (GOTOs and JSRs[18]) are processed differently from the conditional ones (IF-like instructions).

Algorithm 3.4. Byte-code Control Flow Graph Construction - Jumps

```
if i is a label declaration then
  {A label in the program means the creation of a new node}
  * Treat the label declaration case *
else
  if currentNode ≠ ∅ then
    Add i to currentNode.instructions
    if i is a Jump Instruction pointing to label l_i then
      {It might be an IF<condition>(l_i), GOTO(l_i) or JSR(l_i)}
      currentParents ← currentNode.parents ∪{currentNode.name}
      if i is GOTO or i is JSR then
        if there is not a node whose name is l_i in graph then
          Add ⟨l_i , ∅ , ∅ , currentLevel , false , ∅⟩ to graph
        end if
        Add ⟨l_i , GOTO⟩ to currentNode.edges
        currentNode ← ∅
        currentLevel ← 0
      else
        {It is an IF instruction}
        ifInstruction ← true
        if there is not a node whose name is l_i in graph then
          Add ⟨l_i , ∅ , ∅ , currentLevel + 1 , false , ∅⟩ to graph
        end if
        Add ⟨l_i , IF⟩ to currentNode.edges
      end if
      Add currentParents to the field parents from the graph node named l_i
      if there is a node named l_i in currentParents then
        Set the flag insideLoop from the graph node named l_i to true
      end if
    else if i is a RETURN instruction then
      {RETURN instructions point towards the final node}
      Add ⟨l_ω , DIRECT⟩ to currentNode.edges
      currentNode ← ∅
      currentLevel ← 0
    end if
  else
    Error {Unreachable instruction!}
  end if
end if
```

In the case of unconditional jumps, if there is not a node corresponding to the target label in the graph, it is created and added. The node that is currently being analysed receives a new edge, a GOTO-typed one, towards the node corresponding to the target label. The algorithm then resets the currentNode to none and the currentLevel to zero, since one graph

[18]JSR (Jump to Sub Routine) is an instruction indicating jump to a subroutine. JSR and RET (which is used to return from a subroutine) where deprecated in Java 6 due to the complexity it added to byte-code verification (some normal byte-code constraints, such as stack consistency, were relaxed for these instructions).

path ends with this instruction.
In the case of conditional jumps, a flag indicating the execution of an IF instruction is set by the algorithm. This flag is used to notify the algorithm that the label declaration following this instruction indicates one of the branches of an IF instruction (the branch that is executed if the condition specified in the IF instruction is not satisfied). Just as in its unconditional jump counterpart, if there is not a node corresponding to the target node in the graph, a new one is created and added; however the code depth level of this new one is set to the current code depth level plus one. In addition, a new IF-typed edge is added to the list of edges of the node that is currently being analysed.
Independently of the instruction jump type, the set of parents of the node corresponding to the target label is updated with the set of parent nodes of the current node plus the current node itself. A test to see whether the target label node is in a loop is then carried out.
In the case of RETURN-type instructions, a DIRECT-type edge towards a special node indicating the end of the program execution is added. The variables indicating the current code depth level and the node currently being analysed are reset. RETURN instructions may be present anywhere at the program, and just as the GOTO instructions they mean the end of a graph path.
After wandering the instruction list, some finalizations must be performed by the algorithm. It must verify the whole graph to recursively update the set of parent nodes for nodes which were marked as part of loops. It also verifies that the program terminates (that is, the final state is reachable from the initial state). If so, the algorithm returns the program control graph; if not, it returns an empty value, to indicate an error. The pseudo-code corresponding to these finalizations is shown in the Algorithm 3.5. The auxiliary functions used to update the node parents and edges and to verify reachability are presented in the algorithm snippets 3.6 and 3.7. After creating the control flow graph, it is simple to estimate the

Algorithm 3.5. Byte-code Control Flow Graph Construction - Finalization

```
for all CFNode node in graph do
  if node.insideLoop is true then
    updateLinks(node, node.parents)
  end if
end for
if canReach(l_α, l_ω) then
  Return graph
else
  Return ∅
end if
```

partial WCET: since the worst case execution time of each low-level instruction has been estimated beforehand, the cost of an edge is the sum of the WCET of the instructions from its origin graph node. The WCET is considered as the maximum cost from a possible path from the initial node to the final node. The operation may however execute calls towards other operations: in this case, if the code of the called operation is available, its behaviour and WCET is estimated as well; else, if it is, for instance, a call towards a Service Dependency, then a variable is added to the WCET and a function expression (depending on the Service Dependencies) is stored. This expression is evaluated upon the resolution of the Component Implementation and then the total WCET can be estimated (but the expression is still stored, due to the dynamic availability of Component Implementations and the fact the Service Dependency can be satisfied by many Service providers during an execution).

Algorithm 3.6. Byte-code Control Flow Graph Construction - Edge Update

```
Input: a CFNode node and a set of CFNode parents
Output: a CFNode updatedNode
  if parents ≠ ∅ then
    for all Edge edge in node.edges do
      if edge.node.parents do not contain all node names from parents then
        for all parentName in parents do
            edge.node.parents ← edge.node.parents ∪ parentNameif parentName =
            edge.node.name then edge.node.insideLoop ← true
          end if
        end for
        if edge.node.insideLoop is true then
          updateLinks(edge.node, edge.node.parents)
        end if
      end if
    end for
  end if
  Return node
```

Algorithm 3.7. Byte-code Control Flow Graph Construction - Reachability

```
Input: CFNode node1, CFNode node2, set of CFNodes visitedNodes
Output: A boolean value
  if node1 = node2 then
    Return true
  else
    result ← false
    if visitedNodes = visitedNodes ∪ node1 then
      Return false
    else
      for all Edge edge in node1.edges do
        result ← result OR canReach(edge.node, node2, visitedNodes ∪ node1);
      end for
    end if
    Return result
  end if
```

3.2.3.4 Behaviour Layer: UML Diagram

Figure 3.9 presents an UML class diagram relating the behaviour layer concepts. The central concept is that of Service Operation, which is also present in the Assembly Layer, just as Service Descriptions. The set of instructions contained by the Service Operations is platform-dependent, which is symbolized by the black class shadow behind the Instruction class. The Restriction class, which represents semantic restrictions on Service Operations, is a class derived from the Deployment Layer's Rule class, and just as its parent class, is associated to a Policy. Timing constraints are addressed by the class RT Operation Params, which represents the worst case execution time of a Service Operation. The Type Description class is used to estimate worst case execution time as well: for instance, it helps estimating the time it would take to transfer data from a component to another.

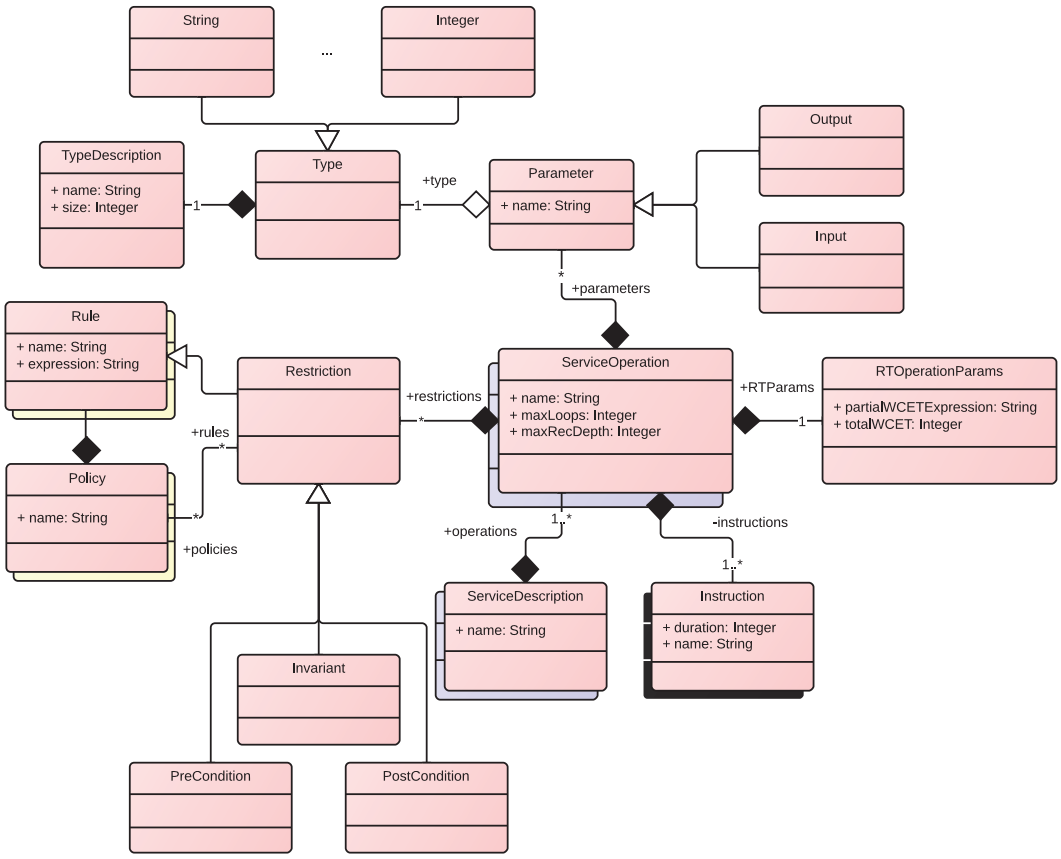

Figure 3.9: URSO Component Model - Behaviour Layer

3.2.4 Revisiting the Deployment Concern

Now that the Assembly and Behaviour concerns were presented, it is possible to refine the Deployment concern to include concepts related to the mapping of applicative components on real infrastructure nodes and resources. These new concepts allow the definition of schedulable real-time tasks, which in turn correspond to a composition of Services and Service calls. Implementations will also be associated to their consumed resources, enabling the platform to implement resource reservation and admission control mechanisms to better improve execution time determinism.

3.2.4.1 Resource Requirements and Capabilities

The Platform contains nodes which are listed as its Capabilities. These Resource Capabilities are shared or divided by the Platform Nodes and activated or not in the different Platform Modes. Software artefacts, like Component Implementations, are going to use these Resources in their execution. These dependencies towards material Resources are called Resource Requirements. In the case of Shared Resources, software artefacts may require Resources as writers (*i.e.* can carry modifications to the Shared Resource), readers (that is, can use a given Resource but they cannot modify it) or both (*i.e.* can read and update the properties from a given Resource). Software artefacts may declare Resource

capabilities as well (*e.g.* files or software libraries).

3.2.4.2 Implementation, Instances and Nodes

A given application may have several copies of a given Composite, due to performance (parallelism, locality factor) or fault tolerance (replication) issues. These several copies can be seen as several Instances of the Component Implementations of a same Composite. An Instance will be associated to a platform Node and will have the same Resource Requirements as the Component Implementations it is bound to. Based on the Implementation's Requirements and on the Node's available Resources, the Platform may then perform automatic Instance allocation or block the placement of an Instance in a given Node due to lack of available Resources.

3.2.4.3 Tasks and Mode Transitions

Traditionally, in real-time systems, applications are modules as a set of tasks, with timing constraints or not, that may have an activation frequency and other timing properties associated to them. In URSO, a task is associated to a Service Composition, which, as described in the Assembly concern, allows structuring a set of Service Operation calls to realize a business function. As depicted in the previous section, it is possible for the Platform to estimate the worst case execution time for Service Operations. Considering that the Platform can also estimate the communication time (based on the Interconnections, on the Communication Protocols timing properties and on the type size informed in the Type Description), the worst case execution time of a Service Composition, and consequently, that of a Task can be estimated. Scheduling analysis can be performed on the set of tasks in a Node in order to verify the feasibility of the placement informed by the application developers that of the application as a whole.
The developer can assign a mode transition behaviour to a task. That means that upon a Platform Mode change, the task can interrupt itself and/or trigger a new task. That would allow, for instance, a real-time task to execute in a degraded mode in non-critical Platform modes. Tasks may also indicate dependencies towards other tasks. That way, a given task will not execute until all the depended tasks are executed.

3.2.4.4 Definitions

URSO Definition 31 (Implementation (revisited))**.** An *Implementation* is a 5-uple

$$\langle language\ ,\ filename\ ,\ componentName\ ,\ capabilities\ ,\ requirements\rangle$$

where *language, filename* and *componentName* are Strings, *capabilities* is a set of Resource Capabilities and *requirements* is a set of Resource Requirements.

The concept of Implementation was updated to inform the Resources provided (like applicative software libraries) and consumed (such as RAM memory).

URSO Definition 32 (Instance)**.** An *Instance* is a 3-uple

$$\langle \underline{name}\ ,\ composite\ ,\ mappedNode\rangle$$

where *name* is a unique String-typed identifier automatically assigned by the platform, *composite* is a Composite and *mappedNode* is a Platform Node.

An Instance is bound to a Composite. It actually corresponds to the instantiation of all Component Implementations necessary to the instantiation of the Composite. It is uniquely identified by a name that is automatically generated by the framework and it is mapped onto a Platform Node. This Node is informed by the Application administrator, which can find an effective placement before adding the Component Implementations' Instances to the framework by executing off-line placement algorithms based on the current Platform configuration.

URSO Definition 33 (Task)**.** A *Task* is a 6-uple

$$\langle name\ ,\ composition\ ,\ mappedNodes\ ,\ properties\ ,\ transitions\ ,\ requiredTasks\rangle$$

where *name* is a String-typed identifier, *composition* is a Service Composition, *mappedNodes* is a set of Nodes, *properties* is a set of Real-time Properties, *transitions* is a set of Mode Transitions and *requiredTasks* is a list of Tasks.

A Task is the schedulable representation of a business process, which in turn is defined by a Service Composition. Tasks can be mapped onto one or several Platform Nodes. Just as Instances, this placement can be performed off-line by the application administrator. Tasks will be characterized by real-time properties, such as deadline, release date and priority. There two special types of Tasks: Periodic Tasks and Sporadic Tasks. As the name suggests, Periodic tasks execute periodically often, whereas Sporadic Tasks execute with a minimum inter-arrival scheduling time. Both have special properties: Periodic tasks contain a period attribute, and Sporadic tasks contain an attribute that specify their minimum inter-arrival time. These properties are used by the framework to create and schedule real-time threads and to perform scheduling analysis. It is worth to mention that all thread creation in the applicative code is forbidden. The framework verifies at deployment time whether the code tries to instantiate code and, if possible, it removes these instructions. It is also important to note that tasks do not depend on Components. Neither do Service Compositions, which only rely on Service Descriptions and their Service Operations. Both can be deployed separately from Implementation artefacts and Composite descriptions.
Tasks may depend on the execution of other threads. For instance, a developer in a business application may express that the payment task always comes after the execution of the authentication task. In URSO, a given task will not be executed unless all the tasks specified in the *requiredTasks* set are executed.
In addition, it is also possible to inform mode change transitions for threads by means of mode transitions. As we will see in the next definition, a Mode transition will be formed by a Platform mode and the new replacement task. Tasks which do not inform mode transitions are able to execute in all Platform Modes.

URSO Definition 34 (Mode Transition)**.** A *Mode Transition* is a 2-uple

$$\langle ancientMode\ ,\ newMode\rangle$$

where *ancientMode* and *newMode* are Platform Modes.

Mode Transitions allow tasks to be interrupted and started automatically upon Platform Mode changes. It is recommended that these changes are controlled by the platform administrator, but some rules for automatizing mode changes may be created in order to ease the administration task. How tasks are interrupted and the transition duration (time during which the executing task set is changed and that both ancient and new Platform Modes are updated) depend on the platform implementation. Mode transition protocols for real-time systems is a well-established research domain in computer science [Real & Crespo 2004].

3.2.4.5 Deployment Concern: UML Diagrams

The UML Class Diagram presented in Figure 3.10 adds the concepts presented above to the Class diagram presented in section 3.2.1. The classes corresponding to Mode Transition, Tasks and real-time tasks' properties, Service Compositions and Instances were included.

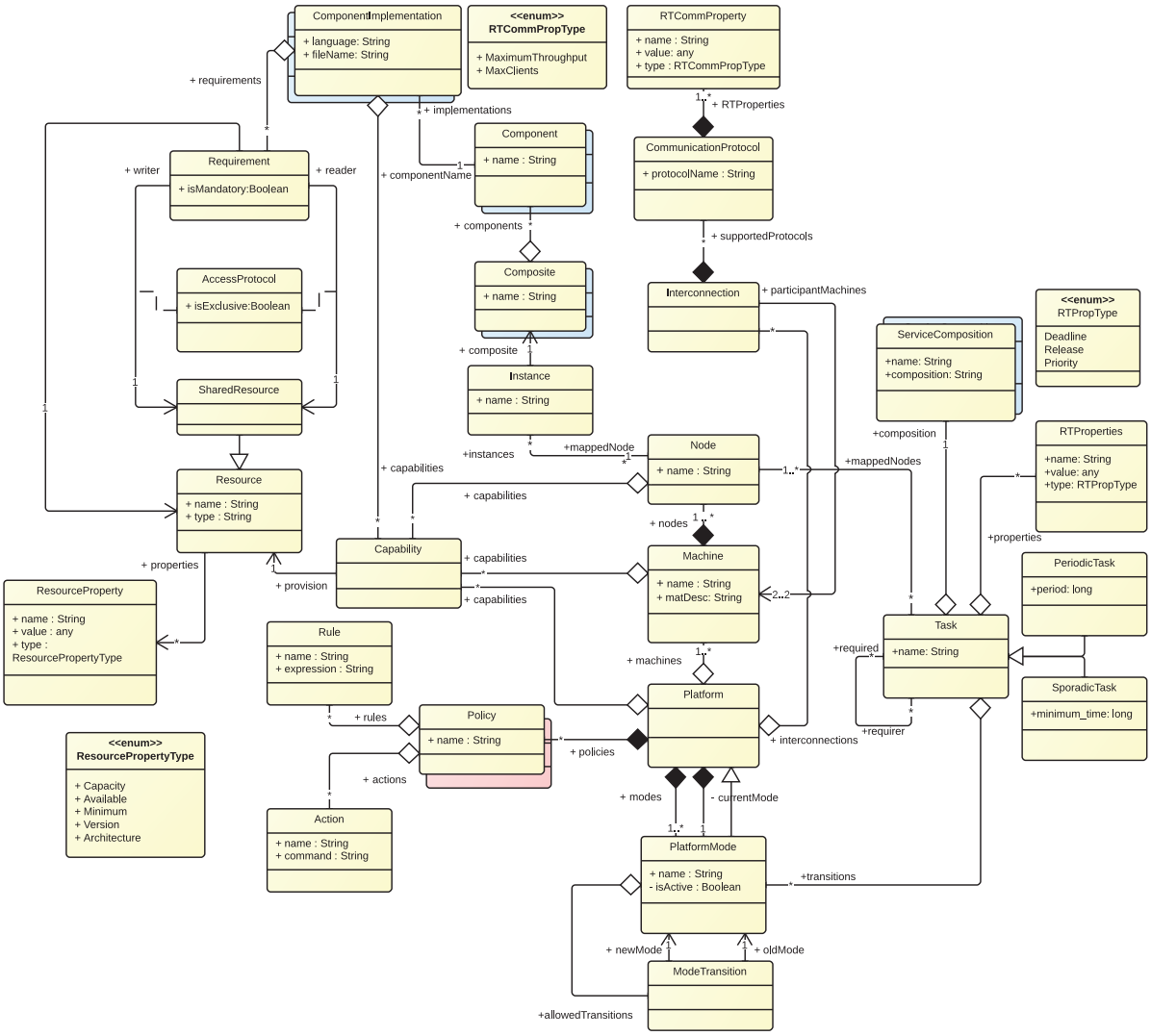

Figure 3.10: URSO Component Metamodel: Deployment Concern

Many important relations were included such as:

- The relation between the cross-cutting concern class Implementation and both Resource Capabilities and Requirements, which allows the framework to reserve Resources for the execution of all Component Implementations related to Composite Instances and the share of applicative libraries or functions between different Components;

- Instances and Tasks relations with Nodes, indicating the deployment mapping of Component artefacts and Schedulable threads onto material Resources and processing units;
- The relation between Tasks and Platform Modes, enabling the automatic interruption of a Task execution upon a Platform Mode change.

Figure 3.11 shows a UML Activity Diagram illustrating the development process in URSO. It completes the activity diagram presented in the Assembly concern to illustrate the Component development process by adding the notions of Service Composition and Task descriptions. Thus, a Deployment unit may contain Composites, Implementation descriptions and artifacts, Task descriptions, Service Compositions or a mixed combination of them all.

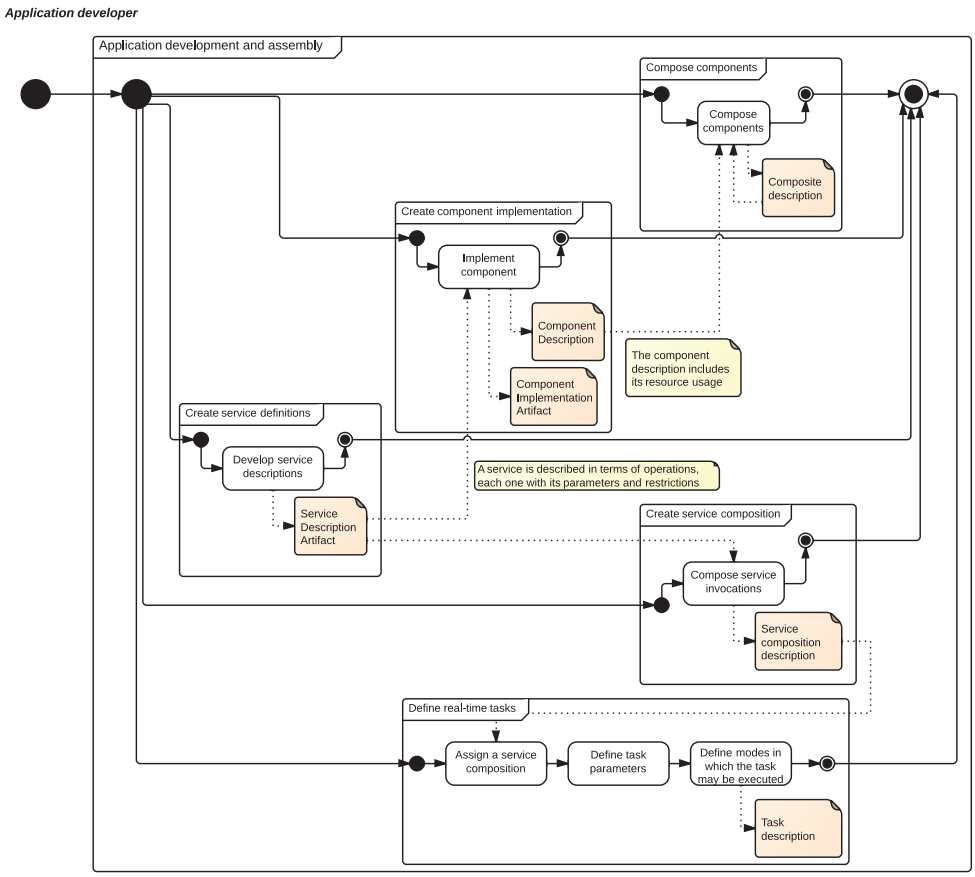

Figure 3.11: Deployment unit development process

Figure 3.12 depicts the process of deploying a Deployment Unit in the framework. Each type of description is treated in a particular way and several verifications are performed before the real installation of Components and execution of Tasks in the framework. Consequently, it is recommended to perform dynamic deployment operations during non-real-time phases, since the deployment of new Services may imply several changes in an application architecture.

If the deployment unit contains Component descriptions, the platform is going to parse

these descriptions and register them, in case other Component Implementations implement them. If Component Instances are part of the description, the platform must retrieve its Resource usage and verify whether there are enough available Resources in the node to which the Instance is being mapped. If there is not, the platform may define policies for replacing the Instance mapping, or to ignore them, or to try and find another Implementation for the same Component in a different technology which requires less Resources, for example. If there are enough Resources, these are then reserved for the Instances and the amount of available Resources in the Platform Node is updated (Resources used by the Implementation are decreased and the Resources provided by the Implementation are added).
When the deployment unit contains Implementation artifacts, the framework parses the Implementation description and the component it implements in order to find its provided Service Descriptions and Service Operations. Then, the platform retrieves the corresponding operations in the Implementation object code, estimates all partial WCETs and stores them. Instances are created afterwards. After creating Instances, they must be resolved. Thus, the platform retrieves the Dependencies of the component they implement, applies the ranking functions on all present Services and Service Properties and bind the best ranked Service provider to each Dependency. When dependencies are satisfied, it is able to calculate the total WCET of Service Operations. Once Instance Services WCET is estimated and Instances are satisfied, the Services provided by the instance are published in the Service Registry (global if it is an exported Services, or local if it is not).
If the deployment unit contains Service Composition descriptions, these are parsed and internally stored. When a Service Composition is referenced by a Task, then the Service Composition is resolved. In fact, it can be seen as the dynamic creation of a Component Implementation: the Services used by the Composition will be selected (they can be filtered as well, just like in Service Dependencies), and the framework will generate the code to glue all the declared Service calls. Since the total WCET of published Services were already been calculated, it is easy to estimate the WCET of the whole Service Composition. This WCET is stored as well.
When Tasks are described in the deployment unit, its parameters are compared to those which were stored for the composition it refers to. For instance, the deadline of the task is compared to the WCET of the Service Composition; if the WCET is bigger than the deadline, that means that the task will never be able to execute the Service Composition with these parameters. The task period must be bigger than the WCET as well. After that, a scheduling analysis is performed in the current set of tasks plus the new task: if all the tasks are schedulable (taking into account the new task parameters, such as its release time), then the Service Composition can really be assigned to the Task thread, as well as its parameters. The framework generates as well parameters to help at run-time: for instance, it keeps the partial WCET expressions ready in case of Service substitution. Once the thread has its parameters set, it can be executed. At the same time, monitors follow the Task execution in order to gather measures for the framework and to inform possible errors to the system administrator.
Figure 3.13 shows the procedure followed in case of a Platform Mode change. Mode changes are triggered by the Platform administrator. Upon a mode change request, the platform wait until the end the execution of the current tasks or the violation of a time-out configured by the administrator. If the time-out occurs and the mode change was not fully performed, then the platform enters an error state.
After the execution of the current tasks, the framework checks among all tasks those that can continue executing, those that must be removed and those that must be added. It removes the removable tasks and then it verifies whether the whole set of tasks remains schedulable by adding the new tasks one by one (it can keep track of the result of the

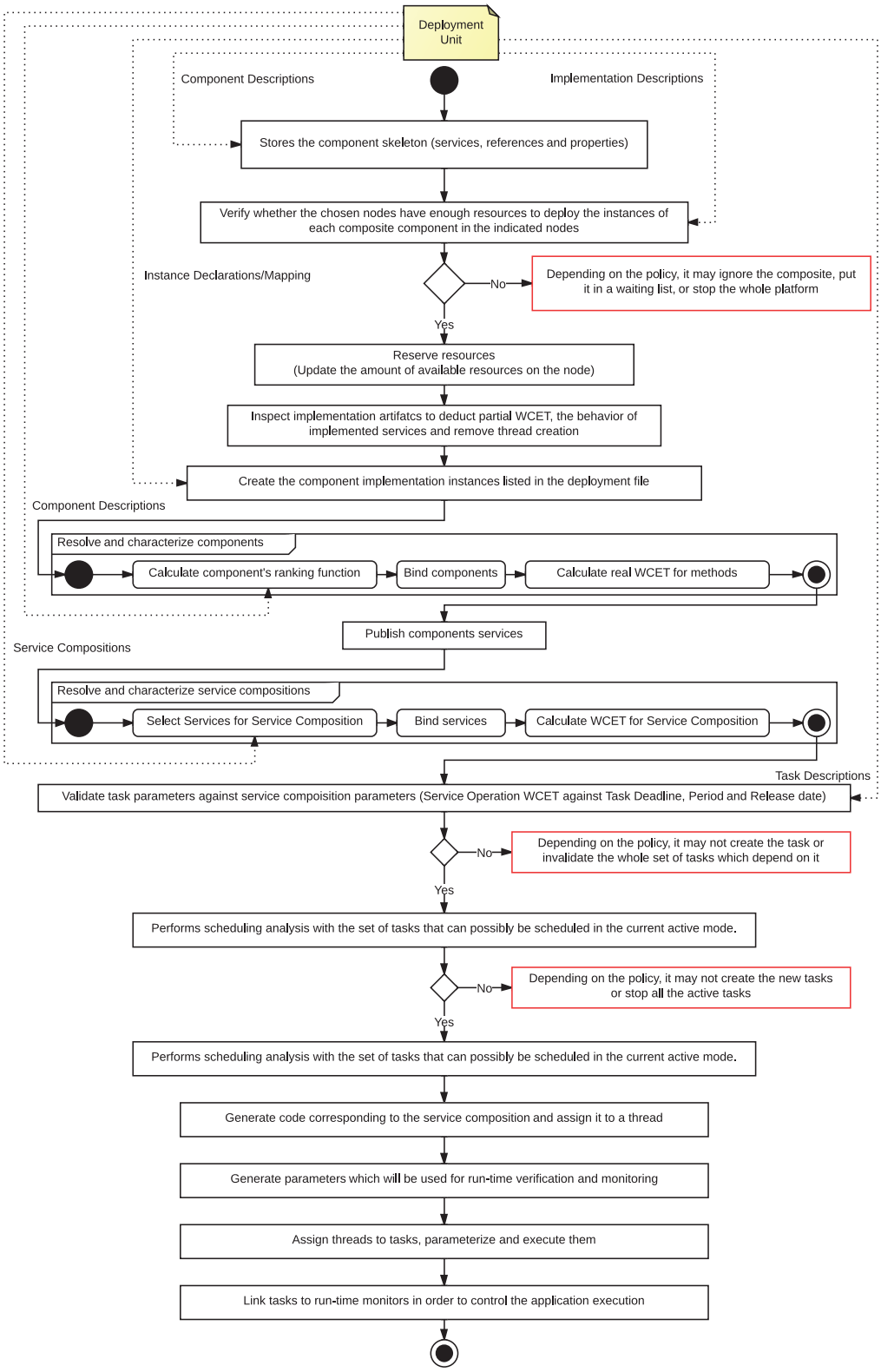

Figure 3.12: Deployment Process Activity

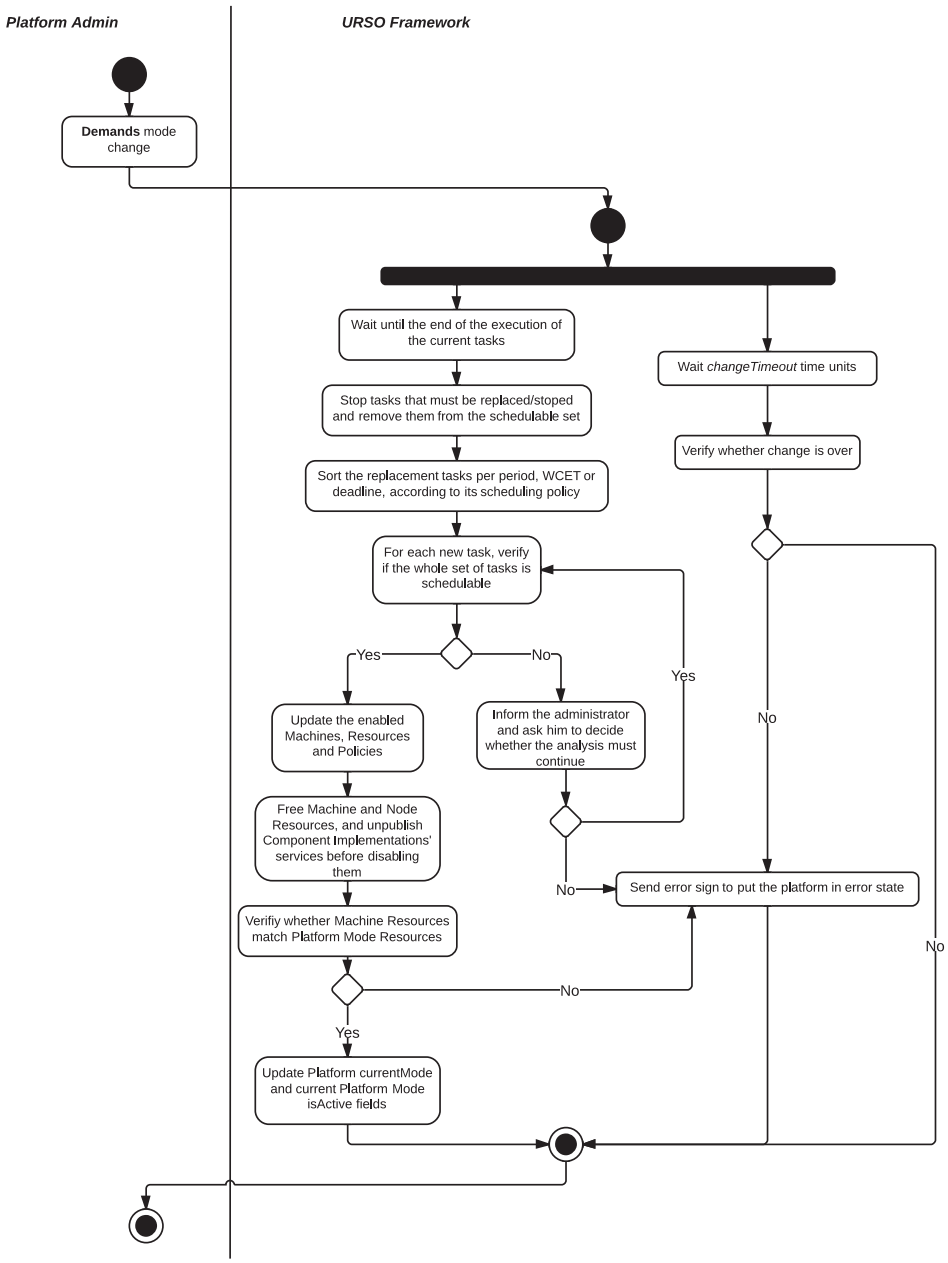

Figure 3.13: Mode Change Activity

schedulability analysis of the last iteration, just like in dynamic programming techniques). When a task can not be added, it asks the administrator if the system can continue executing without it or with a new set of parameters. If not, it puts the system in an error state. If all tasks have been verified and the system is not in an error state, it updates the state of Machines, Nodes and Resources: Those which are not enabled in the new state

are reset (Component Instances are removed from Nodes and Machines and its Services are unpublished, and all Resources are released). The framework then verifies whether the Resources capabilities of active Machines and Nodes match those of the current Platform Mode. Afterwards, it updates the platform to indicate the new Platform Mode as current, finishing the Mode change phase[19].

3.2.5 Other aspects that must be taken into account

Two aspects which are not directly related to business applications must be specifically taken into account in the development and administration of applications in the URSO Component Metamodel: the presence of technical components in the platform and run-time monitoring. Both aspects are not fully detailed, as they are very oriented to the framework implementation, which is not the main focus of this thesis. Still, some attention is given to them in the next sub-sections.

3.2.5.1 Technical Components

Technical components are special components which have access to Platform low-level facilities. They are responsible for automatizing several actions taken by the platform, like scheduling-related actions or Resource management. Services provided by technical components are used uniquely by the platform; however, although the visibility of their services is special, these components are installed in the Platform like any other component, with along with its Resource requirements and Instance mapping. Just like in business applications, several technical components may provide the same service: for instance, two different scheduler components may implement different scheduling policies. It is up to the administrator to choose which services to use according to the needs of the business applications installed in the platform. Other factors may influence the administrator's choice of technical components, like the current active mode and current quantity of available resources.
Technical components are fundamental to the good functioning of the platform, and some of them (like a scheduler) must be present before the installation of any business component. Examples of technical components are:

- *Scheduler*: Schedulers provide services to add or remove tasks to/from the the set of executable tasks, to manage their release, to interrupt their execution in case of preemption, to perform scheduling analysis to the set of tasks and to execute tasks according to its implemented scheduling policy.

- *Resource Manager*: Resource Manager components are responsible for implementing Resource reservation mechanisms. They update Resources availability according to the requirements and capabilities of installed components. Resource Managers may implement dynamic policies where resources can be claimed from a set of components and lend to another set of more important components.

- *Connection Manager*: Connection manager components are responsible for implementing transmission protocols or methods. They are part of the Interconnections between nodes, and the methods they implement is visible as a Binding Protocol in the URSO metamodel.

[19] This Mode change protocol can be generalized to include the platform stop as well; we may consider that the platform stop is a Platform mode with no Tasks and no Machines (thus no Nodes and Resources). Consequently, the only action performed is the reset of Resources, Nodes and Machines before stopping the framework.

- *Shared Resources Access Controller*: As the name suggests, the Shared Resources Access Controller controls the access of Components to Shared Resources. A specific implementation may create copies of a given Resource for every Component that requires it as writer and then it aggregates the values of the copies in the original Resource. Other implementation approaches are allowing only one Component to access the Resource at a time and forbidding modifications.

- *Parsers*: A big amount of information is present at the model at the form of descriptors. All these descriptors must be parsed by the framework, and sometimes sorted under a representation that eases the access of information at runtime. In addition, there is also information which is passed to the platform as expressions (e.g. ranking functions). These expressions must be parsed as well and stored in a way that allows it to easily be evaluated for different input values.

3.2.5.2 Run-time Monitoring

Several aspects of the metamodel must be monitored at run-time. Some of them are local to components and to nodes, whereas some concern the whole platform. In both cases, most technical components cited above are or require run-time monitors (which are technical components themselves, that must as well be characterized in terms of resource usage). Thus, the excessive use of monitors may compromise the quantity of resources available to applications. We will not detail how these run-time aspects should be monitored (monitoring of real-time applications is a big research domain itself indeed), but we do list the most important aspects that should be taken into account about run-time monitoring in the URSO framework.

- *Resource Managers*: Resource Managers should be present at two levels: a node-level Resource Manager, and a global platform Resource Manager. This monitor type has two main functions: first, it must enable resource reservation; second, it can have resource usage enforcement features. The first function does not required much computation at run-time; upon the installation, update and uninstallation of components, the local Resource Manager must verify whether the operation is possible (that is, to verify whether there are enough resources in the node to instantiate the corresponding set of component implementations for installation and update) and then update the amount of available resources in both local (Node) and global (Platform) Resource set. Since these operations are usually performed in non-critical phases, this verification-and-update mechanism should not interfere in applicative tasks' execution time determinism.
  The second function (resource usage enforcement) however must verify whether components are really using the resource quantity they have declared in their descriptor. Since components are passive entities in the URSO metamodel (that is, they do not have a control flow themselves - they are invoked by means of service calls), the best way to verify its resource usage is when it is executed by a task in a Service Composition. However, it is very difficult to verify the resources used by a specific component inside a Service Composition being executed by a thread. An alternative would be to use on-line resource usage analysis techniques, such as Kim's analysis for memory consumption [Kim *et al.* 1999] and on-line schedulability analysis for CPU [Bini & Buttazzo 2004]. Resource enforcement requires background execution, what may interfere in the determinism of real-time tasks.

- *Restriction Managers*: Restrictions over Service Operations Parameters must be performed in background. It usually consists in a sequence of simple comparisons over

the set of inputs and outputs; nevertheless, the Service Operation is not executed unless its set of pre-conditions is completely satisfied. The platform may consider that the evaluation of those assertions corresponds to calling a Service provided by the platform itself, whose WCET is the sum of the WCET of the instructions used to perform all comparisons.
Verifying invariants can be more complicated, because these are assertions that must remain true during the whole Operation execution. In order to implement that, one may create a thread that periodically evaluates the invariant assertion or perform the evaluation only when an instruction uses a given variable (or when it is loaded in the stack, for stack-oriented languages). An optimization could be to verify only in the beginning and in the end of the Operation execution, but using the wrong value inside the Operation could lead to unpredictable consequences. Again, evaluating these values during the method execution may introduce non-determinism in the Service Operation execution time and must be taken into account (in the worst case, the framework may verify the value after the execution of each instruction of the Service Operation, thus its WCET would be the number of instructions multiplied by the WCET of the sum of the evaluation instructions. Post-conditions would raise the same problem of Pre-conditions, but in the end of the execution of a method - it can block the execution of the next activity of the Service Composition, and the time taken to evaluate the parameters (the sum of the WCET of the instructions used to perform all comparisons) must be taken into account in the WCET of the Service Composition.

- *Parsers*: The parsing of descriptors and expressions is mainly performed in non-critical phases, such as installation and service composition or components Dependency resolution. Parsing will trigger some background processing, like the storage of information in a framework-friendly format or object (for expressions or architecture representation) or code generation (for service compositions).

- *Scheduling-related monitors*: Several aspects related to scheduling must be monitored at run-time. Every time a new task is added to the system, a new schedulability analysis must be performed to ensure that the system will have enough resources to execute the current set of tasks without violating tasks timing constraints. This analysis is realized in non-critical phases, when the administrator installs new tasks to the framework. However the system must be able to react in case of deadline violation. The administrator may configure different policies in that case: ignore the deadline violation and let tasks execute, which may incur even more tasks to violate their timing constraints; interrupt the task execution and switch to the execution of the next task in the priority list, which may lead to a functional error, since the interrupted task was not correctly executed until the end and could contain operations to update memory variables used by other tasks; interrupt the whole set of tasks and inform the administrator; change the platform mode and change the whole set of tasks in order to execute in a degraded quality of service state; or many more.

3.2.5.3 Applicative WCET vs. Technical WCET

Estimating the execution time of the instructions of a Service Operation (what we call *Applicative WCET*) is not enough to estimate its real WCET; in fact, several platform features must be used for the application execution to be in good working order, and the execution time of these mechanisms (here called *Technical WCET*) must also be taken into account. First, the communication time must be considered; for that, the framework will

see the Service input and output data types, the communication worst-case throughput (as informed in the Communication Protocol timing descriptor) and estimate the worst case communication time. This time will be added to the WCET of the operation caller, which may be another Service Operation or a Service Composition. If the Operations are restricted by Pre-conditions, Post-conditions and Invariants, the time used to evaluate them must also be added to the Service Operation WCET (as discussed in the section above). Thus, we may consider that the real WCET of a Service Operation S is:

$$\mathrm{WCET}(S) = \mathrm{WCET}(\mathrm{S_{commInput}}) + \mathrm{WCET}(\mathrm{S_{evalPreCond}}) + \mathrm{WCET}(\mathrm{S_{behavior}}) + \mathrm{WCET}(\mathrm{S_{evalInv}}) + \mathrm{WCET}(\mathrm{S_{evalPostCond}}) + \mathrm{WCET}(\mathrm{S_{commOutput}})$$

where $\mathrm{S_{commInput}}$ is the communication time for S inputs, $\mathrm{S_{evalPreCond}}$ is the evaluation time for S Pre-Conditions, $\mathrm{S_{behavior}}$ is the time estimated by analyzing S behavior (generated by the platform), $\mathrm{S_{evalInv}}$ is the evaluation time for S Invariants, $\mathrm{S_{evalPostCond}}$ is the evaluation time for S Post-Conditions and $\mathrm{S_{commOutput}}$ is the communication time for S outputs.

3.3 Example: Dynamic Collision Detection Application

In order to exemplify the concepts introduced in the previous section, we are going to use URSO to model a collision detection application based on the $\mathrm{CD_x}$ benchmark presented by Kalibera *et al.* in [Kalibera *et al.* 2009]. Our collision detection application presents some small differences from that of Kalibera, in order to introduce two aspects relevant to the URSO metamodel that are being more and more frequent to real-time applications: service-oriented computing and dynamism. We call it $\mathrm{DCD_x}$ (for Dynamic Collision Detector).

3.3.1 Overview of the $\mathrm{CD_x}$ Benchmark

The $\mathrm{CD_x}$ benchmark was introduced in 2009 by Tomas Kalibera *et al.* They proposed an open source real-time benchmark suite application for evaluating different Real-time Java virtual machines. The core of the application is a real-time periodic task which reads simulated radar frames and identify potential aircraft collisions. It computes the time between releases of this periodic thread and the time it takes to compute the collisions and uses both values as indicators of the degree of predictability of the underlying virtual machine.
The two main components of the application are the *air traffic simulator* and the *collision detector*.
The *air traffic simulator* generates the the radar frames taken as input for collision detection. Radar frames are lists of aircrafts and their current positions (a point in a three-dimensional Cartesian coordinate system). Aircraft positions are defined by the user as functions whose coordinates are indicated as functions of time. The user can also configure the frame generation rate and the total number of frames.
The collision detector periodically reads frames and, for each aircraft, it estimates an approximated trajectory based on its position in the current and in the precedent frames. Speed is assumed to be constant. The detection is split in two steps: first the whole set of aircrafts is reduced into smaller sets of aircrafts; then, for every small set, the detector verifies the collision potential for every two aircrafts. The detector identifies as a collision potential every pair of aircrafts whose distance is equal or less than a user-defined proximity radius.
The frame transfer from the simulator to the detector is performed by means of a pre-allocated frame buffer. Frames can be simulated before the execution of the detector, concurrently (adding background noise) or synchronously (the detector waits for the generation of frames and the simulator waits for the frames to be processed by the detector). It

is also possible to store the frames in binary format, to increase performance and simplify the benchmark implementation and analysis. Since the list of aircrafts is informed by the user in a configuration file, it is not possible to change the set of aircrafts at runtime.

3.3.2 DCD_x: A Dynamic and Service-Oriented CD_x benchmark

3.3.2.1 A Service-Oriented version of CD_x

Since every concern is well-defined, decoupling the CD_x application architecture and to produce a service-oriented version of it is very straightforward. Figure 3.14 depicts a possible architecture representation of a service-oriented version of CD_x. The Detection Manager is periodically invoked by a thread to perform the collision detection. Before executing the detection algorithm, it retrieves the radar frames by means of the Frame Service. Then, it uses the Motionizer Service to calculate aircrafts trajectory. The motions are taken as input by the Collision Detector's Detection Service, which returns a list of potential collisions. The Frame Pool component contains a buffer which is filled by the Simulator by means of the Frame Service as well. The Motionizer uses the Aircraft Position Service to query for ancient past aircraft positions and to update them. The State Storage contains a table which stores the Aircraft ID and its precedent positions. Decoupling CD_x would increase its flexibility and allow different implementations of the collision detection algorithm. It is possible as well to measure the duration of every service call in order to measure its impact in the application. The Collision Detection Service is invoked by a periodic real-time thread; depending on the user configuration, the Simulation Service can be invoked before the creation of the Detection thread to populate the Frame Pool buffer, at the same time, in a concurrent thread, or synchronously (in the same thread or synchronized by means of events).

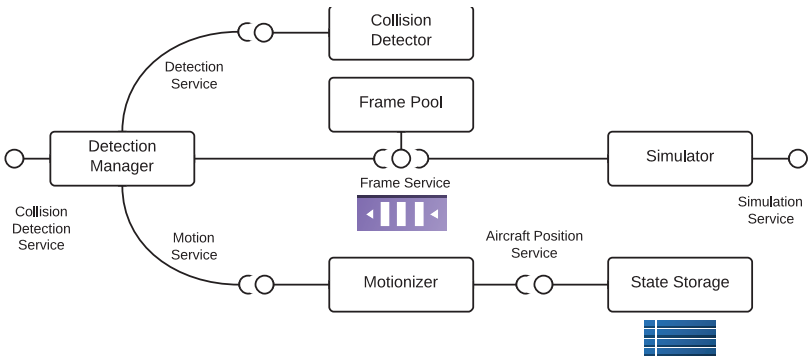

Figure 3.14: A Service-oriented Architecture for the Collision Detector Application

3.3.2.2 Adding Dynamism to CD_x

In its original version, CD_x does not allow the set of aircrafts to change at runtime. We have modified its architecture in order to allow the system to dynamically admit (or not) new aircrafts.

The aircrafts, which before were listed in the configuration file, can now be added to the system at run-time. Aircrafts provide an Aircraft Service, which is consumed by the Simulator. By means of this Aircraft Service, it is possible to retrieve the aircraft's name and trajectory function over time. Another parameter is added to Aircrafts: they now have a maximum

flight time t_{max}, after which they are disabled. After t_{max}, an aircraft unregister its Aircraft Service from the Service Registry. This allow the current set of Aircrafts to be always changing. It is possible to as well to establish boundaries for the x-, y- and z-coordinates of each aircraft, so that whenever an aircraft leaves the delimited region of interest, its position is discarded for the collision detection phase. For the benchmark, both the aircraft arrival distribution (the time when they are installed in the system) and aircraft's t_{max} parameter can be randomly generated in order to have a more realistic workload. Figure 3.15 shows the modification in the benchmark application architecture.

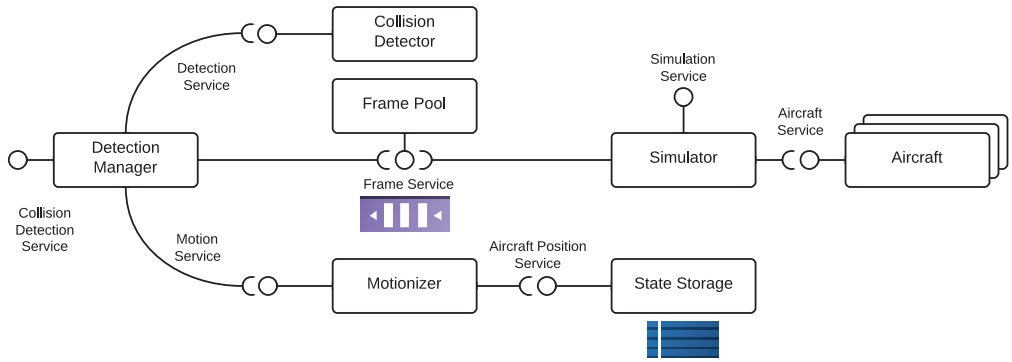

Figure 3.15: Dynamic Collision Detector Application Architecture

3.3.3 URSO Description of DCD_x

3.3.3.1 Application Design Overview

Services are *passive* entities in URSO; that means that they do not have a control flow themselves, they can only be invoked by another entity. Services can be called by other Services or by Service Compositions, which are themselves executed by Tasks. Two Tasks can be identified in DCD_x: the radar simulation and the collision detection. The Service Composition of the former requires gathering all the Service Implementations of the Aircraft Service, calculating its current position, generating a radar frame and storing it in the Frame Pool buffer. The Service Composition of the latter will first invoke the Frame Service in order to get an unprocessed frame, then invoke the Motion Service to get the Aircrafts trajectories and finally invoke the Detection Service on the trajectories to get a list of potential collisions. The Motion Service uses the Aircraft Position Service in order to get and set aircrafts' position.

3.3.3.2 Platform Description: the Deployment concern

For this example we are going to consider as hardware infrastructure the same machine used in [Kalibera *et al.* 2009]: a machine with two Pentium processors (or cores), executing a Virtual Machine Oracle Java RTS 2.1 (with a 300 MB heap memory) on top of a Real-time Linux 2.6 Kernel.
We will start by identifying the available resources in the platform: they are the processor, the virtual machine (along with the libraries it provides), the available heap memory and the Linux kernel. We list below the URSO description of each one of these Resources.

Resource descriptions:

processor1Res =
$\langle$"$Processor1$" , "CPU" , $\{\langle$"$Brand$" , "$Brand$" , "$IntelPentium4$"$\rangle$;
$\langle$"$Frequency$" , "$FrequencyInGHz$" , 2.1$\rangle\}\rangle$
processor2Res =
$\langle$"$Processor2$" , "CPU" , $\{\langle$"$Brand$" , "$Brand$" , "$IntelPentium4$"$\rangle$;
$\langle$"$Frequency$" , "$FrequencyInGHz$" , 2.1$\rangle\}\rangle$
VMRes = $\langle$"$JavaRTS$" , "VM" , $\{\langle$"$Version$" , "$Version$" , 2.1$\rangle$;
$\langle$"$Vendor$" , "$Vendor$" , "$Oracle$"$\rangle\langle$"$Type$" , "$JVMType$" , "$RealTime$"$\rangle\}\rangle$
memoryRes =
$\langle$"$HeapMemory$" , "$Memory$" , $\{\langle$"$Available$" , "$MemoryInMB$" , 300$\rangle$;
$\langle$"$Quantity$" , "$MemoryInMB$" , 300$\rangle\}\rangle$
OSRes = $\langle$"$RTLinux$" , "OS" , $\{\langle$"$Version$" , "$Version$" , 2.6$\rangle$;
$\langle$"$Name$" , "$OSName$" , "$Ubuntu$"$\rangle\}\rangle$

Except for the memory, all the other Resources are shared. The processor Resource is managed by the platform Schedulers, but there may be no control for the access of other Shared Resources.
The benchmark in its original form is executed on top of a Java Virtual Machine and does not use any network connection; all communication is intra-JVM by means of method invocation. In this example, we are going to keep this architecture, but more complex communication protocols could have been used and described by URSO. For the sake of simplicity and due to the fact that our platform is simple (mono-core, with few Resources), we are going to define only one node, which is going to correspond to the machine in which we are executing the benchmark.

Node and Machine descriptions:
node1 =
$\langle$"$Node1$" , $\{processor1Res$; $processor2Res$; $VMRes$; $memoryRes$; $OSRes\}$,
$\langle\{$"$Node1$" ; "$Node1$"$\}$, $\{intraJVM\}\rangle\rangle$
machine1 =
$\langle$"$Machine1$" , $\{node1\}$, $\{processor1Res$; $processor2Res$; $VMRes$; $memoryRes$;
OSRes$\}\langle\{$"$Machine1$" ; "$Machine1$"$\}$, $\{local\}\rangle\rangle$

We consider that the delay of a method invocation intra-JVM and in the local network is negligible.

Communication protocols:
intraJVM = $\langle$"$IntraJVM$" , $\{\langle$"$Throughput$" , "$CommThroughput$" , $\infty\rangle\}\rangle$
local = $\langle$"$LocalNetwork$" , $\{\langle$"$Throughput$" , "$CommThroughput$" , $\infty\rangle\}\rangle$

Our platform will have only one mode, in which all resources and machines are enabled[20].

Platform and Platform Modes:
modeStd =
$\langle$"$StdMode$" , **true** , $\{machine1\}$, $\emptyset$, $\{processor1Res$; $processor2Res$; $VMRes$; $memoryRes$; $OSRes\}\rangle$
platform =
$\langle\{machine1\}$, $\{modeStd\}$, $\emptyset$, $\{processorRes$; $processor2Res$; $VMRes$; $memoryRes$; $OSRes\}$, $modeStd\rangle$

[20]We are assuming an empty set of policies, as the benchmark different possible configurations do not interfere in the platform behavior itself.

3.3.3.3 Services and Components: the Assembly concern

Since the platform infrastructure has been described, we are going now to start describing the application architecture itself. First, we are going to define the Services used in the application. As depicted in Figure 3.15, seven services are present in the application architecture. However only five of them will be useful for us, since the Detection Manager and the Simulator component (and their provided services) fulfill the role of Service mediator in the original architecture and will be transformed in Service Compositions (and assigned to Tasks) in our URSO description. As said in the previous section, Services and Dependencies are bound by means of intra JVM mechanisms.

Bindings:
intraJVMBinding[21] $= \langle \emptyset \ , \ \emptyset \ , \ \mathbf{true} \ , \ intraJVM \rangle$

Service descriptions:
detSvc $= \langle \text{“}DetectionService\text{”} \ , \ detDesc \ , \ \emptyset \ , \ \{intraJVMBinding\} \ , \ \emptyset \rangle$
motSvc $= \langle \text{“}MotionService\text{”} \ , \ motDesc \ , \ \emptyset \ , \ \{intraJVMBinding\} \ , \ \emptyset \rangle$
airSvc $= \langle \text{“}AircraftService\text{”} \ , \ airDesc \ , \ \emptyset \ , \ \{intraJVMBinding\} \ , \ \emptyset \rangle$
airPosSvc[22] $=$
$\langle \text{“}AircraftPositionService\text{”} \ , \ airPosDesc \ , \ \emptyset \ , \ \{intraJVMBinding\} \ , \ \emptyset \rangle$
frmSvc $= \langle \text{“}FrameService\text{”} \ , \ frmDesc \ , \ \emptyset \ , \ \{intraJVMBinding\} \ , \ \emptyset \rangle$

The Service Descriptions[23] below are shared between both Services providing them and Dependencies requiring them.

Service Descriptions:
detDesc $= \langle \text{“}Detection\text{”} \ , \ \{detectColisionOp\} \rangle$
motDesc $= \langle \text{“}Motion\text{”} \ , \ \{createMotionOp\} \rangle$
airDesc $= \langle \text{“}Aircraft\text{”} \ , \ \{getAircraftOp\} \rangle$
airPosDesc $= \langle \text{“}AircraftPosition\text{”} \ , \ \{getPosOp \ ; \ updatePosOp\} \rangle$
frmDesc $= \langle \text{“}Frame\text{”} \ , \ \{getFrameOp \ ; \ storeFrameOp\} \rangle$

Service Operations:
detectColisionOp $= \langle DetectColision \ ,$
$\{(Input)\langle \text{“}Motions\text{”} \ , \ List < Motion > \rangle \ ; \ (Output)\langle \text{“}Collisions\text{”} \ , \ List < Collision > \rangle\} \rangle$
createMotionOp $= \langle CreateMotion \ ,$
$\{(Input)\langle \text{“}RadarFrame\text{”} \ , \ Map < String, 3DPosition > \rangle \ ; \ (Output)\langle \text{“}Motions\text{”} \ , \ List < Motion > \rangle\} \rangle$
getAircraftOp $= \langle GetAircraft \ ,$
$\{(Output)\langle \text{“}ID\text{”} \ , \ String \rangle \ ; \ (Output)\langle \text{“}Trajectory\text{”} \ , \ String \rangle \ ; \ (Output)\langle \text{“}FlightDuration\text{”} \ , \ int \rangle\} \rangle$
getPosOp $= \langle GetPosition \ ,$
$\{(Input)\langle \text{“}ID\text{”} \ , \ String \rangle \ ; \ (Output)\langle \text{“}Position\text{”} \ , \ 3DPosition \rangle\} \rangle$
updatePosOp $= \langle UpdatePosition \ ,$
$\{(Input)\langle \text{“}ID\text{”} \ , \ String \rangle \ ; \ (Input)\langle \text{“}Position\text{”} \ , \ 3DPosition \rangle\} \rangle$
getFrameOp $= \langle GetFrame \ ,$
$\{(Output)\langle \text{“}RadarFrame\text{”} \ , \ Map < String, 3DPosition > \rangle\} \rangle$
storeFrameOp $= \langle StoreFrame \ ,$
$\{(Input)\langle \text{“}RadarFrame\text{”} \ , \ Map < String, 3DPosition > \rangle\} \rangle$

[21] The set of services and dependencies of each Binding instance is automatically filled by the platform

[22] The system could also have been modeled in a way that the motion component itself stores the position. In that case, this Service would not be necessary.

[23] The operations listed below do not contain the two last fields of the tuple, instructions and restrictions, as the former is automatically filled by the framework in Component Implementation instances and the latter is empty in our example

Once the services have been defined, we may group them onto components and composites. We have decided to group the Motionizer, State Storage and Collision Detector components onto a Collision Detection composite; other composites present in the system wrap only one component (Aircrafts composites and Frame Pool Composite)[24].

Component Implementations:
colDecImpl = $\langle$"*Java*" , "*CollisionDetector.class*" , "*collisionDetectorComp*"$\rangle$
motionImpl = $\langle$"*Java*" , "*Motionizer.class*" , "*motionizerComp*"$\rangle$
stateStrImpl = $\langle$"*Java*" , "*StateStorage.class*" , "*stateStorageComp*"$\rangle$
framePoolImpl = $\langle$"*Java*" , "*FramePool.class*" , "*framePoolComp*"$\rangle$
aircraftImplB747 = $\langle$"*Java*" , "*AircraftB747.class*" , "*aircraft1Comp*"$\rangle$
aircraftImplA320 = $\langle$"*Java*" , "*AircraftA320.class*" , "*aircraft2Comp*"$\rangle$

Dependencies used by Components:
storageDep =
$\langle$"*StorageDependency*" , *airPosDesc* , "1..1" , $\emptyset$, $\{intraJVMBinding\}\rangle$

Component descriptions:
stateStorage = $\langle$"*stateStorageComp*" , $\{airPosSvc\}$, $\emptyset$, $\{stateStrImpl\}$, $\emptyset\rangle$
motionizer = $\langle$"*motionizerComp*" , $\{motSvc\}$, $\{storageDep\}$, $\{motionImpl\}$, $\emptyset\rangle$
collision = $\langle$"*colisionDetectorComp*" , $\{detSvc\}$, $\emptyset$, $\{colDecImpl\}$, $\emptyset\rangle$
framePool = $\langle$"*framePoolComp*" , $\{frmSvc\}$, $\emptyset$, $\{framePoolImpl\}$, $\emptyset\rangle$
aircraft1 = $\langle$"*aircraft1Comp*" , $\{airSvc\}$, $\emptyset$, $\{aircraftImplB747\}$, $\emptyset\rangle$
aircraft2[25] = $\langle$"*aircraft2Comp*" , $\{airSvc\}$, $\emptyset$, $\{aircraftImplA320\}$, $\emptyset\rangle$

Composite descriptions:
detCpte = $\langle$"*DetectionComposite*" , $\{collision$; $motionizer$; $stateStorage\}$, $\emptyset$, $\emptyset$,
$\{airPosSvc \in stateStorage.services$; $motSvc \in motionizer.services\}\rangle$
frmCpte = $\langle$"*FramePoolComposite* , $\{framePool\}$, $\emptyset$, $\emptyset$,
$\{frmSvc \in framePool.services\}\rangle$
airCft1Cpte = $\langle$"*Aircraft1Composite*" , *aircraft1* , $\emptyset$, $\emptyset$,
$\{airSvc \in aircraft1.services\}\rangle$
airCft2Cpte[26] = $\langle$"*Aircraft2Composite*" , *aircraft2* , $\emptyset$, $\emptyset$,
$\{airSvc \in aircraft2.services\}\rangle$

These four components would be ready to be packaged along with its implementation artefacts (object codes and descriptors) and dynamically installed in the Platform. As mentioned before, components and composites are passive entities, which are going to provide services that are going to be used by the service compositions or by other composites/components when they are executed (again, by means of service compositions).
Now we are going to define two service compositions and tasks to our system. The first is the composition responsible for simulating the aircraft movement and storing radar frames in frame pool Component. Thus our service composition has associations with the Aircraft

[24] We are considering that all implementations here were realized in Java, just as in the CD_j distribution.
[25] One component with two different implementations would mean that the same component offers different implementations, in different technologies. He we are using two different components to show two aircrafts in the system which are not necessarily alike.
[26] A group of aircrafts could be encapsulated into one composite. However here we assume that both aircrafts were developed by different teams and are going to be dynamically and independently deployed in the platform. It is also possible to create several instances, with different names, of the same composite.

service provided by the aircrafts and the FramePool Service[27,28,29].

Simulation Service Composition and Dependencies
aircDep = ⟨"*AircraftDependency*" , *airDesc* , "0.." , ∅ , {*intraJVMBinding*}⟩
frmPoolDep =
⟨"*FramePoolDependency*" , *frmDesc* , "1..1" , ∅ , {*intraJVMBinding*}⟩
simComption = ⟨"*SimulationComposition*" , *simulate* , {*aircDep* ; *frmPoolDep*}⟩

simulate[30] =
"*counter* = 0;
do
  invoke system.getCollectionItem, (aircDep, *counter*), *aircraft*;
  invoke getAircraftOp ∈ aircraft.description.operations, *null*, *aircraftInfo*;
  invoke system.timenow, *null*, *t*;
  if *aircraftInfo*.t_{max} < *t* **then**
    invoke system.evaluate (*aircraftInfo*.Trajectory, *t*), *position3d*;
    invoke system.addToMap, ((*aircraftInfo*.ID, *position3d*), *frame*), *null*
  counter = counter + 1;
count(aircDep) **times**;
invoke storeFrameOp ∈ frmPoolDep.description.operations, *frame*, *null* "

The second service composition gets frames from the Frame Pool, transform these frames into a set of vectors (motions) and then checks for collisions based on these motions. Several algorithms could be used to detect aircraft collisions; thus, several Implementations could be installed in the Platform and the application could switch between them in case of a mode change (*e.g.* switch to an Implementation which uses less Resources if the Platform enters in an embedded mode).

Collision Detection Service Composition and Dependencies
motionDep = ⟨"*MotionDependency*" , *motDesc* , "1..1" , ∅ , {*intraJVMBinding*}⟩
frmPoolDep =
⟨"*FramePoolDependency*" , *frmDesc* , "1..1" , ∅ , {*intraJVMBinding*}⟩
detectDep = ⟨"*DetectorDependency*" , *detDesc* , "1..1" , ∅ , {*intraJVMBinding*}⟩
detComption =
⟨"*DetectionComposition*" , *detect* , {*motionDep* ; *frmPoolDep* ; *detectDep*}⟩

detect =
"**invoke** getFrameOp ∈ frmPoolDep.description.operations, *null*, *frame*;
invoke createMotionsOp ∈ motionDep.description.operations, *frame*, *motions*
invoke detectCollisionOp ∈ detectDep.description.operations, *motions*, *collisions*
invoke system.print, *collisions*, *null*"

Since the Behavior concern is related to the Components Implementation and its respective Object code, and we are not defining any restrictions in this example, it is not up to

[27]We have defined some system/platform services for this compositions. The operation *timenow* returns the current system time. The operation *evaluate* parses a function represented by a string, calculates it for a given input value and returns the evaluated output value. The operation *addToMap* adds an item to a map. The operation *count* returns the cardinality of a collection. The operation *print* prints the value of a variable to the standard output.

[28]The word *null* is being used in the composition to represent the lack of input or output variables in certain Service Operations.

[29]The Dependency towards the Services provided by the system is implicit. The system provides basic services which are usually present in programming languages libraries (*e.g.* in our example, it offers access to functions to manipulate collections and to access the system clock.).

the Developer/Application Designer to describe concepts related to the Behavior concern. Thus, we are going to return to the Deployment concern and map the assembly elements back to the Platform and Infrastructure Resources.

3.3.3.4 Mapping onto the Deployment Concern

First we must assign tasks to the Service Compositions defined in the Assembly concern. The detection composition will be assigned to a periodic real-time task, whereas the simulation task will be assigned to a non-real-time[31] sporadic task[32,33]

Tasks
detectionTask =
(Periodic)$\langle$"$DetectionTask$" , $detComption$, $node1$, $detRTProp$, $\emptyset$, $\emptyset$, $10\rangle$
simulationTask =
(Sporadic)$\langle$"$SimulationTask$" , $simComption$, $node1$, $\emptyset$, $\emptyset$, $\emptyset$, $10\rangle$

Task Properties
detRTProp = $\{\langle$"$Deadline$" , "$Deadline$" , $10\rangle\}$

Since the platform contains only one operating mode, it is not necessary to define mode changes. It is up to the node to distribute the tasks among its processor units. Policies can be defined to distribute it equally, or to prioritize the processor with higher frequency.
The next step is to characterize our Implementations in terms of Resource usage. We suppose here that the memory consumption values were obtained by means of an off-line resource usage analysis technique based on the allocation rate found in the Java class-file. Thus, we update the Component Implementation definitions presented beforehand in order to include Resource Requirements and Capabilities. Two Resources are present in the Implementations' Requirements: the dependency towards the Real-time Java Virtual Machine (present in the collision detection component which will be executed in a real-time thread and in components holding complex data structure to represent system state, since they may need to use real-time libraries to avoid non-determinism introduction in both thread execution and data manipulation) and their RAM memory requirements.
The match between the Resource Requirement and the Resource in the Platform is done by means of the Resource type and the properties defined over the Requirements (it can be seen as a selection mechanism, just as in its Service counterpart). We consider that Java Virtual Machine is a Shared resource which allows concurrent access and only allows Reader-type access. Consequently, all JVM Requirements are marked as "*Shared - Reader*".

Component Implementations updated:
colDecImpl =
$\langle$"$Java$" , "$CollisionDetector.class$" , "$collisionDetectorComp$" , $\emptyset$, $\langle$**true** , $reqMem10K\rangle\rangle$
motionImpl =
$\langle$"$Java$" , "$Motionizer.class$" , "$motionizerComp$" , $\emptyset$, $\langle$**true** , $reqMem1K\rangle\rangle$
stateStrImpl =
$\langle$"$Java$" , "$StateStorage.class$" , "$stateStorageComp$" , $\emptyset$, $\langle$**true** , $reqMem1M\rangle\rangle$

[31] In our example, the non-real-time factor will be expressed by the fact that the task's set of real-time properties is empty.
[32] The developer must be sure that the detection task, which is real-time and has timing constraints, will have a priority higher than that of the simulation task. That is particularly important if priorities are automatically assigned by the platform following a real-time scheduling policy (like Rate-Monotonic Scheduling or Earliest Deadline First Scheduling).
[33] Real-time properties values are here expressed in milliseconds.

framePoolImpl =
$\langle$"$Java$" , "$FramePool.class$" , "$framePoolComp$" , $\emptyset$, $\langle$**true** , $reqMem1M\rangle\rangle$
aircraftImplB747 =
$\langle$"$Java$" , "$Aircraft-B747.class$" , "$aircraft1Comp$" , $\emptyset$, $\langle$**true** , $reqMem1K\rangle\rangle$
aircraftImplA320 =
$\langle$"$Java$" , "$Aircraft-A320.class$" , "$aircraft2Comp$" , $\emptyset$, $\langle$**true** , $reqMem1K\rangle\rangle$

Resource Requirements:
reqMem1K =
$\{\langle$"$Memory$" , "$Memory$" , $\{\langle$"$Available$" , "$MemoryInMB$" , "0.001"$\rangle\}\rangle\}$
reqMem10K =
$\{(Shared-Reader)\langle$"$JavaRTS$" , "VM" , $\langle$"$Type$" , "$JVMType$" , "$RealTime$"$\rangle\rangle$;
$\langle$"$Memory$" , "$Memory$" , $\{\langle$"$Available$" , "$MemoryInMB$" , "0.01"$\rangle\}\rangle\}$
reqMem1M =
$\{(Shared-Reader)\langle$"$JavaRTS$" , "VM" , $\langle$"$Type$" , "$JVMType$" , "$RealTime$"$\rangle\rangle$;
$\langle$"$Memory$" , "$Memory$" , $\{\langle$"$Available$" , "$MemoryInMB$" , "1"$\rangle\}\rangle\}$

These updated versions of the Implementation description must be installed in the Platform along with the Implementations artifact themselves. Upon the definition of Instances, the Platform will find the Implementations for the referenced Components/Composites and perform all verifications concerning Resource Usage. We describe below the Instances corresponding to our Composites.

Composite Instances:
detInstance = $\langle$"$DetectorInstance$" , $detCpte$, $node1\rangle$
frmInstance = $\langle$"$FramePoolInstance$" , $frmCpte$, $node1\rangle$
air1Instance = $\langle$"$Aircraft1Instance$" , $aircft1Cpte$, $node1\rangle$
air2Instance = $\langle$"$Aircraft2Instance$" , $aircft2Cpte$, $node1\rangle$

After parsing the Instance definition, the Platform will instantiate the Composites (and the Components that form them), resolve them in terms of Dependencies and then publish all Services. After the Service Publication, the Service Composition will be resolved, and then the tasks of our application can be started.

3.4 Summary and Discussion

3.4.1 URSO Overview

In this chapter, we have presented the URSO - Unified Real-time Service-Oriented - Component metamodel. URSO is composed of three concerns: Deployment, Assembly and Behaviour.

3.4.1.1 URSO Metamodel

The **Deployment** concern revolves around the platform description and the mapping of software elements on platform elements. A *Platform* is composed of *Machines*, which in turn may be divided into smaller elements called *Nodes*. Platforms, Machines and Nodes contain *Resources*; the Resources of a Node must be listed among the Resources of its containing Machines and must not be shared among other Nodes (unless it is a *Shared Resource*). The same relation applies for Machine Resources and their containing Platform. The Resources offered by Platforms, Machines and Nodes are called *Capabilities*. Resource

Capabilities can be quantified, so a specific amount of that Resource can be reserved by hosted applications.
Nodes and Machines can be connected two-by-two by means of *Interconnections*. Interconnections may support different *Communication Protocols*. The real-time properties of Communication Protocols must be characterized, so that the communication delay is taken into account in the performance estimation analysis.
Platforms may declare Platform modes. In a given mode, only a subset of Resources, Machines and Nodes may be activated. The Resources of the active Machines and Nodes must be included in the Platform Mode active Resources. In addition, Platforms and Platform Modes may have Policies, that is, actions which are performed when a given event (specified by means of logical rules on Platform variables) is triggered.
The **Assembly** concern addresses the assembly and composition of software components. *Components* can be vertically assembled into *Composites*. Components and Composites provide *Services*. These Services may be qualified with *Service Properties* and required by other Components by means of *Dependencies*. Internal Composite Services and Dependencies may be promoted to *Exported Services* and *Imported Dependencies* (respectively) so that they are seen by other Composites. That works by means of local and global *Service Registries*. Services and Dependencies are described by means of *Service Descriptions*, which contain the operations (*Service Operations*) implemented/required by the Component. *Service Compositions* may be seen as a special type of Composite; they are going to structure calls towards Services to implement a more complex business function. Service Compositions only declare Dependencies towards Services present in the global registry (*i.e.* Imported Dependencies).
In the **Behaviour** concern, Service Operations can be associated with Restrictions such as *Invariants*, *Pre-* and *Post-Conditions* over its *Parameters* (*Inputs* and *Outputs*) and variables. The implementation of Service Operations is constituted by a technology-dependent sequence of *Instructions*. For each Operation, the framework fills the sequence of Instructions and, based on an internal table with the WCET estimation for each Instruction and the WCET of the Services the Component is using, it estimates the WCET of the Operation.
Then we have extended the Deployment concern to include concepts related to the mapping of Assembly elements onto Platform elements. Service Compositions are mapped onto *Tasks*, which may or not have real-time properties. Tasks may be associated to Platform modes by means of *Mode transitions*; that means that when a Platform switches its current Mode, a task may be activated or replaced. Component Instances are associated to Nodes; the *Implementation* of the Component they correspond may provide or require Node Resources through Resource Capabilities and *Requirements*.

3.4.1.2 Methodology

A **platform administrator** is responsible for describing all the aspects of the platform. He must:

- Describe the Machines that compose the Platform;
- Describe the logical partitions of the platform machines (Nodes);
- Describe the Resources of the Platform and affect these Resources to Machines and Nodes accordingly;
- Describe the Interconnections between the Machines and characterize the communication supported by each Interconnection;

- Define Platform modes and the subset of Resources, Machines and Nodes for each mode;

- And define the policies supported by the Platform in each mode.

The platform description will configure the framework, so it must be provided to the platform before the installation and execution of any component.
After the platform configuration, the **framework** must estimate the WCET of low-level technology-independent instructions that will be used to implement the Service Operations. This can be done by providing a calib ration/sampling application that will be executed several time so that a good estimation of the average execution time and the jitter can be done. The WCET of the instructions is kept internally and used by the platform to estimate the WCET of its hosted applications.
Once the platform performance is sampled, it is possible to deploy Technical components (such as Schedulers Monitors and Resource Managers) and components created by **application developers**. Default technical components can be specified. The use of specific Technical components, as well as the trigger of platform mode changes, is controlled by the Platform administrator.
Five software artefacts can be defined by the application Developer:

- Service Descriptions, which are constituted by the list of Operations, their Parameters, Restrictions and associated Policies;

- Components/Composite descriptions, which group Service Descriptions and other Components for a common goal, by means of Services, Dependencies and promotions of both of them;

- Component Implementations, which link a Component description with an object code artefact and the Resources required by it, and whose instances are going to be mapped onto Platform Nodes;

- Service Compositions, which structure Service calls to perform a business function;

- and Tasks, which map Service Compositions to schedulable units that will be executed in a given Platform Node.

Service Descriptions are necessary to the creation of Components and Service Compositions; Component and Platform (more specifically Resource) descriptors are necessary to the description of Component Implementations; and Service compositions are needed to create real-time and non real-time Tasks.
These five descriptors can be dynamically delivered to the platform (if the current mode of the platform permits so), which will store them. Upon the deployment of a Component Implementation the platform will extract the list of Instructions for each Service Operation and, based on its control flow, estimate its WCET. The WCET of a Component Implementation will be incorporated to the WCET of the Services of all Components that use it.
When a Task is deployed, an analysis is performed as well, considering the control flow of the composition code, the real-time constraints in the task descriptor and the WCET of the Services it depends on. If the adding the task makes the system fail on the schedulability analysis, or if the task can not execute within its specified deadline due to the services it is bound to, actions can be executed by the platform depending on the active policies.

3.4.2 Discussion

For the moment, URSO is a very descriptive model. Most of the information depends on the data described by developers and administrator on the required set of descriptors. That may be very error-prone if a tool is not provided to aid the developers to design their applications and to the administrator to model the target execution platform. A graphic tool, generating the URSO descriptors and representation of the system could be a very good solution to this problem. It does not guarantee however that the information given by the developer is true. Indeed, for the moment resources are not verified against their existence: the framework keeps a counter for each quantifiable resource but does not enforce its usage quota. Some of the resources though, such as processors/cores and memory can easily be reserved and verified.

The communication description can also be improved. First, the timing properties could be estimated by means of sampling for some protocols, instead of depending on the timing properties specified by the platform administrator. Indeed, communication and protocols are going to depend on the underlying run-time and model to which URSO is going to be mapped to. For instance, in a URSO-conform version of SCA, the binding between two components using a given protocol will depend on the support of this protocol by the SCA runtime.

There is a lot of work in the WCET estimation applied to modern computer architectures. After establishing the control flow graph of the Service Operation, a simple algorithm that sums the costs of the edges in the most costly path is carried on. However, mechanisms such as branch prediction and pipeline must also be considered to avoid a too pessimistic WCET estimation, which may result in large idle slots, and deny the admission of components and tasks that could have been executed in those slack times. Some techniques could be applied at run-time to identify those slack times and rearrange the timing information available in the model.

Many works have been dedicated to performance estimation of Java applications based on their byte-code. Using byte-code as a WCET indicator is considered an efficient way to keep portability [Bernat *et al.* 2000]. The Java Optimized Processor (JOP) is an open-source bare-metal JVM implementation, which comes with a static WCET analysis tool [Schoeberl & Pedersen 2006]. This tool has been extended in [Pitter 2008] to support multiprocessor systems communicating through a shared memory. Just as the algorithm described in this section, the static analysis adopted by JOP creates a control flow graph representing the different blocks of byte-code instructions in the application. Since fixed execution times are assigned to micro-codes, the exact execution time of each block can be calculated. In our approach though we take into account the fact that services may be replaced or unknown at design-time, and we replace the service invocation execution times by variables, whose values may change dynamically. These changes may affect other service dependencies and components, triggering other service replacements. Another tool, called TetaJ (for "Tool for Execution Time Analysis of Java bytecode") [Frost *et al.* 2011] statically analyses Java programs by modelling its control flow as networks of timed automata and applying model checking techniques on them. Contrarily to JOP approach, it targets software implemented JVMs. The underlying JVM micro-code and hardware actions are modelled as well and considered in the model checking. It could be an alternative to improve the precision and portability of the timing analysis in URSO, which for the moment is based on measurements to estimate the execution time of each micro instruction.

A very important part of the dynamism-related work is attributed to the administrator. For the moment, it is up to him to perform mode change and substitution of technical components. It would be interesting to allow the definitions of rules to automatize these

two mechanisms based on information acquired by monitors.
URSO components are passive entities, which means that they can not instantiate new threads or processes by themselves. A tool may be added to the framework to ensure at deployment that there are no calls in the object code towards threading libraries. Parallelism can be obtained in the service composition by using the parallel (||) activity. Parallelism is subjected to available resources.
In configuration and reconfiguration phases, the URSO framework verifies the satisfaction of component dependencies. If a dependency is mandatory and not satisfied, the components are not started. If the dependency is optional, then the component can be started, but some services may be disabled due to service-level dependencies. That implicitly defines a *life-cycle* for both components and services; components and services may be VALID or INVALID and services may be REGISTERED or UNREGISTERED. Services from INVALID components are always UNREGISTERED. This life-cycle may be refined to support more component and service states.
URSO time descriptions are based on relative times (that is, time metrics are based on an initial event, usually the beginning of the execution). Integrating a more refined time model, like that of CCSL[34] could make time description more flexible and expressive. Another alternative would be to adopt the time model and the user-defined clocks feature of the RTSJ (version 1.1), which can easily be generalized to be used in other technologies [Wellings & Schoeberl 2011].
Models @ runtime enables the construction and update of a model representing the architecture of a running system [Garlan & Schmerl 2004, Morin *et al.* 2009]. An interesting feature for URSO monitor components would be to offer a graphic representation of the current architecture and allow application administrators to interact with the application through this interface (for instance, right-click on a component and select a "disable" option). This interface could also be used to notify errors, control the platform mode and modify the platform active policies.
A more formal alternative for the specification of business processes would be using the Process Specification Language (PSL) [Schlenoff *et al.* 1999]. PSL is a language specified by the NIST[35] whose core is based on concepts specified by logic axioms and ontologies. Timing relations can be defined between these concepts, what makes PSL suitable for defining real-time processes. Services could be seen as special activities or objects. On the other hand, due to the fact that business processes may be defined by actors without a logic or formal methods background, the use of BPEL (or a subset of it) remains more intuitive and straightforward.
We consider that this is a first step towards a standard for flexible and modular real-time applications, and that there is certainly a lot of room for improvement in the URSO metamodel.

3.4.3 Summary

This chapter has presented URSO, a component metamodel for the design of real-time service-oriented and component-based applications. URSO focuses on the flexibility and

[34] CCSL (Clock Constraint Specification Language) is a language annexed by the OMG along with UML-MARTE profile to specify logical and chronometric time constraints [Object Management Group 2008]. CCSL time model is based on the models used in concurrency theory. Several works have proposed semantics for the language, enabling both specification and verification of timing constraints [André & Mallet 2009, Mallet *et al.* 2009, Mallet & Andre 2009, Boulanger *et al.* 2012].

[35] NIST stands for "National Institute of Standards and Technology". It is an American institute which provides standards and technology to improve competitiveness and innovation in the US industry. More details about the NIST in `http://www.nist.gov`.

modularity aspects of these applications; independent components may communicate with each by means of services, both being able to be deployed dynamically. Part of the real-time behaviour of an URSO application depends on the information added by the application developer - semantic restrictions over application parameters and service properties must be informed in the Service description. The time specification of the services is automatically deducted by the URSO platform, which adds this information along with the Service description and uses it afterwards to carry out performance estimation analysis. URSO also depends heavily on the manual platform description. This information is used to execute mechanisms such as resource reservation and logical partitioning to achieve temporal and spatial isolation.

URSO do not intend to establish a new model to develop applications, but a unified metamodel defining concepts and good practices that can be mapped onto existing industry-adopted models. In the next chapter, we are going to present mappings from URSO to three models currently used in the industry to design real-time and/or service-oriented applications: the OASIS specification Service Component Architecture (SCA), the SAE[36] avionics standard Architecture Analysis and Design Language (AADL) and the UML profile for Modelling and Analysis of Real-time and Embedded Systems (UML-MARTE).

[36]SAE stands for 'Society of Automotive Engineers'. More details about SAE at http://www.sae.org.

Chapter 4

Mapping URSO Onto Existing Component Models

"Discovery consists of seeing what
everybody has seen and thinking what
nobody has thought."

Albert Szent-Györgyi

Contents

4.1 Motivations . . . **96**
4.2 URSO and SCA . . . **96**
4.2.1 Overview of SCA . . . 96
4.2.2 Extending SCA . . . 97
4.2.2.1 Dynamic Extensions for SCA . . . 98
4.2.2.2 Real-time Extensions for SCA . . . 99
4.2.3 Mapping extended SCA to URSO . . . 104
4.3 URSO and AADL . . . **105**
4.3.1 Overview of AADL . . . 105
4.3.2 Extending AADL . . . 108
4.3.3 Mapping extended AADL to URSO . . . 110
4.4 URSO and MARTE . . . **113**
4.4.1 Overview of UML-MARTE . . . 113
4.4.1.1 MARTE Foundation Package . . . 114
4.4.1.2 MARTE Model-based design package . . . 115
4.4.1.3 MARTE Model-based analysis package . . . 117
4.4.2 Extending UML-MARTE . . . 117
4.4.3 Mapping extended UML-MARTE to URSO . . . 120
4.5 Summary and Discussion . . . **120**
4.5.1 Overview on the extensions to SCA, AADL and MARTE . . . 120
4.5.1.1 SCA extensions for URSO . . . 120
4.5.1.2 AADL extensions for URSO . . . 121
4.5.1.3 UML MARTE extensions for URSO . . . 122
4.5.2 Discussion . . . 122
4.5.3 Summary . . . 124

4.1 Motivations

In the current design of real-time systems, software engineers are accustomed to start the development cycle by using high-level general-purpose modelling languages (either textual or graphic, such as UML). In a next step, this high-level representation of the application is often mapped onto a proprietary programming or modelling language, with its own concepts and semantics. Once they enter this technology-dependent world there is no turning back: they use tools and techniques that can not be generalized and applied to other languages.
URSO aims to fill this gap for service-oriented and component-based real-time applications, providing a metamodel that can be mapped onto other technologies. In order to be conform to this metamodel though, some models must be extended, as they do not contain in their standard form all the concepts present in URSO. Thus, we have chosen to map URSO onto extensible modelling languages, in which new features can be added effortlessly.
In this chapter, we present the mapping between URSO and three standard modelling languages: Service Component Architecture (SCA), Architecture Analysis and Design Language (AADL) and the UML profile for Modelling and Analysis of Real-time and Embedded Systems (UML-MARTE). For each of these modelling languages, an introductory interview exhibits its main characteristics. Then, we list the extensions which were made to the model to address properties important to establish the conformance relation. Finally, we detail the conformance relation between the model's concepts and URSO's concepts.

4.2 URSO and SCA

4.2.1 Overview of SCA

Service Component Architecture (SCA) [Beisiegel *et al.* 2005] is a component model created in 2005 by a group of major software vendors (such as IBM and Oracle). This group was called the Open SOA Collaboration and it aimed to create a model to facilitate the development of service-oriented applications. The core of the specification can be divided into four parts:

- *Assembly* model, which defines service and component description, application architecture, component packaging and vertical and horizontal composition. The assembly is performed in a file separate from implementation code, which increases reuse;
- *Client and Implementation* model, which defines how services can be accessed and components can be implemented for different programming languages. The dissociation between the implementation models and the assembly is what allows SCA to be technology-agnostic;
- *Binding* specification, in which different communication protocols to access component services are defined. Just as the Client and Implementation model, it improves technology-independence in SCA;
- and the *Policy Framework* model, which defines code-independent policies and intents for quality-of-services aspects.

In the assembly model, the main entities are the *components*. Components can be configured and instantiated as *implementations* in a specific language. Implementations may implement or not business functions. These business functions can be exposed as *services* by the components they implement. Services are composed by operations which can be accessed by the client components. They are described by means of *interfaces* which depend on the

technology used to implement the component (Java interfaces for Java implementations, WSDL for BPEL implementations and so forth). In order to consumer a service, a component declares a *reference* towards the service interface, containing the needed operations. Services and references are linked by means of abstractions called *wires*. SCA components can be combined into larger logical constructs named *composites*. It is inside the composites where component wires are listed. Furthermore, component services and references can be *promoted* to become composite services and references, which allows them to be visible to other composites in the runtime. Composite assembly is described into a configuration file which uses a XML-like language called SCDL (Service Component Definition Language).
Besides the assembly model, another major part of the SCA specification is technology-independent: the Policy framework model. It handles SCA non-functional aspects in two ways. The first is by means of policy sets, which, as the name suggests, hold one or more policies. A policy is a constraint or capability which can be applied to either a component or an interaction between components. Policies are expressed in a concrete form. For instance, Encryption is a policy for service interactions, that can be specified by means of WS-Policy assertions. The second way is through *intents*, which are configuration-independent and abstract high-level forms of policies. For example, *Confidentiality* is an intent. Whenever a developer needs confidentiality, he knows that he must use a policy set containing encryption, among other policies.
SCA applications may contain one or more composites. Component implementations, descriptions and artefacts necessary for their execution are packaged into a deployment unit called *contribution*. The standard format for contributions is a ZIP archive. Contributions contain a file listing all deployable composites. At run-time, composites and components are contained within a *domain*, which in the specification is defined as a set of "SCA frameworks from a specific SCA vendor and managed by a single group". Domains specify, among other aspects, how distributed components communicate. Although composites are confined in one domain, they may communicate with applications (and potentially components) outside their domain through *binding* protocols defined on their services and references.
The 'Client and Implementation' and 'Binding' specifications allow developers to extend the SCA model to support new types of service implementation languages (and consequently service interface description languages) and binding protocols.
Figure 4.1 presents a simplified view of the SCA metamodel in UML. Policy Framework elements were omitted. In fact, components, composites, services and references may specify policy sets and intents. Service callbacks[1] and property values were omitted as well. The black classes indicate the extensions points.

Several implementations and frameworks for SCA are available, including open source projects such as Apache Tuscany[2], Fabric3[3], the already extinct Newton, and OW2 FraSCAti [Seinturier *et al.* 2009] projects.

4.2.2 Extending SCA

As we may see, SCA already provides modularity by means of components and composites. It lacks however two important aspects which are present in the URSO metamodel: dynamism and real-time support.

[1] Callbacks in SCA are used to specify two-way and asynchronous services.
[2] Available at `http://tuscany.apache.org`
[3] Available at `http://fabric3.org`

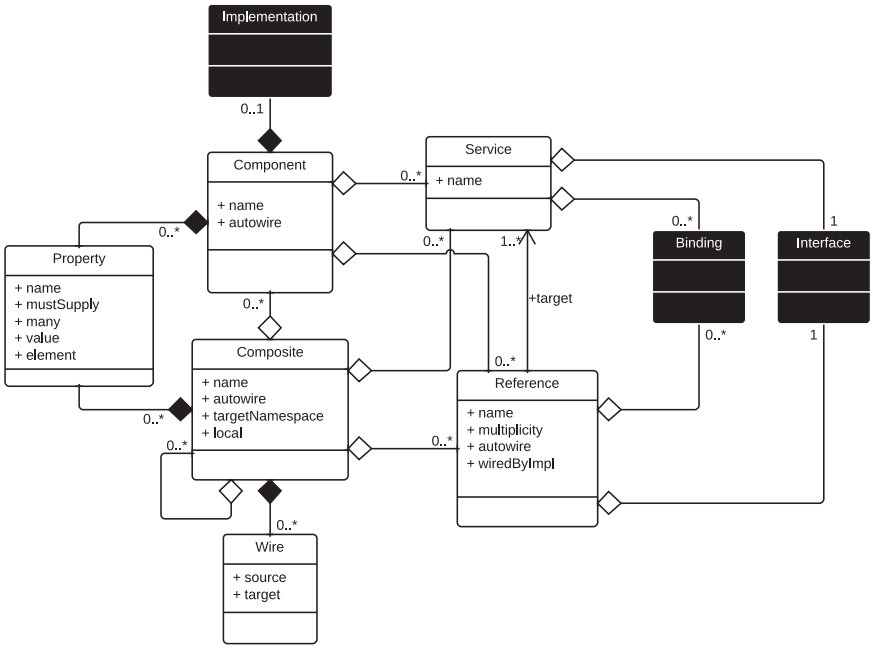

Figure 4.1: SCA Metamodel

4.2.2.1 Dynamic Extensions for SCA

Although SCA claims to be service-oriented, service and references are bound by means of static wires declared in the composite file. Composites may enable the "auto-wire" feature to automatically bind services and references whose service interface match. This is not enough to achieve a dynamic behaviour though: composites are described in a static configuration file and their constitution cannot be changed at run-time, so the wiring can be established off-line, before execution.

This solution does not work when we deal with several composites, which are not supposed to see each others services unless they are included in a larger composite. An alternative for different composites could be adding them to the top-level domain and promote their services and references so that the runtime decides what to do with them. Performing an auto-wire feature is one of the *suggested* behaviours, but that mechanism is an implementation-dependent and domain-specific feature.

We have proposed in [Américo & Donsez 2012] an extension to SCA to handle SOA-inherent dynamism, supporting service selection and dynamic availability aspects. In order to stay conform to SCA's metamodel, we proposed a binding extension (**binding.dynamic**)[4]; it makes sense, since it is constitutes a communication style that can be used by SCA services and references, independently of the technology with which they were implemented. The extension addresses:

- **Dynamic publication and discovery**: By adding the `binding.dynamic` extension, components may have their services published (when put inside a `service` element) or

[4]Extensions in SCA are named according to the following notation: <extension point>.<technology>. Standard extensions are implementation.java, implementation.C++, implementation.bpel, implementation.composite (when a composite corresponds to the implementation of a component), interface.java, interface.wsdl and binding.ws.

discovery other composites' services implementing a given interface (when put inside a `reference` element). In the former case, service properties may be added with a `property` sub-element; in the latter case, filters may be specified to perform service selection through the `filter` element;

- **Component life-cycle**: We have proposed a life-cycle for SCA components supporting dynamic availability of components and dynamic deployment features. Besides being VALID or INVALID according to the presence of services satisfying components references requirements, components can be blocked upon the invocation of a service whose suitable implementation is not available (the BLOCKED stated);

- **Service-level dependencies**: Service-level dependencies allow to select which services will be published according to the satisfied dependencies. They can be specified through `dependency` elements in the service's `binding.dynamic` element;

- **Life-cycle management callbacks**: Life-cycle management callbacks are useful if the developer wants specific actions to be performed upon the binding (removal) of a reference or the (un)registration of a service. A handler method can be specified for each life-cycle event;

- **and SLA support**: SLA terms can be added to services and references. These terms can be considered, negotiated and monitored by a service-level manager (SLM) in the platform. The terms can be added through the `sla` element in both services and references.

The extensions have been implemented by means of a tool that mapped SCA components onto iPOJO components, since the latter supports natively most of the features to support dynamic availability. Figure 4.2 depicts in gray the metamodel of the dynamic binding extension for SCA.

4.2.2.2 Real-time Extensions for SCA

There are few works on the extension of SCA to real-time applications. Martínez *et al.* [Martínez *et al.* 2009b] have extended SCA to support the expression of timing restrictions on services. In order to analyse complex service compositions, SCA services were extended to inform the references they depend on and in which order they use them. In addition, the authors have defined a formal semantics to translate their real-time extended version of SCA to a timed automata, so that model checking techniques can be applied to analyse the predictability and temporal correctness of the system. The extensions were only represented in a graphic form and were not integrated in the SCA metamodel.
Aghajani *et al.* proposed a dynamic real-time service framework based on the extension of SCA with real-time properties and OSGi awareness [Aghajani & Jawawi 2012]. Timing properties (period and priority) and events are associated to components themselves, which may be passive or active. Meanwhile, SCA interfaces for services and references are listed in a separate bundle descriptor.
Finally, [Brugali *et al.* 2012] suggest an integration between Orocos, a hard real-time component framework used in robotics, and SCA. This integration would be mainly performed through a binding extension to enable the communication between Orocos and SCA components (for instance, translating messages sent from one to another). Other suggested mismatches to be solved are the communications' synchronization type (it is not possible to choose a specific mechanism in Orocos, whereas SCA supports both synchronous and asynchronous communication modes) and the lack of vertical composition in Orocos.

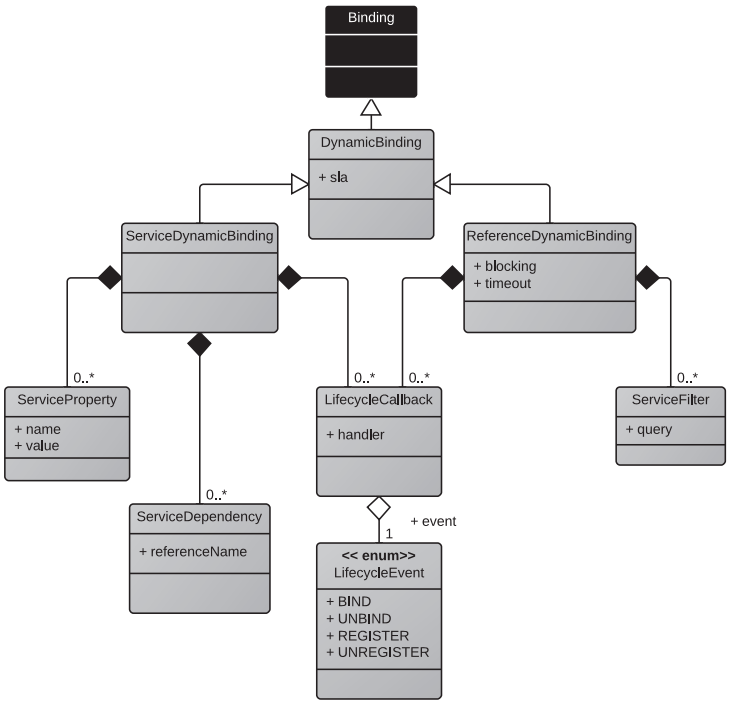

Figure 4.2: Extended SCA Metamodel: Dynamic Binding Extension

In thesis we propose an extension which cover two extension points of the SCA specification (namely service interface and client implementation) and third aspect which is not very exploited in SCA: the deployment file.

Interface extension We propose to extend the SCA interface so that timing and behavioural information may added by the developer and by the framework itself. We will call this extension `interface.urso`. Here we will represent the information in a XML-like file, but any other format (including annotated Java interfaces or extended versions of JSON, WSDL or USDL) could be used. Figure 4.3 shows a tree representation of the elements of our SCA interface extension. It can be integrated directly in the SCA composite description or in a separate file to increase reuse.

As any SCA interface, `interface.urso` contains a list of operations; each operation has its own input and output parameters. In addition, service operations may inform restrictions under the form of logical assertions on its variables. Restrictions can be defined over the input and output variables of a service operation, which can be easily known by the framework; internal (local) variables could have been added as well with a special notation if the execution framework is implemented in a language with introspection features. Information about the service dependencies present in each operation can be retrieved as well by means of introspection; since we know all the service dependencies interfaces for each component, every time we match a virtual method call towards that interface we can consider it as a service call. That information is important for execution time estimation; if the framework implementation language does not enable introspection, service dependencies for each service operation must be listed in the operation description, along with the quantity of times they are executed in the worst case scenario. It is important as well to assign actions to be

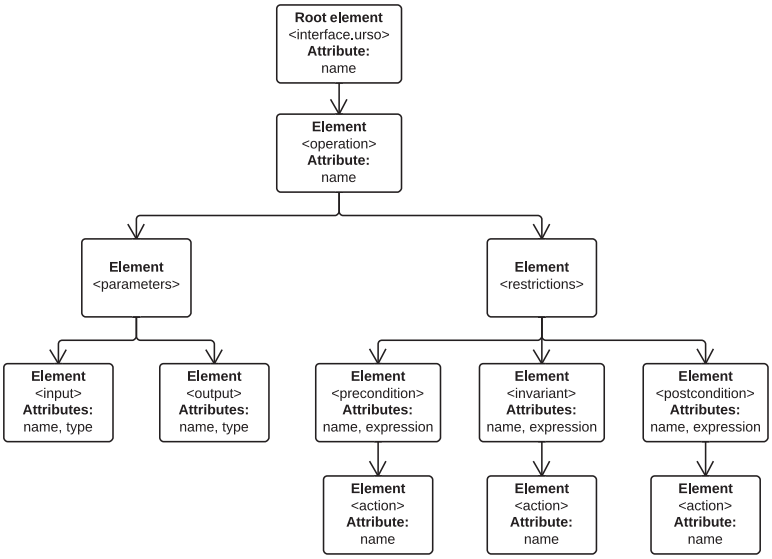

Figure 4.3: SCA Real-time Service Interface Extension

performed in case of violation of the restrictions; in this case, references towards platform actions can be used.
Restrictions can be implemented by adding verifications on throughout of the operation code through instrumentation. More particularly in Java, this can be achieved through aspect-oriented programming [Kiczales *et al.* 1997] and Java assertions[5].

Implementation extension Component implementations must be extended to declare the resources they need for their execution. These resources must be among those listed by the platform and active in the current platform mode. In addition, for quantifiable resources, it is necessary to have enough available resources for the deployment of the implementation; otherwise, it should not be deployed. Our implementation extension elements are depicted in Figure 4.4. Just like in the interface case, the implementation may be described in a separate file, which will be referenced in the composite description.

Deployment Extensions The most important information about our extension is contained in the deployment description. SCA deployment description only lists the name of the deployable composites and enables the explicit import and export of namespaces. We propose to extend the SCA contributions deployment mechanism in two steps: first, producing a *platform description* containing the list of resources, machines, nodes and connections; then, for each contribution, creating a *deployment plan* mapping component instances and tasks to platform entities. The concept of SCA contribution itself was widened as well: a contribution may now contain Component implementations, Composite descriptions (with no implementations), Service Compositions, Tasks or a combination of the precedent elements. Not all of them require a deployment plan: only Component Implementations

[5]Assertion is a mechanism introduced in Java 1.4 which allows developers to test assumptions about variables in a program. More information about Java assertions is available at `http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html`. Before that, a dedicated language to express design by contract constructs in Java, named JML (Java Modelling Language), had been created in 1999 [Leavens *et al.* 1999].

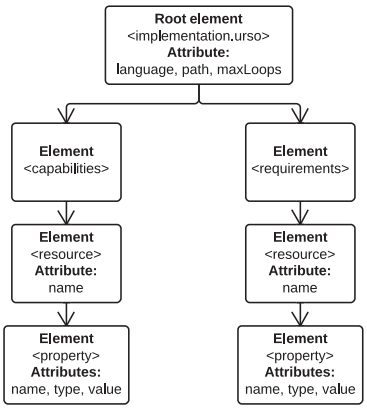

Figure 4.4: SCA Real-time Component Implementation Extension

and Tasks do. Composites inside a contribution will be automatically deployed if all their internal components can be instantiated and all references/dependencies can be satisfied. Concerning Service Compositions, executable code generation will only happen when a Task declares to use them. Then a composite is created to wrap the service composition, and the framework finds the set of services that better satisfy the task timing constraints.
Figure 4.5 shows the elements of the platform description extension. We assume core property and data types can be imported from a file or declared in the platform description itself. With this platform description we are now able to control the available resources in the platform and to implement resource reservation mechanisms. In addition, the information about the machine connections and the protocols with which they can be linked helps us to better estimate execution time.
Figure 4.6 depicts the elements present in the second step of the deployment extension: the deployment plan. SCA metamodel limits the number of Component Implementations to one per component. Since out Implementation extension includes implementations' resource capabilities and requirements, the only data needed to instantiate Composites is the Composite name. The platform will try to deploy all the composite components on the specified node; if it is not possible, the platform may suggest a new placement or just inform that it is not possible to instantiate the composition in that node. In the Task deployment case, the service composition wrapper composite only contains a component generated by the platform, so there is only one component to be instantiated. So if the desired placement is not possible, the platform may behave just like in the composite counterpart. Both node name and machine name can be specified: if only the latter attribute is used, the platform will try to place the artefact in any node of the machine with the given name; if the former attribute is used, the platform will try to find the node with the given name and deploy the artefact over there. In the case where both attributes are used, the node name has priority over the machine name, so if the node placement fails, the platform can still try to place in the specified machine.
The metamodel of the extended version of SCA Assembly, with both real-time and dynamic extensions can be seen in figure 4.7. The figure does not depict the deployment extensions (platform description and deployment plan) for two reasons: first, it would make the diagram more confuse due to the big quantity of information; second, deployment and assembly are two different concerns that should not be mixed, since one is application-independent and the other depends on the underlying platform; and finally, the resultant deployment

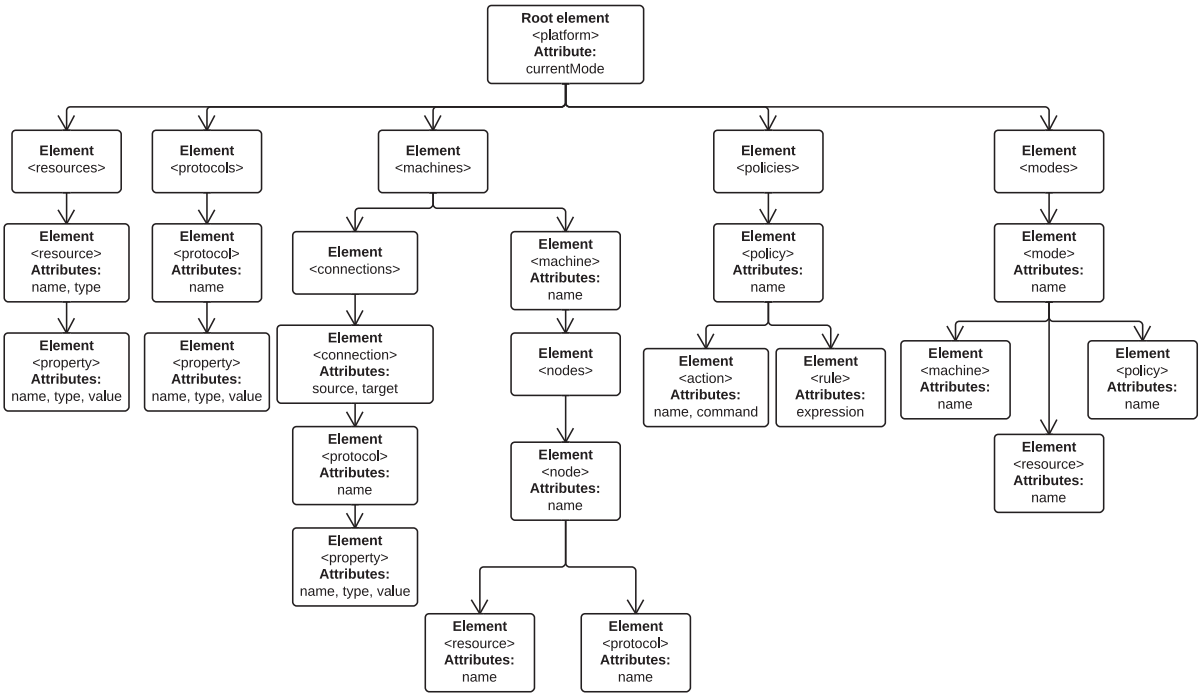

Figure 4.5: SCA Real-time Deployment Extension - Platform Description

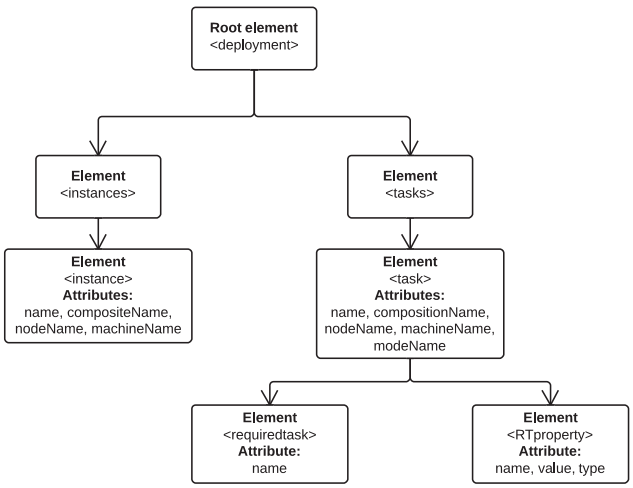

Figure 4.6: SCA Real-time Deployment Extension - Deployment Plan

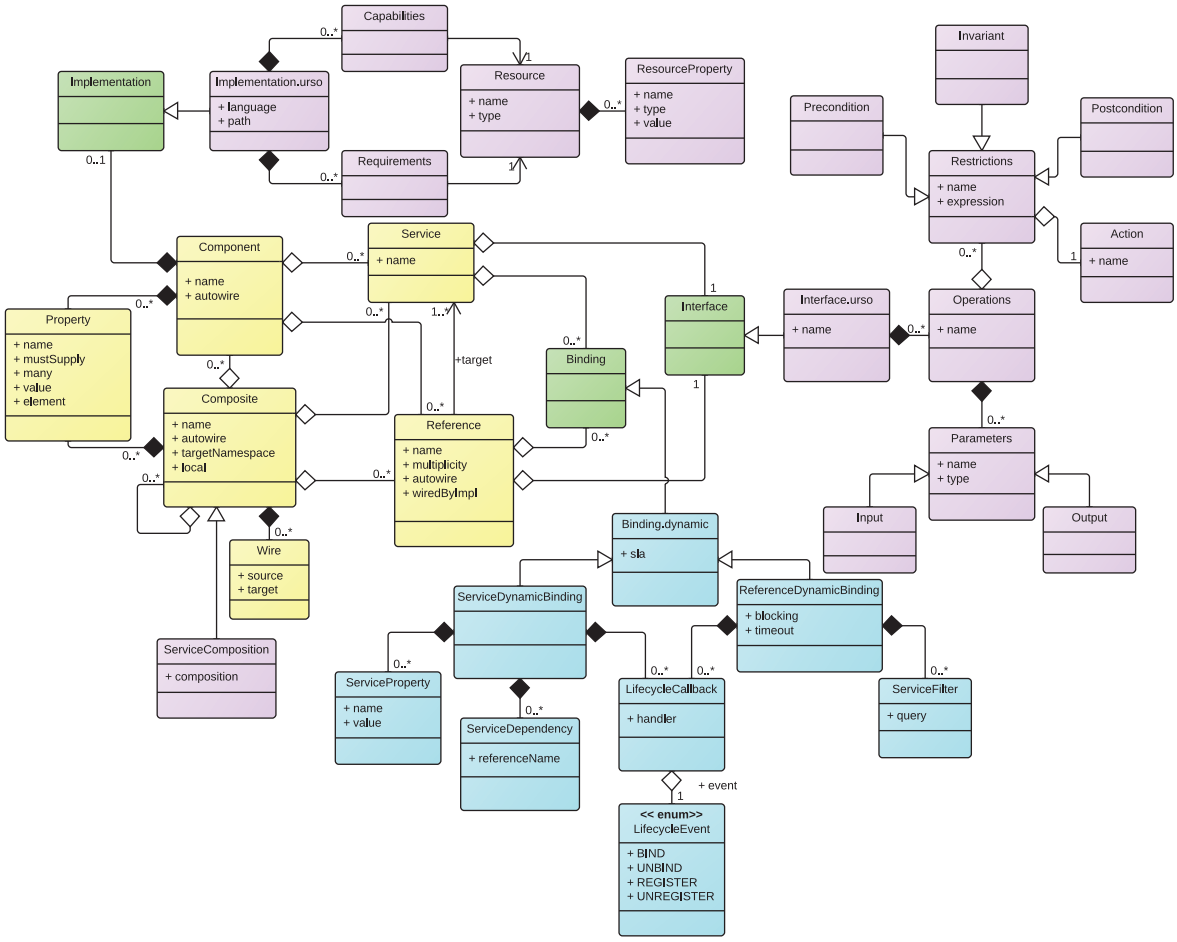

Figure 4.7: SCA Assembly with Real-time and Dynamic Extensions

model is very similar to URSO's Deployment concern diagram (since deployment is a primary concern of SCA, we had to add all the lacking concepts). As we discussed before, Service Compositions can be seen as a special composite, which only imports references.

4.2.3 Mapping extended SCA to URSO

The mapping between SCA concepts and URSO concepts is very straightforward. Since SCA is a service-oriented component model, its assembly model is already very similar to URSO's Assembly concern model. The extensions we have provided in the last section added the missing concepts so that both models can be easily mapped to each other.
Most of the concepts present in URSO's assembly model are already present in SCA. The only concepts that needed to be added were those related to service selection (filters and service properties) and service-level dependencies. Table 4.1 summarizes the correspondences between URSO and (Extended) SCA Assembly models.
URSO's Behaviour model can be mapped onto the information that is both described in the SCA interface extension (operations, parameters and restrictions) and gathered by the platform (instructions and timing properties). The concepts present in URSO's Deployment concern can be mapped onto the information from both platform description (platform,

URSO Concept	Extended SCA Concept
Service	Service
Service.name	Service.name
Service.description	Service.Interface
Service.properties	Service.BindingDynamic.properties
Service.bindings	Service.bindings
Service.dependencies	Service.BindingDynamic.dependencies
Exported Services	Promoted (Composite) Services
Dependency	Reference
Dependency.name	Reference.name
Dependency.description	Reference.Interface
Dependency.multiplicity	Reference.multiplicity
Dependency.ranking	Reference.BindingDynamic.Filter
Dependency.bindings	Dependency.bindings
Imported Dependencies	Promoted (Composite) References
Binding	Binding + Wire
Service Description	Interface + InterfaceUrso
Component	Component
Component.name	Component.name
Component.services	Component.services
Component.dependencies	Component.references
Component.implementations	Component.implementation
Component.properties	Component.properties
Implementation	Implementation + ImplementationUrso
Composite	Composite
Composite.name	Composite.name
Composite.expServices	Composite.services
Composite.impDependencies	Composite.references
Composite.components	Composite.components
Service Composition	Implementation.bpel or another composition language

Table 4.1: Mapping between URSO and SCA Assembly

machine, node, interconnection, resource, platform mode and policy) and deployment plan (task and instance) extensions.

4.3 URSO and AADL

4.3.1 Overview of AADL

The Architecture Analysis and Design Language (AADL) [Feiler *et al.* 2006] is a component model developed by the Society of Automotive Engineers (SAE) and published as a standard in 2004. It was designed for the specification and automated analysis and integration of real-time systems. AADL enables modelling properties such as timing behaviour, schedulability, fault tolerance, security and many other aspects, thanks to its extensibility feature. The AADL core standard contains textual language for the specification of components, with defined semantics. In addition, four annexes have been added to the standard, including a graphical notation, compliance rules for specific programming languages, APIs

for interaction between applications and the execution environment, interchange formats for AADL system representations, and a behaviour model to express high level composition concepts. The specification is currently in its 2.1 version.
AADL model covers not only the real-time software architecture but also the underlying computer hardware and the interfaces between the application and the physical world. Statements in AADL are called *declarations*. Declarations inside *packages* are global to all other packages; otherwise, they remain local to the program specification. These declarations can be classified into four types:

- *Component types* specify the functional interface of a component, that can be seen by other components. They can be characterized by means of *features* (a part of the interface through which control and data can be exchanged), *flow specifications* (an externally observable flow of information throughout a component), *modes* (operational states, like in traditional real-time systems) and *properties*. Features can be required and provided, and they can refer to either *ports* (more specifically *data*, *event* or *event data* ports), *subprograms* (that is, synchronous calls between threads, which can be parameterized by means of subprogram parameters) and shared *access* to component internal data;

- *Component implementations*, as the name suggests, describe components' internal structure in terms of *subcomponents*, *connections* between subcomponents' features, *flows* (directional information transfer between a sequence of connected components or subcomponents), *modes* and *properties*. A set of connections from an ultimate source to an ultimate connection is called a *semantic connection*, which represent a specific pattern of data and control flow between components;

- Feature Group Types, which characterizes groups of features;

- and Annex libraries, which may define annex-specific sublanguages whose constructs may be added to component type and implementation declarations.

Component types and component implementations model are used to describe both hardware (also called execution platform components) and software components of the system. The following categories of components are supported by the AADL standard:

- *Abstract components*, which may be refined into any other category;

- *Software components*

  - *data components*, which corresponds to static data in a source code. This data can be shared by other components through access protocols specified as properties. Their types can be modelled in AADL by data component type and data component implementation declarations;

  - *subprogram components*, which represent a callable set of instructions in a source code that are executed sequentially. Subprograms can be invoked by other components (*e.g.* threads or other subprograms) and may access (or provide access) to data components. Subprograms can be grouped into *subprogram groups* (*e.g.* a software library);

  - *thread components*, that model potentially concurrent tasks in a source code. Just as subprograms, thread components can contain subprogram and data components, whose access can be provided or required. Thread execution is managed by schedulers, and its dynamics can be modelled by means of automata.

Some standard release protocols are defined for periodic, sporadic and aperiodic threads, but additional protocols can be defined. Threads can be grouped in *thread groups*, that is, a group of threads within a process which may share data and common properties;

  - *process components*, which provide enforced spatial isolation through address space partitioning and runtime protection. Process must contain at least one thread component. In addition, they may contain thread groups and provide or require access to data components.

- *Execution platform components*
  - *memory components*, which represent random accessible storage support (*e.g.* ROM and RAM memories). Memory subcomponents may be nested inside larger memory components. They have properties such as size and number of addressable locations;
  - *bus components*, that represent channels through which data and control can be exchanged between processors, devices and memory components. Buses may support different protocols. Buses may access and be connected to other buses. Virtual data transmission channels can be modelled as *virtual buses*;
  - *processor components* correspond to the platform elements responsible for the schedule and execution of threads. They may support different scheduling protocols. In addition, they contain memory components and access bus components. Hierarchical schedulers and virtual machines can be modelled by *virtual processors*;
  - *device components* correspond to physical world entities, like sensors and actuators. They may also represent a software that simulates physical devices. Devices are associated to processors (on which driver software may be executed) and connected to other software components. Devices use bus components to send data;

- System components which are *compositional components*, *i.e.* they permit the hierarchical organization of both software and execution platform components and their communication through well defined interfaces. System components may contain all components described above.

AADL is a hierarchical component model; just as components, subcomponents can be classified by means of component types and component implementations. A component may contain many subcomponents.
Properties in AADL are used to represent attributes and characteristics of AADL model elements, such as latency, deadlines, protocols, and so forth. They have a name, a type and a value. The standard defines some pre-declared properties and property types, but additional ones can be introduced through property sets.
Some elements in AADL may be partially specified. These template elements can later be parametrized and refined to assign values to properties and introduce additional elements such as features and subcomponents. This template mechanism is called *prototype* in AADL.
Figure 4.8 depicts a simplified version of the AADL metamodel. In the original model, the component type and implementation concepts had several subtypes corresponding to the component category; depending on its category, a component may or not have a given feature. Its category also limits the category of its subcomponents; it would be wrong, for instance, for a sub-program component to have a memory subcomponent. Most of these

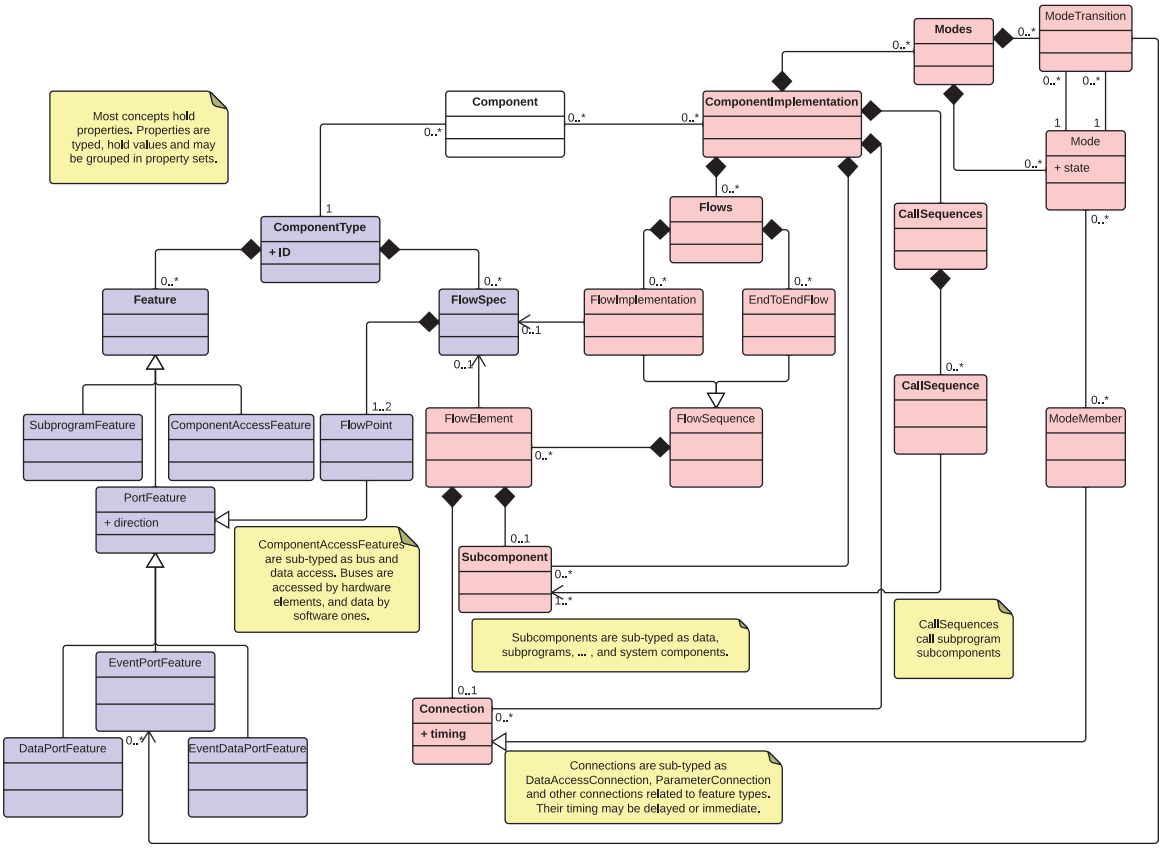

Figure 4.8: AADL Simplified Metamodel

details will be abstracted: they will be necessary though for mapping other models towards AADL.

4.3.2 Extending AADL

Differently from SCA, AADL already includes deployment and real-time-related concepts. AADL lacks however the dynamic and loosely coupled architecture from the SOA approach. Indeed, modes and mode transition are the only native mechanisms that may provide dynamism in the architecture of AADL applications.

In [Aminpour *et al.* 2011], the authors provided a means to model and analyse SOA architectures in AADL. Three distinct types of AADL processes can be specified: service providers, service requesters and discovery agencies. Three operational modes were defined: initialization (initial mode which activates the system), query (where the requesters query the discovery agency) and interaction (where data is exchanged between service providers and requesters). The service provider and the discovery agency are aperiodic processes. Adding and querying for services is modelled as aperiodic threads in the discovery agency. The Service provider communicates with the discovery agency via an event data port through which service registration is performed. The service requester is modelled as a periodic process, with an aperiodic thread which may be used to receive service data from service providers and a periodic thread that is used to request services to the discovery agency. The

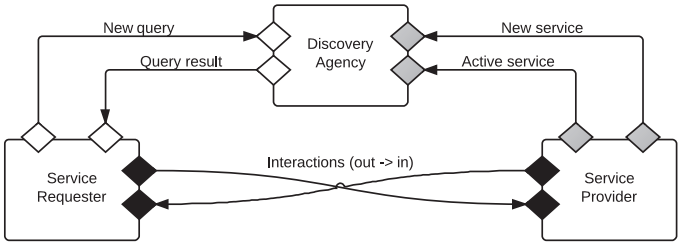

Figure 4.9: SOA model in AADL

communication between the service requester and the discovery agency is done by means of two event data ports: one, from the requester to the agency, sends queries; the other, in the opposite direction, sends query results. Figure 4.9 illustrates the processes and how they communicate with each other. Links between white diamonds symbolizes connections that are only active in the query mode, whereas links between black diamonds are connections active in the interaction mode. Grey diamonds links are connection active in both modes (all the time).

The weak point of this approach is that the set of provided (and required) services is statically informed in the service provider (and request, respectively) process. In addition, the links between all processes are statically defined, making it impossible to add new unforeseen services at runtime. This limitation is actually related to AADL itself and not to the authors' approach. Still, some of the ideas can be reused for a dynamic service-based approach:

- A component corresponds to an AADL process;
- Each service is a thread subcomponent and their operations can be implemented as thread subcomponents;
- Service requesters have two periodic threads which set the system operational mode: one for interaction with service providers and another for querying for services in the discovery agency.
- We can use modes to define which service operation is called through mode-dependent connections. That would mean however that only one component at a time can access a service and that would block the entire component (only one mode can be active at a time);
- Differently from extended SCA, where services are seen from the outside and can be automatically registered in the service catalogue, here, services are explicitly published by the service provider into discovery agencies. That means that there may have two levels of service visibility, just as in URSO. However, auto-binding (or auto-wire, in SCA) is not supported by AADL.

Based on the authors' approach, we have defined some special types to adapt their solution to dynamic systems:

- A data type `URSOServiceRecord`, in which the reference towards the interface file and the set of service properties are declared;
- A data type `URSOServiceQueryRecord`, in which a reference towards the interface file and a filter/ranking function are declared;

- A process `URSO` and a process implementation `URSO.i`, in which we declare a *global service registry* and a *global message multiplexer* thread subcomponents. This implementation must be extended by URSO composites;

- And a thread group `URSOComponent` which must be implemented by URSO components.

With these elements in mind, we may now propose a mapping procedure.

4.3.3 Mapping extended AADL to URSO

We make two assumptions in this mapping proposition. First, the global service registry (GSR) thread component is a special component, implemented in a language which allows adding and querying data dynamically (for instance, a piece of code which manages data in a centralized database). This component is able to add data concurrently, due to the fact that service records are independent from each other. That way, all components that desire to register their services may do so through the same port. It is important though that the component registers the component implementation name, so that it can send this information in the query result, and the service requester may find its required services. The component has an AADL description just as the local service registry (LSR), with ports corresponding to service addition, removal and query operations.
Second, we assume that the global message multiplexer (GMM) thread component is a special component as well, which enables several components to be linked through the same in and out ports. Which component will send and receive information is determined by the active mode (and connections) in each system. This component has an AADL description with two port groups, namely `interactIn` and `interactOut`, and is aperiodically dispatched.
Our mapping procedure, based on the works in [Aminpour *et al.* 2011], works as follows:

- A URSO "deployable" composite is mapped onto an AADL process that extends `URSO.i`, the URSO process implementation that contains the GSR and GMM thread group subcomponents. By "deployable composite", we mean a composite which is not component of any other composite (in SCA, that would be a composite in its composite file which is not included in any other composite file). This process will be named `URSOComposite.<`*`composite name`*`>`;

- Each component in a composite is modelled as a thread group subcomponent. This subcomponent will be declared as a `URSOComponent.<`*`component name`*`>`. We have chosen to model components as thread groups because besides data, thread groups are the only component category which may have subcomponents of its same category. It is important to model the recursive nature of components. In addition, each composite has a thread subcomponent, named LSR, of type `ServiceRegistry.<`*`composite name`*`>`;

- For each non-exported component service, we create event data port connections between the component and the LSR. Each service implies three connections (between the `newService`, `removeService` and actService ports). These connections are active in two modes - a query and an interaction mode particular to the component (named `query<`*`component name`*`>` and `int<`*`component name`*`>`);

- For each non-imported component dependency, we create event data port connections between the component and the LSR. Each service implies two connections between

the `newQuery` and `queryResult` ports. These connections are active in one mode - the query mode particular to the component (`query<`*`component name`*`>`);

- Between every pair of components, we add a connection between one's `interactOut` and other's `interactIn` port groups. This connection will be responsible for the exchange of data between the components and will be active in the modes `int<`*`one component name`*`>` and `int<`*`other component name`*`>`;

- Just as in the non-exported services case, for each exported service we create three connections between event data ports of the component and the GSR. The three connections (again between the `newService`, `removeService` and actService ports) are active during both `queryG<`*`component name`*`>` and `intG<`*`component name`*`>` `modes`.

- For each imported dependency, we also create two event data port connections (between the `newQuery` and `queryResult` ports) between the component and the GSR. These connections are active in the `queryG<component name>` mode;

- For each component that import dependency or export services, we connect the components `interactIn` and `interactOut` port groups to the GMM `interactOut` and `interactIn` port groups, respectively (always in the direction from one out port to other's in port). This connection is active during the `intG<`*`component name`*`>` mode;

Listing 4.1 shows code snippets illustrating the custom data types, the URSO and URSO-Composite processes, the URSO.i implementation and the URSOComponent thread group. Listing 4.2 depicts the component type and implementation for the registries. Local and global service registry have the same component type, but the implementation might be slightly different, due to the multiple connection factor we have mentioned earlier. A composite implementation declaration example has been included in the Annexes section A.3.
Provided services in component implementations are modelled as threads. Each thread has a particular mode indicating that it is being executed in the component. Inputs from the interactIn port group can be redirected to service threads as input; symmetrically, service threads' output can be redirected to component's interactOut ports.
Required services are implemented by adding two periodic thread subcomponents: one for querying services in the service registry and another for consuming the service itself. These threads change the system mode to query and interaction modes, respectively. Component type declared ports are linked to its respective component implementation ports. Section A.3 includes the implementation declaration of one of its components.
Just as in SCA, service compositions can be seen in our AADL mapping as special types of composites which only request services globally. The expression representing the composition may be put in an external file and referenced as a property in data type.
Service-level dependencies can be modelled through event ports. Upon the reception of a query result, a dependency may send an event to the `actService` port of a given service thread, so that its thread operations may be activated.

Listing 4.1: URSO AADL Process and Data Declarations

```
data URSOServiceRecord
  -- Includes interface file
  -- it must also include service properties
end URSOServiceRecord;

data URSOServiceQueryRecord
  -- We must have the interface file and the filter
end URSOServiceQueryRecord;

process URSO
end URSO

process URSOComposite extends URSO
  features
    initialized: in event port;
end URSOComposite;

process implementation URSO.i
  subcomponents
    GSR: thread group ServiceRegistry.global ;
    GMM: thread GlobalMessageMultiplexer.i ;
end URSO.i

thread group URSOComponent
  features
    newService: in out event data port URSOServiceRecord;
    removeService: in out event data port URSOServiceRecord;
    actService: out event port;
    interactIn: port group (basic::Input_PT);
    interactOut: port group (basic::Output_PT);
    newQuery: in out event data port URSOServiceQueryRecord;
    queryResult: in out event data port URSOServiceRecord;
end URSOComponent;
```

Mapping software elements to hardware is natively supported by AADL. Components will be deployed on the platform execution elements of its system and processor. In order to analyse whether a system has enough resources to execute its software elements, execution platform components and software components may declare their capacity and budget respectively. The platform may then sum the budget totals and compare with the system capacities. Moreover, non-functional properties, such as policies and protocols, can be specified by means of AADL properties.

Figure 4.10 illustrates the structure of services, dependencies and discovery agencies (service registries) in a URSO-compliant AADL application. Service registries have three thread subcomponents, one for service addition, one for service search and one for service removal. Service providers structure corresponds to a service provided by a component. It has as many threads as it has operations in its service description. Service requesters structures corresponds to a service required by a component. Service requesters have two threads, one for periodically requesting services (this thread can be used as well to verify the availability of the consumed services), and one for effectively consuming services. A communication mechanism may be created if a task requires more than one service dependency.

Figure 4.11 depicts the inclusion of service-related concepts (services, dependencies and service registries) into components. Components have as many service providers as they have services and as many service requesters as they have dependencies. Besides, they have an internal service registry where services and dependencies which are not exported or im-

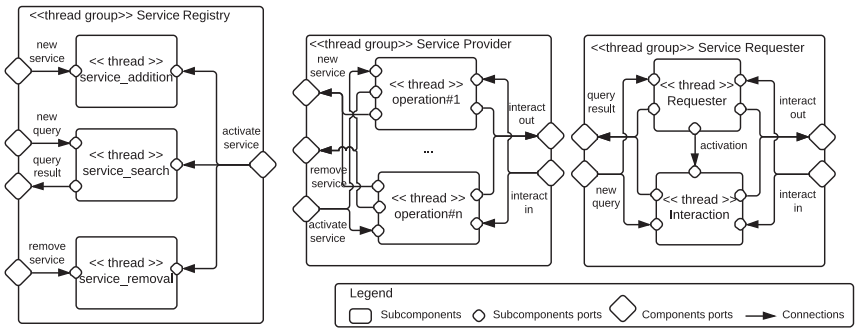

Figure 4.10: AADL URSO Extension: Service-related concepts

ported can be published. This registry can also be used by their inner components which export or import services. Components also have a message multiplexer, which redirects messages depending on the current mode. Deployable composites can be themselves an AADL process, with a structure similar to that of components.

Listing 4.2: URSO AADL Registry Thread Declaration

```
thread group ServiceRegistry
  features
    newService: in out event data port URSOServiceRecord;
    removeService: in out event data port URSOServiceRecord;
    actService: in out event port;
    newQuery: in out event data port URSOServiceQueryRecord;
    queryResult: in out event data port URSOServiceRecord;
end ServiceRegistry

thread group implementation ServiceRegistry.local
  subcomponents
    Service_addition: thread ServiceAddition;
    Service_search: thread ServiceSearch;
    Service_removal: thread ServiceRemoval;
  connections
    event data port newService -> Service_addition.newService;
    event data port newQuery -> Service_search.newQuery;
    event data port removeService -> Service_removal.removeService;
    event data port Service_search.queryResult -> queryResult;
    event port actService -> Service_addition.actService;
    event port actService ->   Service_search.actService;
    event port actService ->   Service_removal.actService;
end ServiceRegistry.local
```

4.4 URSO and MARTE

4.4.1 Overview of UML-MARTE

The UML profile for MARTE[6] [Object Management Group 2008] adds capabilities to the UML2 standard for specification, design and verification of real-time and embedded systems. It was designed to replace the extinct UML profile for Schedulability, Performance

[6]MARTE stands for Modelling and Analysis of Real-time and Embedded Systems.

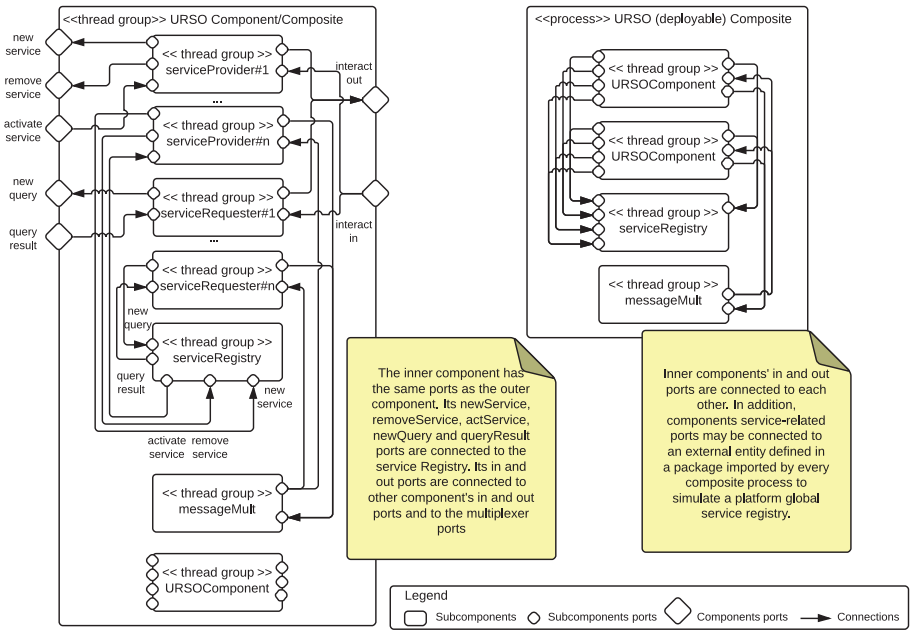

Figure 4.11: AADL URSO Extension: Component-related concepts

and Time (SPT). Both software and hardware aspects can be independently modelled in MARTE. Constraints are expressed in OCL and quality of service characteristics use the UML profile for Modelling Quality of Service and Fault Tolerance (QoSFT) Characteristics and Mechanisms.
The MARTE specification has two major parts: one dedicated to the application and hardware design model (*MARTE design model* package) and another addressing the analysis model (*MARTE analysis model* package). Both parts share common concerns (MARTE foundations package) related to time and non-functional descriptions and resource management. Annexes to MARTE include specification languages such as CCSL (Clock Constraint Specification Language) and VSL (Value Specification Language) and data type declarations.

4.4.1.1 MARTE Foundation Package

The MARTE Foundation Package contains foundations for modelling and analysing real-time systems. It is composed by five clauses:

- *Core Elements* package, which as the name suggests, contains the concepts used to define every other concept in the MARTE specification. It defines Model Elements, which may be Classifiers (element type) or Instances (element instance). In this package it is also possible to model the system (basic and operational mode-dependent) run-time behaviour and semantics. Behaviours may be specified through different mechanisms supported by UML, such as automata and Petri nets. In addition, the package describes mechanisms to activate mode transitions and behaviours through actions and event-based triggers and to communicate between concurrent units through requests;

- *Non-Functional Properties* (NFP) package, which describes a framework for specifying

non-functional properties in UML-MARTE models. Property values are set through the Value Specification Language (VSL), an annex of the MARTE specification. NFPs can be defined as qualitative or quantitative, the latter being defined in terms of sample realizations and statistic functions over these realizations. Elements may also be annotated with mode-dependent restrictions defining minimum level of quality of service and the value space for its NFPs;

- *Time* package, which enables modelling time and time-related concepts and mechanisms for real-time applications. It defines three classes of time abstractions: causal/temporal, in which the only preoccupation is instruction precedence and dependency; clocked/synchronous, in which time is divided into a discrete succession of slots (events occurred in the same time slot are considered as simultaneous or coincident and communications in the same slot are considered as instantaneous); and physical/real-time, in which duration real time values precision and accuracy are necessary. Time concepts are associated to behavioural, events and objects. Complex time structures can be establish through relations between different time lines (called time bases). The package also defines clocks, which are structures related to one time base through which time measures can be accessed. Clocks may trigger events. Timed elements (such as timed behaviours, executions or constraints) benefit from this mechanism;

- *Generic Resource Modelling* (GRM) package, which contains constructs for modelling both hardware and software executing platform at different detail levels. Resources offer services, which are described by behaviours. Timed and non-timed NFPs may be specified for resources. Resources also can be specialized in sub-types (memory, processor, clock, etc.). The resource reservation can be performed through the standard resource services acquire and release. Allocation and de-allocation of resources are managed by the Resource Broker, whereas its creation, maintenance and deletion are managed by the Resource Manager. A scheduler is seen as a special type of Resource Broker and the schedulable entities are the resources it keeps. This package also enables the specification of resource usage, that is, consumption demands of an amount of a given resource. A resource usage may be static or dynamic, depending on its temporary nature;

- *Allocation* modelling package, which models the allocation of applicative elements onto the execution platform and its resources (spatial distribution). Since the scheduler is a resource broker, temporal scheduling aspects are also addressed by this subpackage. The timing information in both application and execution platform entities must be taken into account during the pairing. It is also possible to refine models with different levels of abstraction. For instance, by mapping an application on a platform, it is possible to refine back the timing model of the application to deduct its WCET. Allocation maps individual application elements to individual execution platform elements, so it is expected that the application keeps its structural and behavioural consistency after the allocation process.

4.4.1.2 MARTE Model-based design package

The MARTE model-based design package specializes the foundation concepts to enable modelling real-time systems. It is composed by three clauses:

- *Generic component model* (GCM), which enables modelling MARTE applications following a component-based approach. The generic component model is not associated to any specific semantics, and real-time concerns are added by another package. It

is based on UML structured classes and SysML blocks. Structured components are self-contained entities which encapsulate data and behaviour and whose properties can be used as interaction ports and assembly parts. They are linked through delegation (vertical composition) or assembly (horizontal composition) connectors to other components' interaction ports or assembly parts. Components interact via message passing, and these messages may represent operation calls, events or simple data. The interaction ports of both ends of a connector must be compatible in regards to features (client-server) or flow specification and direction (data-flow).

- *High-level application modelling* (HLAM), which provides high-level modelling abstractions and features for real-time systems. Applications are composed by at least one Real-time Unit (*RtUnit*) which is similar to an active object in UML. RtUnits may own several schedulable resources, which may be created dynamically or not (pool). They are executed autonomously and may be executed concurrently and can be seen as system processes or task servers. In addition, RtUnits may invoke services from one another, communicate through message passing and define operational modes, behaviours and properties regarding the scheduling of their schedulable resources. Shared data is modelled through protected passive units (*PpUnits*). PpUnits provide services that are accessed by a schedulable resource of a RtUnit and whose concurrent access policies can be configured through properties. Messages in RtUnits are stored in a queue and processed according to its defined policies. Actions that are part of a communication may be declared as real-time actions and contain properties like deadline, period, synchronization type and message size.

- *Detailed Resource modelling* (DRM), which details the modelling of both application and platform resources and specializes the concepts presented in the GRM clause. It is divided in two sub-clauses:

  - *Hardware resource modelling* (HRM), which specifies a high-level and generic framework for modelling hardware elements and platform architectures. The model is composed by two complementary views: one is centred on hardware physical properties and the other that classifies hardware according to their functional and logical properties. These two views specialize a more general hardware model. The *general* hardware model is composed by hardware resources which may encapsulate other hardware resources and provide and/or require services. In the *logical* view, resources are categorized as *computing*, *storage*, *communication*, *timing* or *device* resources. Each of these categories has its own sub-resources (which may or not be in the same category) and properties (*e.g.* computing resources may be specialized into processors, ASICs[7] or PLDs[8], and are linked to storage-type resources). In the *physical* view, it is possible to model the resource's size, position, shape, power consumption and other physical properties. It contains two sub-packages, *layout* and *power*. Layout provides mechanisms to physically describe the platform architecture, whereas power provides means to annotate layout elements with power-related information.

  - *Software resource modelling* (SRM), which specifies a set of constructs to model real-time applications as a set of concurrent units that interact through mechanisms (API) provided by a RTOS, which acts as execution support and is in

[7] ASIC stands for Application-specific integrated circuit, which denotes a highly-specialized and efficient integrated circuit, but whose flexibility is very limited.

[8] PLD stands for Programmable Logic Device, which denotes digital circuits whose functions can be reconfigured.

charge of real-time features. It is divided in four packages: *core* (which presents the concept of Software Resource, gathering the resource itself, its manager and services provided by both concepts), *interactions* (data flow communication and execution flow synchronization resources, which act according to specific policies), *concurrency* (that defines Software Concurrent Resources, which compete for computing resources at physical and logical levels to execute their instructions, with an execution context and an execution flow) and *brokering* (resources that manage other hardware and software resources, their allocation, configuration, access and so forth).

4.4.1.3 MARTE Model-based analysis package

The MARTE model-based analysis package extends the foundation concepts to enable analysing real-time systems. It is composed of three clauses:

- *Generic quantitative analysis model* (GQAM), which gathers common concepts from different analysis domains and may specialized onto new analysis models. This model imports concepts from the NFP, Time and GRM packages. Analysis are based on the description of how behaviours use system resources, and based on some input NFPs, it may produce a set of output NFPs concerning aspects such as feasibility of the task set, application timeliness, or the total power consumption. The analysis model is centred on the concept of analysis context. An analysis context takes into account the end-to-end system behaviours of a workload, the platform resources necessary for the execution of those behaviours and a parametrized expression, which defines what is being considered for the analysis. In addition, it defines observers, which are entities that define predictions and requirements for measures of a given expression on an interval between two events in a behavioural model.

- *Schedulability analysis model* (SAM), which extends the GQAM to specifically address schedulability analysis. It also defines a set of NFPs related to this concept. The workload behaviour corresponds therefore to an end-to-end processing flow, with time-related observers for requirements (*e.g.* deadlines, and maximum jitter) and predictions (*e.g.* latencies, jitters and other scheduling metrics). Furthermore, the platform resources descriptions is focused on processing resources and properties related to the communication between different schedulable resources.

- *Performance analysis model* (PAM), which extends the GQAM to describe the analysis of temporal properties such as throughput and delay. The techniques used to analyse performance often include simulation and formal models such as Markov chains and Petri nets.

Annex section A.4 presents a summarized version the metamodels of the different packages present in the UML-MARTE profile.

4.4.2 Extending UML-MARTE

MARTE presents most of the fundamental concepts presented in the URSO metamodel. Its core elements and non-functional properties packages enable modelling application's behaviour; its resource and allocation models can be used to address deployment issues; its generic component and high-level application models can model most of the concepts of the assembly concern. However, one important aspect that is clearly not natively supported by the UML-MARTE profile is services-oriented computing [Aziz *et al.* 2013]. Components

connect to each other by means of static client-server or data-flow ports. It would be necessary to find a mechanism so that components can provide services (operations) which can be published and discovered in a broker-like structure, as is the case for resources. Substitutability and dynamic reconfiguration are not covered by the MARTE profile either.
In [Marcos *et al.* 2011], the authors have incorporated some of MARTE features to enable real-time SOA applications modelling. They have used NFPs to enrich service implementation and specification description and GRM concepts such as scheduler, storage and processing resources to model infrastructure aspects. Dynamic SOA mechanisms such as service publication and discovery are not addressed though. Similarly, the SOA4DERTS framework proposed in [Muhammad *et al.* 2012] adopted the NFP package concepts to model non-functional properties and SoaML [9] to model the service interface, but dynamism and substitutability techniques are not handled either.
A different approach was proposed in [Alhaj & Petriu 2010]. The aim of the work was to generate a performance model from SOA systems modelled in UML. The authors have integrated several models to express different concerns of the application:

- A work-flow model (like BPEL, BPMN or an UML activity diagram) is used to express business processes. The same model can be used to express the activities internal to a given service;
- An ADL or component model is used to express the structure of the system (participant modules, participant ports and the connections between them);
- A deployment model which describes mapping between software resources and physical/hardware resources. It also describes the complete infrastructure of the execution platform;
- A NFP model which can annotate all models cited beforehand.

Despite not detailing dynamism and substitutability mechanisms (thus, possessing the same flaws as the solutions presented above), this approach has the same structure as URSO's and will be the basis for our MARTE extension proposition. While real-time- and platform-related aspects will be described with MARTE GCM, component- and service-related aspects are going to use adaptations of both GCM and SoaML concepts.
In summary, components in SoaML are called *participants*. Participants may either provide or require services through their *ports*. Provided or required services are described by *service descriptions*, which may be classified into *simple/UML interfaces*, *service interfaces*, and *service contracts*. Simple interfaces design one-way, anonymous service interactions without callbacks. Service interfaces can be bi-directional, associated to a given protocol, and callbacks to the service consumer may be informed. In addition, in service interfaces, the service provider may describe an expected interface for the service consumers. It is also possible to combine several simple interfaces onto one service interface. Service contracts focus on the service specification, the roles of both service consumer and provider, the service choreography and the terms of the communication established between both parts. It can be used to design multi-party service choreographies (service interactions with multiple providers and consumers). Ports are also associated to service *capabilities*, which are abstractions used to represent functions and/or resources of services independently of their implementing participant. Thus, service providers may use capabilities to generate an

[9] Service Oriented Architecture Modelling Language (SoaML) [Object Management Group 2012b] is a UML standard profile defined by OMG through which SOA applications can be described. The profile is in its 1.0.1 version and upcoming extensions are expected to integrate other service-related OMG extensions such as BPMN.

implementation-independent view of their services and publish this view on service agencies. Similarly, service consumers may use capabilities to inform the functions needed for a given required service, without specifying a particular candidate service. The communication path between requests and services is called service channel, and is completely transparent from a participant point of view.

Thus, we have created the concepts of URSO Service and Dependency, which refine both MARTE GCM 'Component' and SoaML 'Participant' concepts. Services and Dependencies are associated to SoaML Service and Request ports. Each port is associated to a Service Interface, which corresponds to URSO's Service Description, and a capability, which represents services' dynamic availability. Components and their services and dependencies are connected through new types of connector, *service* and *dependency* connectors, respectively. Services and exported services, just as Dependencies and imported Dependencies are connected through another special type of connector, a *promotion* connector. Figure 4.12 depicts the URSO extensions and their relation with MARTE and SoaML stereotypes[10].

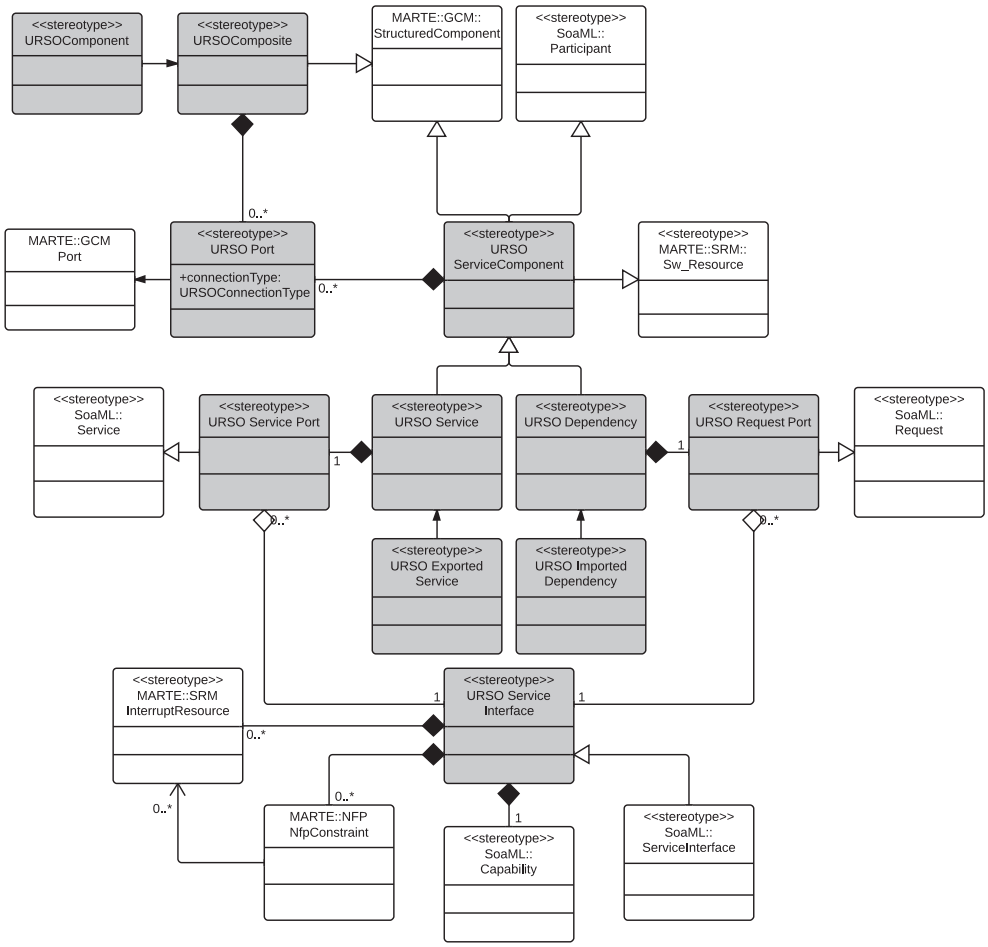

Figure 4.12: URSO Extensions for MARTE, based on SoaML profile

[10]Stereotypes are an extensibility mechanism in UML. Through UML stereotypes it is possible to define new elements derived from the existing ones, but with specific properties suitable for a particular domain [Object Management Group 2006c].

4.4.3 Mapping extended UML-MARTE to URSO

As stated in the previous section, UML MARTE natively supports concepts from URSO Deployment concern. A URSO **platform** would correspond to a set of (software and hardware) resources, allowed modes and allowed mode transitions. The set of resources would include all URSO **resources** (which can be specialized accordingly to their type, *e.g.* storage resource for memories and computing resource for processors units), the set of **machines** (and other physical resources, with their respective owned resources), and the connections between these machines. It is also possible to consider other technical components as resources, like schedulers and monitors. URSO Instances correspond to passive protected units (PpUnits), and the behaviour associated to it corresponds to the implementation of the components that form the composite. This behaviour may have resources associated to it, just as implementations inform the resources they use in URSO. A task can be seen as a schedulable resource (which is a specialization of concurrency resources). Each node with computing resources owns a Real-time unit (RtUnit) with which all allocated threads are associated. Service compositions associated to tasks are execution behaviours.
In the Behaviour concern, an URSO operation corresponds to a method in a SoaML interface. Extra information, such as constraints and actions, can be mapped to NFP Constraints and Interrupt resources, respectively, over the URSO service interface. The internal behaviour of the method is extracted by the platform itself, just as properties concerning operations' partial WCET. Other service properties can be included in URSO service interface, as it inherits the property specification support from SoaML Service Interfaces.
In the Assembly concern, components and composites correspond to MARTE GCM's structured components. Hierarchical relations can be specified through delegation connectors. Services, dependencies and service interfaces match the URSO extensions described in the previous section for the same concept. Filters, service properties and supported binding protocols can be added as properties in the service interface. Static bindings can be performed by means of assembly connectors between GCM components.
To conclude, the deployment of tasks and instances on execution platform elements can be performed by means of allocations whose ends point towards the software element and the platform resource.

4.5 Summary and Discussion

In this chapter, we have presented extensions to SCA, AADL and the UML MARTE profile so that they match the concepts present in the URSO metamodel. These extensions were based on ongoing research in the field of real-time service-oriented applications. This section summarizes these extensions and discusses their applicability in real systems.

4.5.1 Overview on the extensions to SCA, AADL and MARTE

In the next paragraphs, we present an overview of the extensions presented in this chapter. It details the motivations to extend each model and what approach was used to do so.

4.5.1.1 SCA extensions for URSO

SCA (Service Component Architecture) is a component model developed by major IT vendors to ease the development of service-oriented architectures. SCA is extensible and new interface, binding and implementations types can be developed according to the developers' needs. Despite based on the service concept, SCA does not fully support the publish-

discover-bind SOA interaction pattern. Indeed, SCA components are statically linked by means of entities called wires, which specify a source service and target reference (the term used to designate service dependencies). In addition, non-functional properties support in SCA is limited. It consists in a set of non-configurable policies, policy sets and intents that can be attached to composites, components and services.
In order to add the support to dynamic composition, we have described in [Américo & Donsez 2012] a set of extensions to SCA bindings. By means of these extensions, it is possible to designate which services must be published and which references must discover services at run-time. In addition, we have added mechanisms to support service selection, life-cycle management, service-level dependencies and SLAs.
For the purpose of supporting real-time applications, we have added extensions to support real-time component implementations, real-time service interfaces and real-time component deployment. Real-time component implementations require the description of resource usage for each implementation. In the real-time service interface extension, we have added information concerning constraints over service parameters and variables. The extensions for the deployment of real-time components include a platform description, with platform resources, modes and nodes, and a deployment plan, which maps tasks and composite instances to nodes in the platform.

4.5.1.2 AADL extensions for URSO

AADL (Architecture Analysis and Design Language) is a SAE standard component model used in the avionics domain. By means of this language, it is possible to specify and perform analysis on hierarchical component-based real-time systems. With AADL it is possible to specify both software and hardware components. All components can be annotated with properties, which will be evaluated during the analysis of the system.
The lack of dynamism in AADL prevents it to fully support the SOA architectural style. It has led us to create prototypes for SOA components based on the approach developed in [Aminpour *et al.* 2011], in which three prototypes were developed: one for service providers, one for service requesters and one for discovery agencies. The authors have developed all prototypes based on AADL process components. Internal mechanisms were mapped to thread components. In addition, the authors have suggested to add all services provided inside the same service provider component, which decreases code modularity.
Based on these observations, we have extended the approach so that it becomes more general. Service operations are mapped to thread components, but services themselves are mapped to thread groups, one per service. Similarly, service dependencies are individually mapped to thread groups. Services and dependencies are contained inside components, which now also have an internal discovery agency and a message multiplexer, which changes the current input and output connection according to the current platform mode. Some components services and dependencies can be exported/imported so that their services can be published/discovery in the discovery agency of its parent component. Finally, components can be contained inside composites (modelled as AADL process components), which here represent a deployment unit. Composites exported services and imported dependencies are published and discovered, respectively, in a global discovery agency and a global message multiplexer. Both component declarations are published in a package and visible to all composites.
These software components can be sub-components of complete AADL systems, directly included inside the declaration of the node in which they are deployed. They may also include information about their resource usage as properties. Restriction, filters and service properties can be added as properties in service provider/requester entities as well. Service

compositions and tasks can be simulated by creating a composite with service dependencies and a special sub-component (thread or program) to represent the composition itself. This sub-component can indicate the file in which the composition expression was stored.

4.5.1.3 UML MARTE extensions for URSO

MARTE (Modelling and Analysis of Real-time and Embedded Systems) is a UML2 profile for the specification and analysis of real-time and embedded systems. MARTE replaces the Schedulability, Performance and Time profile for UML2. It enables modelling both hardware and software aspects of real-time systems detailedly and independently. MARTE can be integrated with other OMG specifications, like OCL for constraints and QoSFT for quality of service characteristics. MARTE is divided into three major parts: a common foundational package, a package dedicated to the design of real-time systems and a package dedicated to the analysis of such systems. The foundation package includes aspects shared by the two other packages, such as the core elements, non-functional properties, time model, allocation and a generic resource model. The design package specializes the generic resource model into software and hardware resource models, and contains a generic component model and a high-level application model. In its turn, the analysis model contains a generic quantitative analysis model which can be used as template for other analysis model. It also includes two derived analysis model, for specification and performance aspects.
The generic component model (GCM) present in MARTE enables components to interact by means of static ports and connectors. Most UML-based approaches for the specification of real-time SOA applications use SoaML, a SOA profile for UML, enriched with the non-functional properties model from MARTE. Since we aim to model applications which are both component-based and service-oriented at the same time, we have added new concepts to GCM to represent services as components that are linked by means of special connectors to the component that provides them. Similarly, dependencies are modelled as components connected to the component that requires them. Service properties, filters and restrictions can be added in the service interface and exposed as capabilities to the system. SoaML does not impose a technology on the publication and discovery of services, leaving it up to the implementation run-times. We assume the service is published and retrieved by means of service registries, just as in the previous extensions.
Deployment concern concepts are natively supported by MARTE. Its detailed resource model enables modelling storage, computing and other resources, describing resource usage and allocating hardware resources to software resources through the allocation and refinement package. Behavioural aspects are equivalent to behaviour and behaviour execution concepts which are present in the core elements package.

4.5.2 Discussion

Since SCA is basically constituted of an assembly model, extending it to support dynamic and real-time systems through its extension points was less complex. However, given the fact that this support is not native, real-time analysis, resource enforcement and many other mechanisms required for the correct and trustworthy implementation of real-time systems must be implemented from scratch and integrated to existing run-times. Although the dynamism part can be directly integrated in the SCA assembly model, most part of the real-time extension information is added by means of separate descriptor files. We have chosen to do so to improve the reuse of such files. Moreover, a key feature is the fact that no technology was imposed on these descriptor files. Real-time service interfaces may be written in Java enriched with annotations, in XML files or in JSON. The important is that

the information contained is conform to that specified in the extension.
The URSO execution model in which components are passive entities that provide services and tasks are active entities that invoke those services fits perfectly SCA. Generally speaking, URSO can be seen as a super set which contains all SCA assembly concepts and specializes them to dynamic and real-time systems. That is not the case with AADL and MARTE, which are more complete models and which have far more concepts than URSO. Mapping and extending more complete models to find mapping relations from URSO is more complex. Furthermore, on account of the fact that they are more complex and have more elements than URSO does, contrarily to SCA, with which is possible to establish a two-way mapping, mappings to AADL and MARTE are one-way. As a consequence, although URSO applications can be mapped to extended MARTE and extended AADL, not all MARTE and AADL can be mapped to URSO applications. Using the concepts and structures introduced by the extensions is necessary to be able to establish mappings in the opposite way.
AADL foresees a complex hierarchy of component types. Since we wanted inner components to be treated as their parent components, the component type needed to support sub-components of its own type. Only two sub-component types hold that property: data and thread groups. In view of the fact that thread groups have a higher abstraction level than data components, we have chosen thread groups as our basic unit for components, services and dependencies. Please notice that the same approach was used in the MARTE extension: services and components are described as if they were both components linked by delegation-like connectors[11]. However, it may result in a highly complex connection network between sub-components. Developing conform systems may be very difficult for untrained developers. In addition, it complicates model analysis, since the message multiplexer and discovery agency components have special semantics.
Moreover, the dynamism degree in the AADL extension is smaller than with that of the SCA extension: since nodes are also declared in AADL through declarations, and the components which are hosted in this node are its subcomponents, adding new components to node, would mean changing its declaration file, which could not be possible at run-time without restarting the whole node. Consequently, new components can not be individually added to a running system. It is possible to consider adding new components by adding new machines and new nodes with new resources to the system. These components could normally publish their services in the global registry and interact with already deployed components by means of services.
MARTE extension used concepts from other UML profile, SoaML, an emerging profile for SOA-based systems. We have borrowed their 'participant' concept and combined it with MARTE GCM structured components to create special components to represent services and dependencies. These components are also software resources. In MARTE, resources provide and require services, and may interact by means of resource brokers and managers. We can use this mechanism to publish and discover URSO services. Our MARTE extension is not very intrusive and does not require many special arrangements to integrate MARTE semantics and syntax. In addition, UML provides a graphical language which is widely known, more clear than AADL graphic language, and may ease the design of URSO-compliant systems. This graphical language can also be used to provide a model @ runtime representation of the hosted applications.
MARTE's model specifications may be the basis for the integration of new and more detailed time and resource models in URSO. We also consider integrating SoaML service contracts to enable more complex service interactions between service providers and consumers in

[11] In AADL, delegation is expressed through sub-components and connections between component and sub-component ports.

MARTE and in URSO.

4.5.3 Summary

In this chapter, we have presented URSO-based extensions to SCA, AADL and UML-MARTE. These three models consist in widely known standards to develop either service oriented or real-time systems. We aimed to develop extensions based on URSO metamodel so that they can all be service-based and real-time at the same time. Seven principal requirements were identified to establish mapping relations between URSO and these models.

To support a *component* model: All three models natively contain a hierarchical component model in their specifications. In SCA and AADL, hierarchy is explicit, whereas in MARTE it can be achieved through delegation connectors.

To support *service-based* communication between components: Only SCA natively meets this requirements. We have thus developed extensions to MARTE and AADL to allow their components to communicate in a service-oriented fashion. The extensions consisted in creating new types of subcomponents to represent services and dependencies and connect them to their respective components.

To support service *dynamic* publication, discovery and binding: None of three models provides dynamism for service mechanisms. We have added a binding extension for SCA to mark publishable services, besides a service selection support. In the AADL extension, dynamism is limited by the fact that the whole platform constitutes a unique block of declarations, but deployable composites may dynamically interact with each other through a common discovery agency and global message multiplexer. In the MARTE extension, SoaML-based services capabilities can be published and discovered in a resource broker.

To support the specification of *resource usage* on component implementations: MARTE natively supports resource usage specification through its generic resource model. In AADL, it is possible to add resource usage as properties to software entities. In SCA, it was necessary to extend component implementations to describe their resource usage through an external descriptor.

To support the decomposition of the system into *tasks* and assign them *real-time* properties: In MARTE, tasks can be modelled as schedulable units linked to real-time units owned by URSO nodes. In AADL, it is possible to model tasks as thread components. These threads can be associated to a sub-program component which would correspond to a URSO service composition. In SCA, it was added the possibility to define threads through external descriptors and associate them to (already deployed or not) service compositions.

To support the specification of the *execution platform* and its resources: Detailed software and hardware resource specification is natively supported by MARTE. Furthermore, one resource may own another to express hierarchy. Resource amount and access information can be included. In AADL, hardware components can be hierarchically organized into nodes. It is also possible to annotate these components with information concerning their resource amounts and their access policy (shared or not). SCA originally does not support either platform description or resource modelling. Support to both was added in our extension through external files.

To enable the *allocation* of execution platform resources to software entities: Resource allocation is natively supported by MARTE. In AADL, there is no explicit method for such allocation; it is performed considering the execution platform components of the system in which the software system is declared. In SCA, we have added a deployment plan extension in which instances and tasks can be placed in specific hardware resources.

Table 4.2 summarizes the lack of each model in each of the define requirements and how the issue was addressed in the proposed URSO extension.
It is possible to create mappings from SCA to URSO, due to its generality. Since AADL and MARTE are more complex, mappings from these elements to URSO must respect the structures added with the extensions. Both models have interesting concerns (*e.g.* time, resource and extensible non-functional properties models) which may be integrated to URSO in a future version.
In the next chapter, a proof-of-concept implementation of a SCA URSO-compliant lightweight framework, based on the extensions proposed in this chapter, will be introduced. In addition, a use-case based on the application introduced in Chapter 3, the Dynamic Collision Detector, will be detailed. We will also detail its implementation, and test its performance and timeliness in order to validate our proposition.

Features	**SCA support**	**AADL support**	**MARTE support**
Component Model	Explicitly hierarchical	Explicitly hierarchical	Hierarchical (delegation connectors)
Service-Based	Native support	Extension through special components	Extension through SoaML components
Dynamism	Extension through dynamic bindings	Extension through global registry	Extension through service broker
Resource Usage	Extension through external descriptor	Extension through properties	Native support
Task support	Extension through external descriptor	Native support	Native support
Execution platform specification	Extension through external descriptor	Extension through properties	Native support
Allocation	Extension through deployment plan	Native support	Native support

Table 4.2: Overview on the URSO extensions to SCA, AADL and MARTE

Chapter 5

Implementation and Validation

"Think like a man of action, act like a man of thought."

Henri Louis Bergson

Contents

5.1 Implementation **128**
- 5.1.1 SCA:PlatformDesc Command 129
- 5.1.2 SCA:PlatformInfo Command 130
- 5.1.3 SCA:ChangeMode Command 131
- 5.1.4 SCA:List Command 131
- 5.1.5 SCA:Deploy Command 131
- 5.1.6 SCA:Undeploy Command 132
- 5.1.7 URSO+NaSCAr framework architecture 132

5.2 Usecase: Revisiting DCD_x **135**
- 5.2.1 Platform Description 135
- 5.2.2 Service Compositions, Instances and Tasks: the Detector example . 138
- 5.2.3 DCD_j, a Java-based implementation of DCD_x 141

5.3 Validation **144**
- 5.3.1 Methodology overview 144
- 5.3.2 Platform description validation analysis 145
- 5.3.3 Platform information analysis 146
- 5.3.4 Contribution deployment analysis 146
  - 5.3.4.1 Passive entities' deployment 147
  - 5.3.4.2 Active entities' deployment 148
- 5.3.5 Contribution undeployment analysis 149
- 5.3.6 Mode change analysis 150
- 5.3.7 Execution timeliness analysis 152

5.4 Summary and Discussion **154**
- 5.4.1 Overview 154
- 5.4.2 Discussion 155

This chapter presents a proof-of-concept implementation of a SCA-based URSO-compliant lightweight runtime. In the first section, we detail technology-related aspects of the implementation. In the second section, we revisit the DCD_x benchmark application and show its mapping to extended SCA, so that its components may be deployed on the run-time prototype. Then, we use the benchmark application to evaluate timing aspects of the prototype. Finally, we discuss the results and draw conclusions about the modelling and the performance achieved.

5.1 Implementation

In order to implement and validate the contribution of this thesis, we have developed a tool to deploy URSO-compliant SCA applications. This tool is called NaSCAr (for "Not Another SCA Run-time") and was first introduced in [Américo & Donsez 2012]. Originally, NaSCAr was developed to validate the dynamic binding extension to SCA. It parses SCA contribution files and creates equivalent iPOJO components on top of an OSGi Service Platform. The OSGi Platform enables dynamically deploying new modules (called *bundles*) without restarting the platform. It has a service registry in which service implementations objects are stored along with their service interfaces and service properties. Upon a service query, the platform returns a list containing service implementation objects that satisfy the query. Then, service requesters may invoke service operations on these objects. Besides managing service dependencies automatically, iPOJO also enables hierarchical composition. Each composite has its own internal service registry, which is not visible to other composites. Services from inner components can be exported and dependencies can be imported. This behaviour corresponds to that of expected of a URSO-compliant framework, and hence, we have chosen to adapt NaSCAr to support other URSO concerns.
Developers and administrators can interact with NaSCAr through shell commands on the OSGi platform. Originally, NaSCAr possessed the following commands: `sca:deploy <contribution path>`, `sca:undeploy <contribution ID>` and `sca:list`. As the names suggest, `sca:deploy` was used to deploy composites from a SCA contribution on the OSGi platform, `sca:undeploy` was used to remove the composites from a given contribution from the platform and `sca:list` listed all the contributions currently deployed on the platform. We have thus added the following commands to NaSCAr:

- `sca:platformdesc <file path>`: This commands adds the platform description to the URSO framework. It must be executed before any applicative component is deployed. The framework stores the platform description and will be responsible for resource amount update upon resource reservation.

- `sca:changemode <mode name>`: This commands changes the platform operational mode. The name passed as parameter must be a valid mode, already listed in the platform description. During the mode transition, the platform may replace, stop and start new tasks and instances.

- `sca:platforminfo`: This commands displays information about the platform, like its current mode, the available platform modes, resource consumption, tasks and instances deployed, nodes, and machines.

Original commands semantics were modified as well:

- Initially, `sca:deploy` was used to deploy all composites from a SCA contributions. Now, this command may be used to deploy component implementations instances and tasks informed in a deployment plan inside the contribution file.

- Similarly, `sca:undeploy` was used to undeploy composites from a contribution. Now it will be used to undeploy instances and remove tasks.

- `sca:list` now lists not only the composite definitions per contribution, but also tasks and instance definitions it contains.

NaSCAr consists in three sub-projects: NaSCAr-core, NaSCAr-shell and NaSCAr-URSO. We have chosen Apache Karaf 2.3.2 as OSGi target distribution, with an Apache Felix

Framework 4.0.3 core. Commands were implemented based on Apache Karaf shell API and exposed to the framework as Apache Aries Blueprint services. Apache Felix iPOJO core (1.8.0), API (1.6.0) and Composite (1.6.0) libraries were used. Next section details the implementation of each of the framework commands.

5.1.1 SCA:PlatformDesc Command

`sca:platformdesc` command allows platform administrators to enter the platform configuration to the framework. The framework stores this information and uses it to decide whether instances can be deployed or not. At the implementation level, after receiving the command from the user, the NaSCAr implementation retrieves a URSO controller instance, which parses the platform description file and creates an object representation of the description. This object representation is then validated by the URSO controller against a set of restrictions:

1. The platform object must not be null, else there is no platform on which instances and tasks can be deployed;
2. The platform must declare an initial mode. From this initial mode, only transitions declared by mode transitions are allowed;
3. The list of machines in the platform must not be null or empty. Else, there is no target machine on which instances and tasks can be deployed;
4. There must exist at least one node per machine. Nodes are machines' logical partitions to which tasks and instances are mapped. A machine without a node means a machine which can not be used to deploy URSO entities;
5. The platform must have at least one mode. In cases where the platform has only one mode, this mode must be the platform's initial mode. Only transitions informed in the description file are allowed, so modes which are not referenced in any transition are never reached.
6. Machines' capabilities must match platform's capabilities. It is also important to verify whether the resource values do not conflict with each other (*e.g.* a machine declares more resources than those which were declared for the platform).
7. Nodes' capabilities must match machines capabilities. Just as machines resources must not conflict with the resources of the platform that contains them, nodes resources must not conflict with the resources of the machine that contains them.
8. Machines declared in interconnections must match platform machines. Interconnections indicate a link between two machines on the platform. These machines must be listed among the platform machines.
9. Supported protocols in interconnections must match the protocols declared by the platform. Each interconnection lists a set of supported protocols. These protocols must match those supported by the platform. The set of supported protocols in interconnections may change according to operational modes. For each interconnection, at least one supported protocol must be listed.
10. Machines, resources, interconnections, and policies declared in modes must match platform's machines, resources, interconnections, and policies, respectively. In addition, resources declared by machines and nodes in different modes must not conflict with those declared in the corresponding platform machine declaration.

11. Each machine must declare at least one connection: a local connection towards itself. This way, it is possible to characterize the connection between instances hosted in a same machine.
12. Just as interconnections must reference existing platform machines, mode transitions must reference existing platform modes.

According to these rules, the minimum platform description contains one machine (with one node and no resources), with one local interconnection supporting one protocol (with no declared properties), and one mode (the initial one), containing the machine, the node and the interconnection description (which references the supported protocol). The object representation of this platform has 22 URSO objects, one for the platform, one for each named element in the platform description XML file (11), plus the objects representing the machines, nodes, interconnections (with its lists of supported protocols), protocols, modes (plus the machines, nodes and interconnections (with protocols) lists of the initial mode). For performance reasons, XML parsing was implemented with Javolution's [1] XML stream reader library.

5.1.2 SCA:PlatformInfo Command

`sca:platformInfo` command enables administrators to retrieve current information about the platform. For a given platform, it displays:

- Current platform mode name;
- Platform **resource information**. For each resource, it displays resource's name and value;
- Platform **machine information**. For each machine, it displays:
  - Machine's name and description;
  - Machine's resource information (see **resource information**);
  - Machine's **node information**. For each node, it displays:
    - Node's name;
    - Node's resource information (see **resource information**);
    - Node's deployed tasks. For each task, it displays the task's name;
    - Node's deployed instances. For each instance, it displays the instance's name and the name of the composite it represents;
- Platform **interconnections information**. For each interconnection, it displays its participant machines and the list of supported protocols;
- Platform **policy information**. For each policy, it displays its name, its rule's name and expression and its action's name and command;
- Platform **mode information**. For each mode it displays:
  - Mode's resource information (see **resource information**);
  - Mode's machine information (see **machine information**);
  - Mode's interconnection information (see **interconnection information**);
  - Mode's policy information (see **policy information**);
- Platform **mode transition information**. For each mode transition, it displays its ancient and the new modes.

[1] Java library available at `http://www.javolution.org`

5.1.3 SCA:ChangeMode Command

`sca:changemode` command enables administrators to change the platform current operational mode. When the command is received, the platform performs a series of verifications before effectively changing platform's mode. First, it is verified if there is a mode whose name matches the one used as parameter for the `changemode` command. If it exists, it is verified whether the mode indicated is not the current mode (if it is, there is nothing left to do). Else, it is verified whether a transition between the current mode and the new one is specified in the list of allowed transitions. Finally, if it is, the URSO internal controller removes all tasks that are not allowed to execute in the new mode, and add all tasks which are allowed to execute but were not executing yet (not allowed in the previous platform mode). Mode changes put the platform in a reconfiguration state (non real-time). Additionally, if nodes are disabled, it also undeploy the node instances allocated in these nodes. Its execution time depends on both the quantity of tasks and instances deployed in the platform and the quantity of tasks and instance changes to be performed. More details on the performance of task and instance deployment and undeployment are available on the commands `sca:deploy` and `sca:undeploy`.

5.1.4 SCA:List Command

`sca:list` command enables administrators to list all component instances and tasks currently executing on the platform. The instances and tasks are grouped by contribution. Its performance depends on the number of installed contributions, component instances and tasks on the platform. During task and instance deployment, the platform keeps track of which contribution they come from. That information is useful afterwards to undeploy contributions (and consequently, their tasks and instances).

5.1.5 SCA:Deploy Command

`sca:deploy` command enables administrators to deploy new component instances and tasks in the platform. This command takes contribution file paths as inputs, which are used as contributions identifiers (*i.e.* the same contribution can not be deployed twice). The contribution file parsing generates two lists: a list of composite instances, which are instantiated as iPOJO composites in the underlying OSGi platform; and a list of tasks, whose service compositions are transformed into composite instances (which are also instantiated as iPOJO composites) that are called by schedulable elements (threads). These threads are automatically created by the platform.
During the contribution processing, the platform generates partial (represented by an unevaluated expression) or total (represented by a number) WCETs for all methods in the classes contained in the contribution file. These WCETs are kept in a hash map, whose key is the concatenation of the class name, the method name and the method description. For classes implementing interfaces, the interface name is added to the key as well. Each time a WCET is inserted in the WCET table, the platform updates partial WCETs so that they may be evaluated onto total WCETs. The WCET of invocations towards interfaces are estimated by picking the greatest execution time among all classes implementing that interface and method. Since the execution time of a path in the control flow graph may depend on the invocations performed in this path, the partial WCET expression for most methods consists of a max operation among the WCETs off all paths[2]. In addition,

[2]Paths which may be included in other paths in the graph are excluded, as their execution time can be upper bounded by the WCET of the paths that contain them.

OSGi bundles are created separating interfaces from implementation classes, so that service providers and consumers may always refer to the same interface. Bundles exported and imported packages are declared accordingly.
After inspecting contribution's classes, processing is divided into composite instance processing and task processing.
During instance processing, for each instance, the framework checks whether the node onto which the instance should be deployed has enough resources. If it does not, it raises an exception indicating so. If it does, it creates iPOJO components for each composite component and an iPOJO composite to represent the deployed composite. Services and dependencies, whether exported/imported or not, are created accordingly and linked to the interfaces associated to them. When this instance is started, the framework stores a reference towards it associated to the contribution where it came from. This way, when command is given to undeploy a contribution, the framework is able to recognize all composite instances that should be undeployed as well.
During task processing, for each task, the framework first identifies the service composition it refers to. For simplicity sake, instead of generating executable code, service compositions were coded in Java. Their description is similar to that of a component, with one or more dependencies and one implementation class. These implementations have one particularity: they expose a `java.lang.Runnable` service. From service composition descriptions, composite instances are created. After instance creation, tasks are created by a scheduler module. In this module, first the scheduler implementation retrieves the `Runnable` service from a given composite instance through its service composition name. Then, it creates real-time threads (object instances of `javax.realtime.RealtimeThread` class) configured with the properties informed by the task entities (release, priority and deadline). The scheduler then invokes the WCET manager (which contains the WCET table cited beforehand) to check whether the service composition implementation WCET is smaller than its deadline. If it is, it checks whether it is still possible to schedule all its threads if the thread in question is added to the thread set. Then, if the feasibility test is positive, it starts the thread. The framework also associates the thread name to the contribution name, in order to be able to interrupt it and remove it from the thread set when its contribution is undeployed.

5.1.6 SCA:Undeploy Command

In the sca:undeploy, the platform identifies the contribution which must be removed and removes all its component instances and threads with it. For this reason, it keeps two maps: one that associates composite instances to the identifier of the contribution they were declared on, and one similar map for tasks and contributions identifiers. iPOJO composite instances are disposed, and real-time threads are interrupted.

5.1.7 URSO+NaSCAr framework architecture

The URSO compliant NaSCAr implementation was developed in real-time Java. JamaicaVM[3] was chosen as real-time JVM implementation. Some third-party libraries were used as well in the project:

- OW2 ASM library was used to inspect classes byte-code in order to estimate WCET.
- Javolution's collections were used to have a better performance;

[3]JamaicaVM is a RTSJ-compliant Java virtual machine. More information about JamaicaVM can be found in http://www.aicas.com/jamaica.html.

- Karaf's shell commands API was used to implement NaSCAr shell commands. These commands were published in the Karaf framework by means of Apache Aries Blueprint services.
- Apache Felix iPOJO API and Composite libraries were used to create new components and composites dynamically.

The instruction table used by the WCET manager module, can be filled by creating special classes directly with byte-code instructions to measure the average performance of each instruction. In these classes, we execute a byte-code command n times between the execution of two `Clock.getRealtimeClock.getTime()` invocations and then divide the difference between both times by n to get the average execution time. Some commands may require a special manipulation of the Java stack in order to work property (*i.e.* stack pop commands). This approach was based on the works in [Albert *et al.* 2007, Lambert & Power 2008, Schoeberl *et al.* 2010], which describe ways to measure byte-code performance and discuss the use of byte-code as portable WCET estimations.
Figure 5.1 depicts the overall architecture of the prototype. The shell commands are interfaced with the SCA container through a SCAContainer object. Our implementation to SCAContainer, named SCAContainerImpl, was a singleton class, so all command invocations would be executed in the same context. The SCAContainerImpl class contains instances of classes related to both SCA and URSO concepts. The diagram below uses the circles and half-circles notation from UML2 Component diagrams to represent provided and required interfaces.

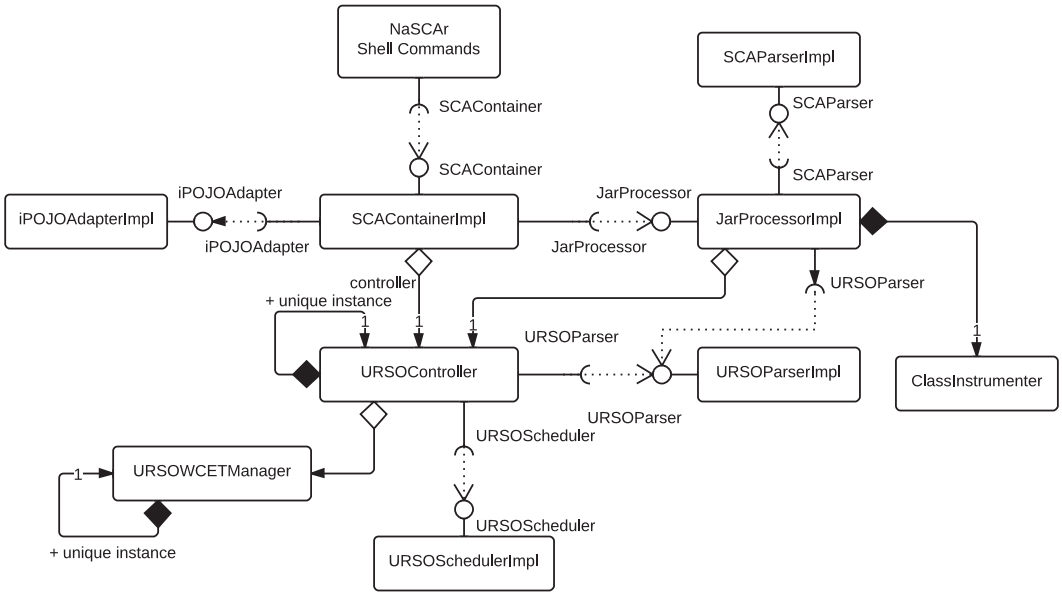

Figure 5.1: URSO+NaSCAr framework architecture

SCA-related classes were present in NaSCAr's original project, as described in [Américo & Donsez 2012]. Among them, we highlight the following classes and interfaces:

- `com.bull.nascar.impl.JarProcessor`: The `JarProcessor` interface (and its implementation `JarProcessorImpl`) contains the method `process()`, which inspects a

`.jar` contribution file and returns an object representation of the URSO Contribution. Our JarProcess implementation uses both URSO and SCA model parsers and an instance of the URSO Controller. It also creates bundles from the contribution files. There is a similar class, ZipProcessor, which does the same processing for `.zip` contribution files.

- `com.bull.nascar.impl.iPOJOAdapter`: The `iPOJOAdapter` class is responsible for creating iPOJO components and composites from a SCA assembly. Its method `adapt()` receives a URSO `Composite` object and a `BundleContext` instance and returns an equivalent iPOJO `CompositeComponentType` object.

- `com.bull.nascar.impl.SCAParser`: The `SCAParser` interface (and its implementation class `SCAParserImpl`) is responsible for parsing SCA-related files and producing object representations of them. Its method `parse()` receives as parameter the path of the file to parse and returns a URSO `Composite` object.

- `com.bull.nascar.impl.ClassInstrumenter`: The `ClassInstrumenter` class is responsible for inspecting classes and checking whether they need to be modified. We consider that a class needs to be modified when it has a service-level dependency. In this case, we add an iPOJO service controller field and two callback methods, one for the case when a service implementation is bound to the given dependency and one for the case when a service implementation is unbound to the dependency. The service controller field is a boolean variable which controls the publication of a given service: when this variable is set to true, the service is published; otherwise it is not. Thus, in the callback methods, we set this variable when the dependency is satisfied, and unset it otherwise.

URSO-related classes were added to manage URSO-related information. The most important interfaces and classes among them are listed below:

- `com.bull.urso.URSOController`: The `URSOController` class is a singleton class which concentrates all methods related to the URSO model life-cycle. It is through this class that the `SCAContainerImpl` has access to platform-related information and the scheduler facilities. The `URSOController` class has methods for platform creation (from a platform description) and validation, mode change, task deployment and undeployment, WCET analysis, and information about the platform and its subcomponents. For this, it uses other URSO-related classes, such as `URSOScheduler` and `URSOWCETManager` implementations.

- `com.bull.urso.URSOMethodAnalyser`: The URSOMethodAnalyser class is used to analyse classes' byte-code instructions and create a control flow graph from them. From this control flow graph, the WCET Manager may derive a WCET expression for class' methods. The method analyser employs the algorithms described in the URSO Behaviour concern section.

- `com.bull.urso.URSOParser`: The `URSOParser` interface (and its implementation class `URSOParserImpl`) is responsible for parsing URSO-related files (platform description, interface description, service compositions and deployment plans) and producing object representations of them (A URSO `Platform`, `Interface`, `ServiceComposition` and `URSOContribution` object, respectively).

- `com.bull.urso.URSOScheduler`: The `URSOScheduler` interface (and its implementation class `URSOSchedulerImpl`) is responsible for adding and removing tasks from the

Metric	NaSCAr	URSO	Shell	Total
Number of classes	11	60	6	77
Number of methods	50	375	6	431
Total lines of code	1221	3456	102	4779
Avg McCabe Cyclomatic Complexity	3.34	3.64	1.167	1.974

Table 5.1: NaSCAr project metrics

executing task set. Its methods `addTask()` and `removeTask()` create threads from tasks and remove the selected threads from the task set, respectively. The method `addTask(Task t)` is overloaded for periodic and sporadic tasks, since each task type has its own scheduling parameters.

- `com.bull.urso.URSOWCETManager`: The URSOWCETManager class is a singleton class that manages WCET-related aspects. It contains methods for creating WCET expressions from control flow graphs, evaluating these expressions and updating the value of these expressions with already-evaluated WCET expressions. A WCET expression is URSO is represented by an Expression object which has an operator (which can be ADD, MAX or NULL, the latter being used to represent evaluated expressions) and two other expressions joint by the operator. The object has also three fields: a `boolean` field *evaluated* to indicate whether the expression has been evaluated (and in this case, the two other inner expressions are null objects), a `long` field *value*, which contains the value of the evaluated expression, another `long` field *coefficient* which is used to multiply the value of the value of an expression (in loops, for instance) and a `String` field which indicates the name of the method that is invoked, if it was not evaluated yet.

Metrics about the Java implementation prototype can be found in Table 5.1. These metrics were obtained by a plug-in[4] installed in the Eclipse IDE. In the table columns, *NaSCAr* corresponds to the NaSCAr sub-project, *URSO* corresponds to NaSCAr-URSO sub-project and *Shell* corresponds to NaSCAr-shell sub-project.

5.2 Usecase: Revisiting DCD_x

In this section, we will retake the collision detection benchmark application that we have described in the Chapter 3 and create a URSO-conform SCA description for it, in order to deploy it in our implementation framework.

5.2.1 Platform Description

First of all, we will create the description of the platform on which the application is going to be deployed. The description depicted in the listing 5.1 corresponds to the description of the machine we have used for our tests. No policies were defined. We have defined two similar modes: on in which all tasks will be enabled, and one in which no tasks will enabled. Since we have used only one machine, there is only one interconnection (machine's local interconnection). For sake of brevity, we have omitted machine's and node's capabilities, since we have declared only one of each and they contain the same elements as platform's capabilities list. We have also omitted the mode description, since its machines, nodes,

[4]More information about the metrics plug-in can be found in `http://metrics.sourceforge.net/`.

capabilities and interconnections are equal to those of the platform.

Listing 5.1: Test platform description

```
<platform defaultMode="Mode1">
  <capabilities>
    <shared-resource name="JVM" type="Software">
      <property name="Brand" type="Brand" value="JamaicaVM"/>
      <property name="Type" type="Type" value="Real-time"/>
    </shared-resource>
    <resource name="RAMMemory" type="Storage">
      <property name="Quantity" type="MemoryInMB" value="4096"/>
      <property name="Available" type="MemoryInMB" value="4096"/>
      <property name="Type" type="Type" value="DDR2"/>
    </resource>
    <resource name="ProcessorUnit" type="Computing">
      <property name="Quantity" type="Core" value="2"/>
      <property name="Available" type="Core" value="2"/>
      <property name="Brand" type="Brand" value="Intel i5"/>
      <property name="Frequency" type="FrequencyInGHz" value="2.4"/>
      <property name="Architecture" type="Architecture" value="64-bit"/>
    </resource>
    <shared-resource name="OS" type="Software">
      <property name="Brand" type="Brand" value="Linux"/>
      <property name="Distribution" type="OSDistribution" value="Ubuntu"/>
      <property name="Kernel" type="Kernel" value="3.8.0-29-lowlatency"/>
      <property name="Architecture" type="Architecture" value="64-bit"/>
    </shared-resource>
  </capabilities>
  <machines>
    <machine name="Machine1" description="Our test machine">
      <capabilities>...</capabilities>
      <nodes>   <!-- node description detailed below -->  </nodes>
    </machine>
  </machines>
  <interconnections>
    <connection end1="Machine1" end2="Machine1">
      <protocol name="LocalJavaInvocation"/>
    </connections>
  </interconnections>
  <modes>
    <mode name="Mode1"> ...   </mode>
    <mode name="Mode2"> ... </mode>
  </modes>
  <comm-protocols>
    <protocol name="LocalJavaInvocation">
      <property name="MaxThroughputInBytesS" type="MaxThroughputBytesMS"
          value="99999999" />
      <property name="MaxDelayInMS" type="MaxDelayInMS" value="0" />
    </protocol>
  </comm-protocols>
  <mode-transitions>
    <transition from="Mode1" to="Mode2"/>
    <transition from="Mode2" to="Mode1"/>
  </mode-transitions>
</platform>
```

Since DCD_x is composed by two main tasks, we have divided the machine into two nodes, each one with one processor unit. Other quantifiable resources are shared equally. Listing 5.2 details nodes' description.

Listing 5.2: Test platform node description

```
<nodes>
  <node name="Node1">
    <capabilities>
      <shared-resource name="JVM" type="Software">
        <property name="Brand" type="Brand" value="JamaicaVM"/>
        <property name="Type" type="Type" value="Real-time"/>
      </shared-resource>
      <resource name="RAMMemory" type="Storage">
        <property name="Quantity" type="MemoryInMB" value="2048"/>
        <property name="Available" type="MemoryInMB" value="2048"/>
        <property name="Type" type="Type" value="DDR2"/>
      </resource>
      <resource name="ProcessorUnit" type="Computing">
        <property name="Quantity" type="Core" value="1"/>
        <property name="Available" type="Core" value="1"/>
        <property name="Brand" type="Brand" value="Intel i5"/>
        <property name="Frequency" type="FrequencyInGHz" value="?"/>
        <property name="Architecture" type="Architecture" value="64-bit"/>
      </resource>
      <shared-resource name="OS" type="Software">
        <property name="Brand" type="Brand" value="Linux"/>
        <property name="Distribution" type="OSDistribution" value="Ubuntu"/>
        <property name="Kernel" type="Kernel" value="?"/>
        <property name="Architecture" type="Architecture" value="64-bit"/>
      </shared-resource>
    </capabilities>
  </node>
  <node name="Node2">
    <capabilities>
      <shared-resource name="JVM" type="Software">
        <property name="Brand" type="Brand" value="JamaicaVM"/>
        <property name="Type" type="Type" value="Real-time"/>
      </shared-resource>
      <resource name="RAMMemory" type="Storage">
        <property name="Quantity" type="MemoryInMB" value="2048"/>
        <property name="Available" type="MemoryInMB" value="2048"/>
        <property name="Type" type="Type" value="DDR2"/>
      </resource>
      <resource name="ProcessorUnit" type="Computing">
        <property name="Quantity" type="Core" value="1"/>
        <property name="Available" type="Core" value="1"/>
        <property name="Brand" type="Brand" value="Intel i5"/>
        <property name="Frequency" type="FrequencyInGHz" value="?"/>
        <property name="Architecture" type="Architecture" value="64-bit"/>
      </resource>
      <shared-resource name="OS" type="Software">
        <property name="Brand" type="Brand" value="Linux"/>
        <property name="Distribution" type="OSDistribution" value="Ubuntu"/>
        <property name="Kernel" type="Kernel" value="?"/>
        <property name="Architecture" type="Architecture" value="64-bit"/>
      </shared-resource>
    </capabilities>
  </node>
</nodes>
```

Once the platform has been created and validated, applicative contributions can be deployed.

5.2.2 Service Compositions, Instances and Tasks: the Detector example

We will exemplify service composition, task and instance deployment by detailing DCD_x's detector contribution.
Listing 5.3 depicts the service composition descriptor of our task. As it can be noticed, the Java class implementing the detector has three dependencies: one towards a frame buffer service, one towards a collision set reducer service and last one towards a motionizer service. All dependencies are indicated as dynamic. A Runnable service is published, so that the implementation can be seen by the URSO Scheduler implementation. The description also contains the implementation requirement in terms of heap memory. Concerning the content of the URSO interface files:

- the `frameService.interface` file lists one operation, `getFrame`, with no inputs and a `Frame` object as output. A `Frame` object is constituted of a list of 2-uples ⟨`AircraftInfo`, `float[]`⟩, where `AirCraftInfo` is a structure holding information about an aircraft and the `float` array is a three dimensional position (an array with three `float` objects, equivalent to 12 bytes). In its turn, an `AircraftInfo` object keeps an `String` ID with at most 30 Unicode characters (each represented by two bytes, so 60 bytes in total), a trajectory function (which has a six `float` long array to represent a time-dependent function for each coordinate, totalling 72 bytes for function) and an `int` field informing the duration of its flight (four more bytes). Summing it all, an item inside a frame has 148 bytes. If we limit the number of aircrafts in a single radar frame to 30, we would receive 4440 bytes. Since we are using the local Java invocation communication protocol to communicate, the communication delay is assumed to be zero, but a more efficient data structure would need to be used for other communication protocols.

- the `motionizerService.interface` file also lists one operation, `createMotions`, which take a `Frame` as input and returns a list of `Motion` objects. `Motion` contains three `float` array objects indicating its old position, its actual position and the motion vector in a three dimensional plan (thus, three `float` array with three `floats` each = 36 bytes) and an `AircraftInfo` object (136 bytes). Consequently, a motion consumes 172 bytes. Again, if we limit the number of aircrafts to 30 aircrafts, we would need 5160 bytes to represent a `Motion` list.

- the `reducerService.interface` file lists one operation, `reduceCollisionSet`, whose input is a list of `Motion` and output is a list of lists of `Motion`.

After reducing the collision set, the detector verifies itself whether there are real collisions, by checking motions interceptions two by two.
Listing 5.3 also depicts the deployment plan of the task corresponding to the service composition. It is a periodic task, whose period and deadline are equal to 100ms (we are considering that the simulator produces 10 frames per second).
On the deployment of this contribution, the URSO framework will inspect the Detector class and generate a control flow graph and a WCET expression for it. Figure 5.2 shows the control flow graph for its main method, run, which is used to execute the composition. Nodes filled with grey were identified by the algorithm presented in the Assembly section as being in a loop, and their WCET is multiplied by a constant (*maxLoop*) which estimates the maximum number of iterations of the loop. In addition, nodes in black are nodes whose WCET can not be totally estimated due to their calls towards functions that may not be

present in the WCET Manager map[5].

Listing 5.3: Detector service composition and deployment plan

```
<composition name="detector">
 <implementation.urso class="com.bull.dcdj.detector.Detector.class">
   <requirements>
     <resource name="Memory" type="Storage">
       <property name="Available" type="MemoryInMB" value="15"/>
     </resource>
   </requirements>
 </implementation>
 <reference name="frameBuffer" multiplicity="1..1">
   <interface.urso class="com.bull.dcdj.api.FrameService.class" file="
       frameService.interface"/>
   <binding.dynamic/>
 </reference>
 <reference name="reducer" multiplicity="1..1">
   <interface.urso class="com.bull.dcdj.api.ReducerService.class" file="
       reducerService.interface"/>
   <binding.dynamic/>
 </reference>
   <reference name="motionizer" multiplicity="1..1">
   <interface.urso class="com.bull.dcdj.api.MotionizerService.class" file="
       motionizerService.interface"/>
   <binding.dynamic/>
 </reference>
 <service name="detector">
   <interface.urso class="java.lang.Runnable" file="runnable.interface"/>
   <binding.dynamic>
     <property name="name" value="detector"/>
   </binding.dynamic>
 </service>
 <semantics>
   invoke getFrameOp in frameBuffer.description.operations, null, frame;
   invoke createMotionsOp in motionizer.description.operations, frame,
       motions;
   invoke reduceMotionsOp in reducer.description.operations, motions,
       motions;
   invoke detectCollision, motions, collisions;
   invoke system.print, collisions, null;
 </semantics>
</composition>

<deployment>
  <tasks>
      <periodic-task name="Detector" composition="detector.composition" node
          ="Node1" period="100">
        <property name="Deadline" value="100" type="DeadlineInMS"/>
        <modes>
        <mode name="Mode1"/>
      </modes>
      </periodic-task>
    </tasks>
 </deployment>
```

After creating the graph, the WCET manager looks for all paths (without repeating edges) from the initial node until the final node. In Figure 5.2, the WCET Manager would find

[5]It is assumed that the WCET of JVM methods were previously calculated and inserted in the WCET Manager map.

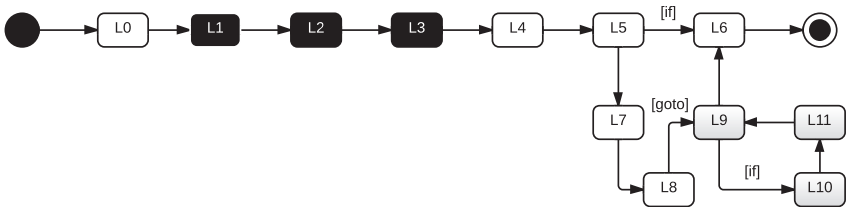

Figure 5.2: Control flow graph for the Detector Implementation

the paths: 1) L0 - L1 - L2 - L3 - L4 - L5 - L6, 2) L0 - L1 - L2 - L3 - L4 - L5 - L7 - L8 - L9 - L6, and 3) L0 - L1 - L2 - L3 - L4 - L5 - L7 - L8 - L9 - 10 - L11 - L6.
The next step is to reduce the number of paths by seeing which paths are included in other paths. The WCET Manager considers that a path p_1 is included in (or is a subpath of) a path p_2, if p_2 visits the same nodes as p_1 in the same order. In that case, p_1 can be removed from the paths list. In the given example of the Detector implementation, all paths can be included in path 3), which will be the only path considered for the WCET calculation.

Listing 5.4: WCET Estimation of Detector Implementation Nodes

```
WCET(L0) = WCET(ACONST_NULL) + WCET(ASTORE 2)
WCET(L1) = WCET(ALOAD 0) + WCET(GETFIELD frameBuffer) + WCET(INVOKEINTERFACE
     FrameService, getFrame) + WCET(ASTORE 1)
WCET(L2) = WCET(ALOAD 0) + WCET(ALOAD 0) + WCET(GETFIELD motionizer) + WCET(
    ALOAD 1) + WCET(INVOKEINTERFACE MotionizerService, createMotion) + WCET(
    ASTORE 3)
WCET(L3) = WCET(ALOAD 0) + WCET(ALOAD 0) + WCET(GETFIELD reducer) + WCET(
    ALOAD 0) + WCET(GETFIELD reducer) + WCET(ALOAD 0) + WCET(ALOAD 3) + WCET
    (INVOKEINTERFACE ReducerService, voxelMap) + WCET(INVOKEINTERFACE
    ReducerService, reduceCollisionSet)
WCET(L4) = WCET(ALOAD 0) + WCET(ALOAD 0) + WCET(ALOAD 2) + WCET(
    INVOKEVIRTUAL determineCollisions) + WCET(ASTORE 4)
WCET(L5) = WCET(ALOAD 4) + WCET(IFNULL L6) + WCET(ALOAD 4) + WCET(
    INVOKEINTERFACE List, size) + WCET(IFLE L6)
WCET(L6) = WCET(RETURN)
WCET(L7) = WCET(GETSTATIC System.out) + WCET(NEW StringBuilder) + WCET(DUP)
    + WCET(ALOAD 4) + WCET(INVOKEINTERFACE List, size) + WCET(INVOKESTATIC
    String, valueOf) + WCET(INVOKESPECIAL StringBuilder, init) + WCET(LDC) +
     WCET(INVOKEVIRTUAL StringBuilder, append) + WCET(INVOKEVIRTUAL
    StringBuilder, to String) + WCET(INVOKEVIRTUAL PrintStream, println)
WCET(L8) = WCET(ALOAD 4) + WCET(INVOKEINTERFACE List, iterator) + WCET(
    ASTORE 6) + WCET(GOTO L9)
WCET(L9) = maxLoop(WCET(ALOAD 6) + WCET(INVOKEINTERFACE Iterator, hasNext) +
     WCET(IFNE L10))
WCET(L10) = maxLoop(WCET(ALOAD 6) + WCET(INVOKEINTERFACE Iterator, next) +
    WCET(CHECKCAST Collision) + WCET(ASTORE 5))
WCET(L11) = maxLoop(WCET(GETSTATIC System.out) + WCET(ALOAD 5) + WCET(
    INVOKEVIRTUAL Collision, toString) + WCET(INVOKEVIRTUAL PrintStream,
    println))
```

Next, the WCET Manager generates the WCET expression for the method. In order to do so, it takes each node from the graph path and calculates its WCET as the sum of the WCET of its instructions, as shown in the Listing 5.4[6]. If the instruction is not an invocation instruction, its execution time can be found in a Instruction table kept by the

[6]For sake of clarity, WCET(x) denotes the function which returns the WCET of an instruction x.

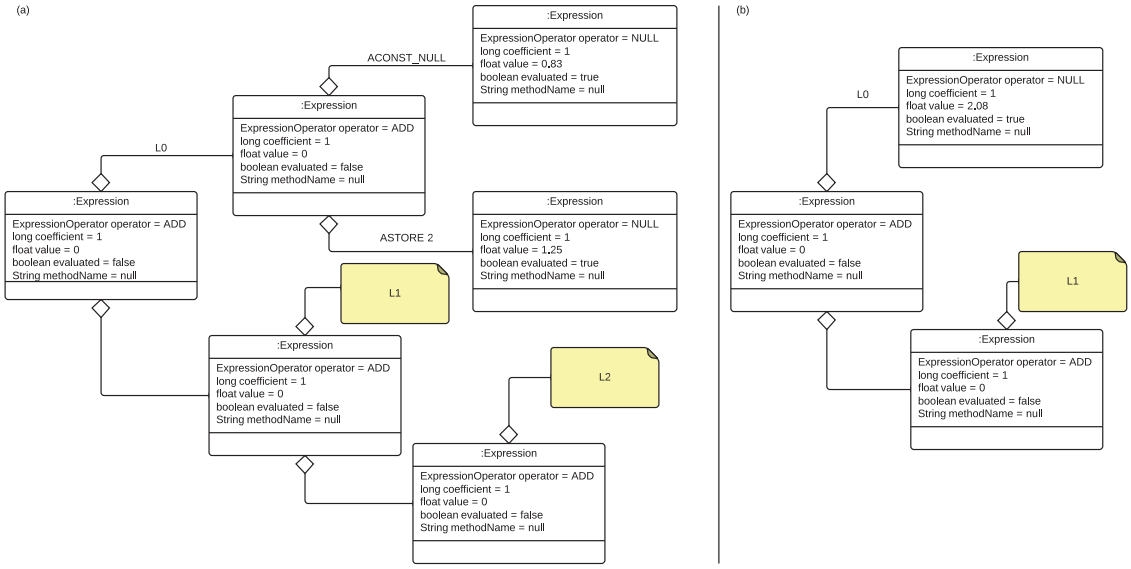

Figure 5.3: WCET Expression for Node L0

framework. Otherwise, the framework queries for the WCET of the invocation in its WCET Manager table. Figure 5.3 *(a)* exemplifies the WCET expression object representation detailing L0 node WCET. As depicted in the Figure in *(b)*, when two children nodes (left and right sides) of an expression are evaluated (that means, both have an associated cost value), their parent expression can be evaluated as well.
Figure 5.4 *(a)* shows the WCET expression for the L1 node. L1's instruction set contains a service invocation (characterized by a INVOKEINTERFACE instruction), so it can not be evaluated until a service implementation is found to satisfy its dependency. Every time a new item is added to the total WCET table, the Manager updates all items of the partial WCET table. Although L1's WCET can not be evaluated, it can be simplified as shown in part *(b)*.
After analysing the class' byte-code, the framework generates an OSGi bundle from the SCA contribution. After the deployment of this bundle, the iPOJO adapter module creates an iPOJO composite wrapping the service composition and exposing its dependencies and Runnable service. Then, the task object is transmitted to the URSO Scheduler, which retrieves the task Runnable service and, based on the task type and properties, creates a real-time thread for it. Before starting the task, the run-time verifies whether the complete set of tasks can be scheduled without violating their deadlines through an upper bound response time [Bini *et al.* 2009] test. In the execution of the schedulability test, it is assumed that all service composition's dependencies have been satisfied and its WCET has been evaluated. Otherwise, it is not possible to establish the task's cost.

5.2.3 DCD_j, a Java-based implementation of DCD_x

The DCD_j implementation is composed by seven SCA contributions which are detailed in the next paragraphs.
As detailed in the last section, the **detector** contribution is formed by a service composition implementation class, a service composition description (with an implementation descrip-

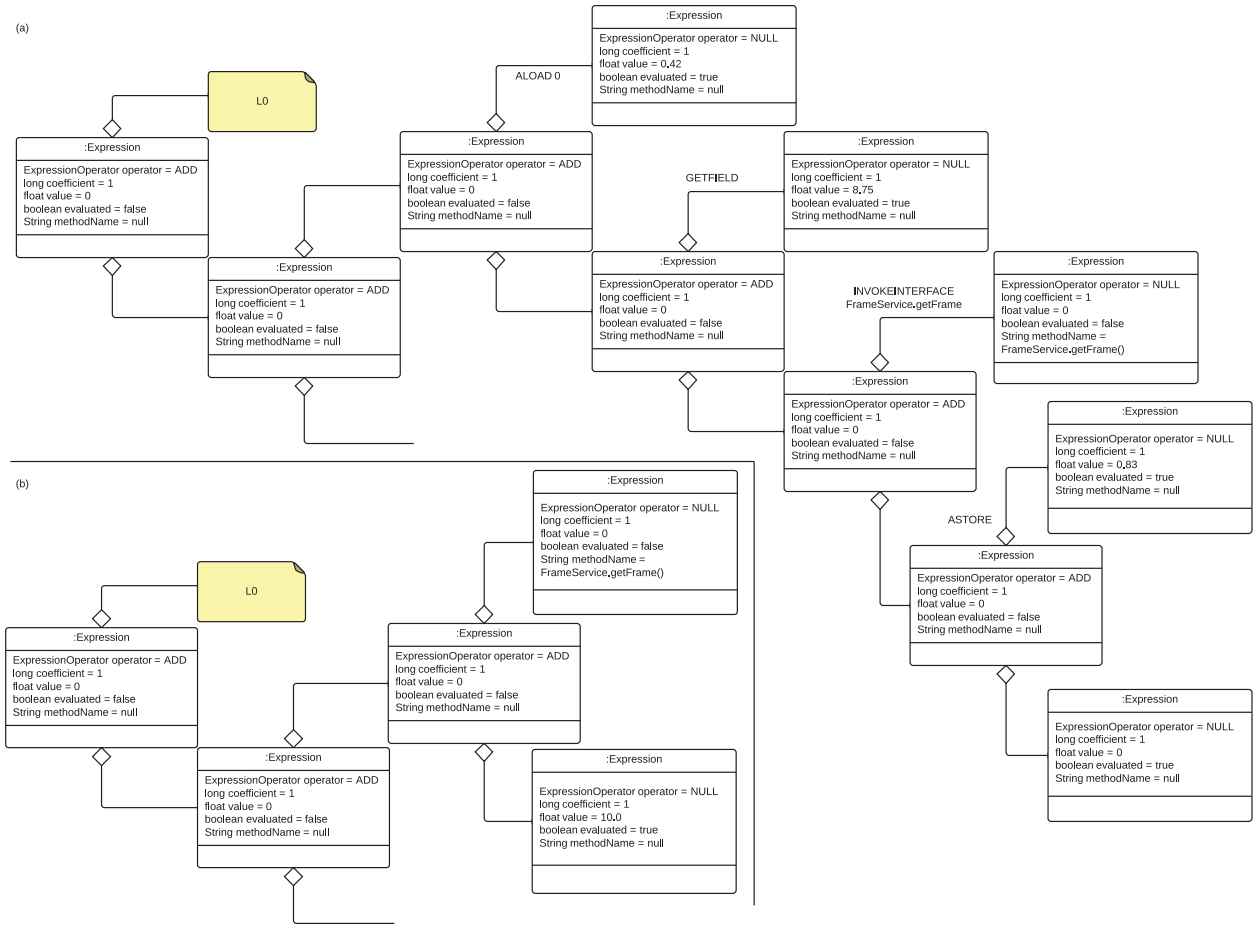

Figure 5.4: WCET Expression for Node L1

tion), a deployment plan defining a periodic task for collision detection and three interface descriptions corresponding to the interface of the service dependencies it has. Its task and service composition are responsible for retrieving a radar frame containing a set of aircrafts, analysing aircrafts' trajectory and detecting potential collision. To this end, first the detection module reduces the set of possible collisions by analysing aircrafts in a bi-dimensional plan. Potential collisions are then analysed in a three-dimensional plan.

The **frame buffer** contribution contains the FrameBuffer, an entity which manages radar frames. Frames are kept in an array blocking queue structure. Frame Buffer provides a service called Frame Service, which possesses operations to add new frames (which receives a Frame object as input parameter and has no return value) and to get the next frame in the queue (with has no input parameters and returns a Frame object). Both operations are described in its service interface. The contribution also contains the FrameBuffer URSO-conform SCA composite file and a deployment plan mapping one FrameBuffer instance to the node named 'Node1' in the platform.

The **motionizer** contribution contains the Motionizer, which is an entity that produces motions from frames. Motions are structures which stores an aircraft information and the current and last positions of such aircraft. In order to do so, the motionizer stores himself

a map structure associating an aircraft identifier with the last motion produced for such aircraft. The motionizer provides a service named Motionizer Service. Its only operation is createMotions, whose input is a Frame object and output is a list of Motion objects. Besides the service description, this contribution also contains a composite file and a deployment plan mapping one Motionizer instance to 'Node1'.

The **reducer** contribution contains the Reducer, an entity that reduces the set of potential collisions by analysing aircrafts motions and eliminating those which are too far to be able to collide. This is performed by an operation called reduceCollisionSet, present on the interface of the Reducer Service, whose input is a list of Motion objects and output is a list of list of motions (grouping in inner lists motions which are likely to collide). Motion set reduction is based on bi-dimensional voxel[7] colouring algorithms. Reducer contribution also contains a composite file and deployment plan mapping one instance of it to 'Node1'.

The **simulator** contribution contains the simulator, which is a task and a service composition that creates radar frames containing the registered aircrafts. Simulation tasks are sporadic non real-time threads. Their service composition has a mandatory/simple dependency on the Frame service and an optional/multiple dependency on the Aircraft service. From the Frame service, the simulator uses the operation storeFrame, which adds a frame to tail of the Frame buffer queue. Upon the registration of a new implementation of the aircraft service, the simulator adds the implementation object in a list and evaluates its position based on the trajectory function provided by the aircraft. An aircraft has a maximum flight time, after which it is removed from the simulator's aircrafts list. Besides the service composition implementation class, the composition also contains a deployment plan mapping the sporadic non-real-time simulator task to 'Node2', a service composition description and service descriptors for its dependency/reference interfaces. Just as the detector task, the implementation provides a Runnable service that is used to instantiate its executing thread.

The **aircraft creator** contribution is a special contribution responsible for the dynamism of the DCD_j benchmark application. It defines a periodic non-real-time task that dynamically creates new Aircraft instances with different configurations. It is important to notice that some aircrafts may not be instantiated due to lack of resources in the platform. These component instances are stored in a component instance pool; once the pool is empty, the oldest aircraft instance is removed (which means that its reserved resources are released) to be replaced by a new one. It does not provide or require any service; the aircraft instances that it creates are responsible for providing the Aircraft Service used by the simulator. The Aircraft service interface provided by these instances contains one service operation, getInfo, which returns an AircraftInfo object containing the aircraft identifier, trajectory and maximum flight time. The contribution contains the service composition implementation class, a deployment plan mapping its periodic threads to 'Node2' and descriptors for its service composition.

Table 5.2 shows metrics for each contribution mentioned above. The 'Descriptors' column indicates the number of URSO descriptors (interface, composite, service composition or deployment plan) for each contribution. The 'Complexity' column indicates average McCabe's cyclomatic complexity for the set of classes in each contribution. The 'Aircraft' contribution corresponds to the aircraft composites dynamically created by the Aircraft creator contribution. The 'Common' contribution is actually an OSGi bundle which contains common classes used by all contributions. It also contains all Java service interface classes. Its only descriptor is the OSGi Manifest file.

[7] Voxels (contraction of volumetric elements) are n-dimensional (usually three-dimensional) pixels. In the context of DCD_x they designate bounded areas in a bi-dimensional plan. In order to create voxels from the three-dimensional space, we ignore the z dimension.

Contribution	Methods	Classes	Lines of Code	Descriptors	Complexity
Aircraft	5	1	36	3	1.4
AircraftCreator	1	1	31	2	3
Common	30	6	247	1	1.76
Detector	2	1	42	5	2.5
FrameBuffer	3	1	20	3	1.33
Motionizer	1	1	24	3	1
Reducer	7	1	135	3	2.28
Simulator	5	1	66	4	1.6
Total	**54**	**13**	**601**	**24**	**-**

Table 5.2: Contributions' project metrics

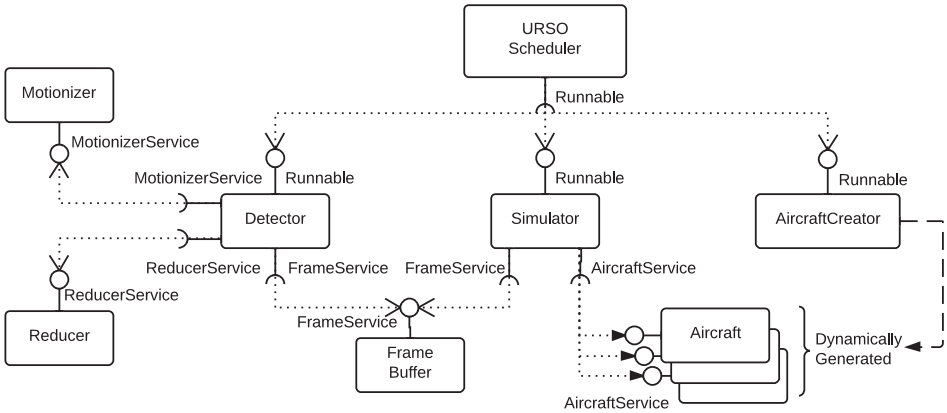

Figure 5.5: DCDj Service Architecture on NaSCAr

The resulting overall architecture is depicted in Figure 5.5.

5.3 Validation

5.3.1 Methodology overview

Based on the application architecture presented in the previous section, validation was carried out adopting the following methodology:

1. First, the command `sca:platformdesc` was executed with the platform description file presented in the previous section (analysis of platform validation time);

2. Then, the command `sca:platforminfo` was executed to check whether the stored information matches with the data contained in the platform description. This command was executed several times during the validation phase to certify that the instances and tasks were assigned to the right nodes and to check the impact of the current number of active entities in the platform on the command execution time;

3. Next, contributions that only provide services were deployed through the command `sca:deploy`. For each contribution, it was measured parsing, byte-code instrumentation, WCET analysis, iPOJO instantiation, and complete deployment time;

4. Then, contributions containing tasks were deployed. Just as for the contributions in the previous step, we measured separately each sub-activity of its deployment and analysed its impact in the total value. Additionally, it was be measured their impact on the `sca:changemode` command, the exactitude of its WCET estimation and the deadline miss ratio of real-time threads;

5. Finally, contributions undeployment was tested by means of the command `sca:undeploy`.

Consistency checks were performed throughout these steps by means of the `sca:list` command.

5.3.2 Platform description validation analysis

In order to analyse the platform description parsing and validation command `sca:platformdesc`, platform descriptions of different sizes were used. The test was performed for a minimum platform description (22 objects), platform descriptions with 68, 162 and 268 objects, and the platform description depicted in the previous section, which contains 485 objects.
The chart in Figure 5.6 shows the parsing and platform creation methods execution time variation in seconds according to the number of elements declared in the XML file. Platform creation process includes parsing the platform description, and performing the validation on the object representation. It was executed in a JamaicaVM 6.2-4 virtual machine on a Ubuntu Linux 13.04 with a low-latency kernel version *3.8.0-27-lowlatency*. Underlying processor was an Intel® Core ™ i5 CPU with two cores (with two threads each) running at 2.40GHz. Each test takes into account the minimum execution time from ten benchmark rounds[8], after one warm-up execution round. As it can be seen, the platform validation time (difference between the platform creation - red line - and parsing - blue line) increases at a higher rate than the parsing time as the number of object increases. Table 5.3 contains the resulting values for platform parsing and creation. Values are expressed in seconds.

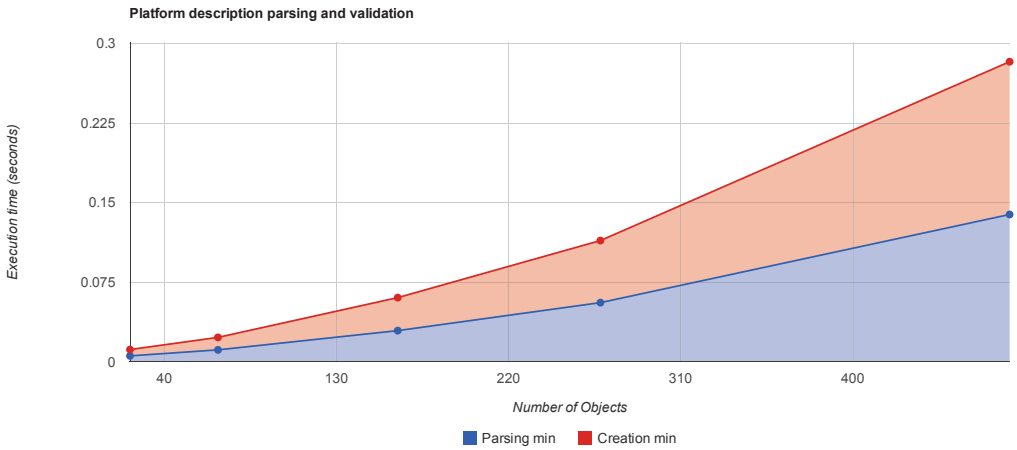

Figure 5.6: SCA:PlatformDesc command: XML parsing and platform creation

[8]The minimum execution time was picked because it represents the execution time of the method with least external (*e.g.* context switch) interference. The model creation and parsing occur during the initialization of the platform, and do not require high performance or predictability.

	22 objects	68 objects	162 objects	268 objects	485 objects
Parsing minimum	0.005429	0.011102	0.029219	0.055592	0.138624
Parsing maximum	0.011402	0.017945	0.038836	0.066363	0.158608
Parsing average	0.008506	0.013840	0.033137	0.059640	0.144773
Parsing stdev	0.001827	0.002465	0.003374	0.003366	0.005385
Creation minimum	0.005968	0.011739	0.031101	0.058480	0.143937
Creation maximum	0.011833	0.018624	0.040860	0.069263	0.162211
Creation average	0.009017	0.014596	0.035062	0.062527	0.149921
Creation stdev	0.001787	0.002555	0.003410	0.003380	0.004957

Table 5.3: Platform description parsing and validation results

5.3.3 Platform information analysis

Similarly to platform description parsing and creation, in order to analyse platform information retrieval, we have tested the command against platform descriptions with different complexity levels (number of objects). The same platform descriptions employed in the previous test were used in this test.
The chart in Figure 5.7 shows a execution time comparison of the `sca:platforminfo` command for different platform configurations. As previously, the chart presents the obtained minimum execution times per configuration. Again, we have used a minimum platform description, configurations with 68, 162 and 268 objects, and the use-case platform description described in the previous section. Implementations uses Java's `StringBuilder`'s method `append` to assemble the information from different levels of the platform. As expected, execution time grows linearly with the number of objects. Table 5.4 summarizes the results for a suite with ten benchmark rounds and one warm-up round.

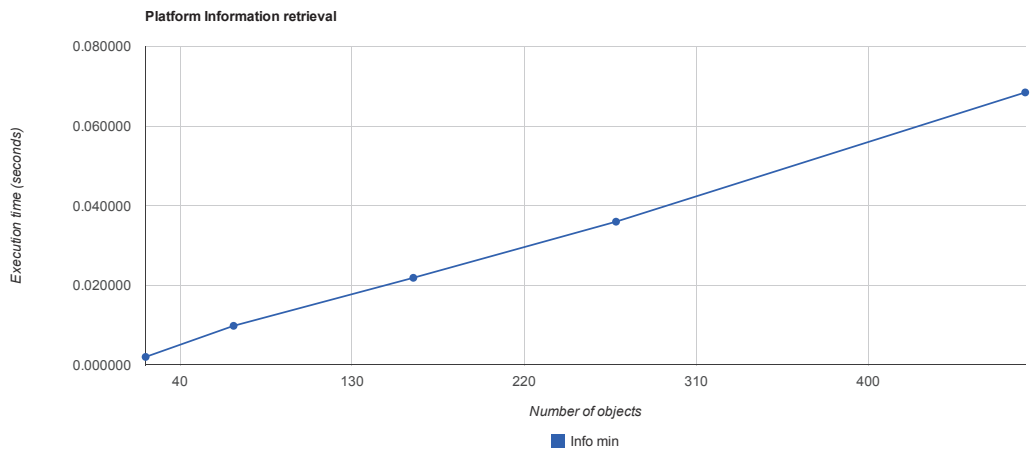

Figure 5.7: SCA:PlatformInfo command: Information retrieval and string composition

5.3.4 Contribution deployment analysis

In order to understand contribution deployment analysis, it is important to understand the different sub-activities that are performed to deploy a contribution. Two main activities can be identified. For contributions containing passive (composite instances) or active (tasks)

	22 objects	68 objects	162 objects	268 objects	485 objects
Info minimum	0.002011	0.009821	0.021884	0.035982	0.068453
Info maximum	0.006942	0.014996	0.028616	0.056437	0.080609
Info average	0.005497	0.012718	0.025207	0.042899	0.073897
Info stdev	0.001756	0.001956	0.002414	0.007152	0.004146

Table 5.4: Platform description information display results

entities, the first activity is the same:

1. *Contribution processing*: Contribution processing consists in inspecting the input contribution file and transforming it so that the file and its internal content are conform with the platform. This activity is composed of three sub-activities:
   (a) *Textual file parsing*: In this activity, descriptors are parsed and object representations of them are created. Deployment plans and interfaces are descriptors common to both passive and active entities. Passive entities composites are declared in composite files, whereas active entities are associated to service composition descriptors;
   (b) *Implementation class instrumentation and analysis*: This sub-activity is responsible for modifying class files byte-code in case of service-level dependencies and inspecting classes' methods to establish WCET estimations;
   (c) *OSGi bundle creation*: Bundle creation activity actually includes class instrumentation and file parsing; while parsed textual files are not included in the final bundle, modified classes are added to it. In addition, an OSGi manifest file is created for the contribution, importing its provided and required service interface class packages and exporting the component implementation classes so that the iPOJO framework can instantiate them in components and composites.

The second activity is particular to the nature of the entities in the deployed contribution. The case of passive and active entities is developed separately in the next sections. The deployment of hybrid contributions, containing both active and passive entities, is performed by executing the activity set union.

5.3.4.1 Passive entities' deployment

For passive entities deployment, the second activity is the following:

2. *Composite instance deployment*: After creating the OSGi file with the necessary classes and descriptors, instance creation and deployment may take place. This activity is composed by three sub-activities:
   (a) *OSGi bundle installation and start*: As the name suggests, in this activity the generated OSGi bundle is installed and started;
   (b) *iPOJO-based adaptation*: In this activity, the framework uses iPOJO's Composite API to create component and composite instances for the URSO composites declared in the composite file;
   (c) *Node instance deployment*: This activity consists in checking whether the desired node has enough resources to deploy the composite instance (*i.e.* it must have enough resources to attend the resource requirements of all composite inner components implementations) and then deploys the instance in the OSGi platform, storing its reference in case of undeployment.

	Reducer	Motionizer	Aircraft	FrameBuffer
Act. 1-a	0.002889	0.002757	0.003242	0.004564
σ	0.000636	0.000107	0.000745	0.001550
Act. 1-b	0.217476	0.016426	0.218405	0.013179
σ	0.010482	0.001843	0.002995	0.003874
Act. 1-c	0.232013	0.032631	0.240410	0.031448
σ	0.009416	0.003757	0.002270	0.005911
Act. 2-a	0.019629	0.027616	0.023181	0.028609
σ	0.003876	0.002218	0.006109	0.002785
Act. 2-b	0.096553	0.050689	0.072306	0.069643
σ	0.002962	0.002955	0.003819	0.004399
Act. 2-c	0.141063	0.128975	0.124125	0.144452
σ	0.012160	0.003787	0.003525	0.002281

Table 5.5: Contribution deployment activity results for passive entities

Figure 5.8 depicts the average execution time for each deployment activity in passive entities' deployment. As it can be seen, activity 1-b (implementation class instrumentation and analysis) execution time impacts heavily on the deployment of contributions whose component instance implementation contains several methods. Other activities affected uniformly in all tested contributions. After instrumentation on complex implementations, activity 2-c (node instance deployment) is the most time-consuming in all contributions, as it consists first, on checking whether the deployment node has enough resources for all components implementation requirements, second, on performing resource reservation, and third, on deploying the iPOJO instance (and performing all necessary bindings) on the platform.

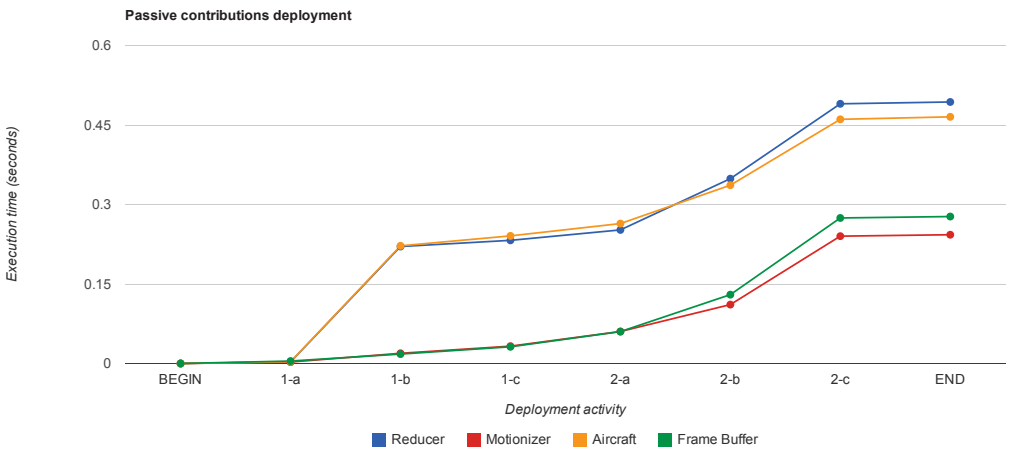

Figure 5.8: SCA:Deploy command: Contributions with passive entities

Table 5.5 summarizes the average execution time for each activity. Values are measured in seconds and were obtained by executing the `sca:deploy` command five times after one warm-up execution. The second line in each cell represents standard deviation.

5.3.4.2 Active entities' deployment

For active entities deployment, the second activity is the following:

2. *Task deployment*: As for passive entities, after creating the OSGi file with the necessary classes and descriptors, service composite instance creation and task deployment may take place. This activity is formed by two sub-activities:

   (a) *Service composition composite deployment*: In fact, this activity comprises the activity 2 for passive entities. For each service composition, the framework creates iPOJO composite instances that are deployed in the node indicated for the task deployment if it has enough resources;

   (b) *Task scheduling*: In this activity, the scheduler queries runnable services exposed by the service compositions and instantiates real-time threads with them, whose scheduling and release parameters are based on the properties listed in the task description;

Figure 5.9 presents the average time for each deployment activity during active entities deployment. Activities 2-a-a, 2-a-b and 2-a-c correspond to the three activities that form activity 2 in passive entities. Since active entities implementation are basically constituted of a class with a single method `run()`, their instrumentation and analysis time is shorter than for passive entities. On the other hand, the impact of node deployment for active entities is more significant, due to the fact that the deployment of their instances require using service selection mechanisms to satisfy their dependencies.

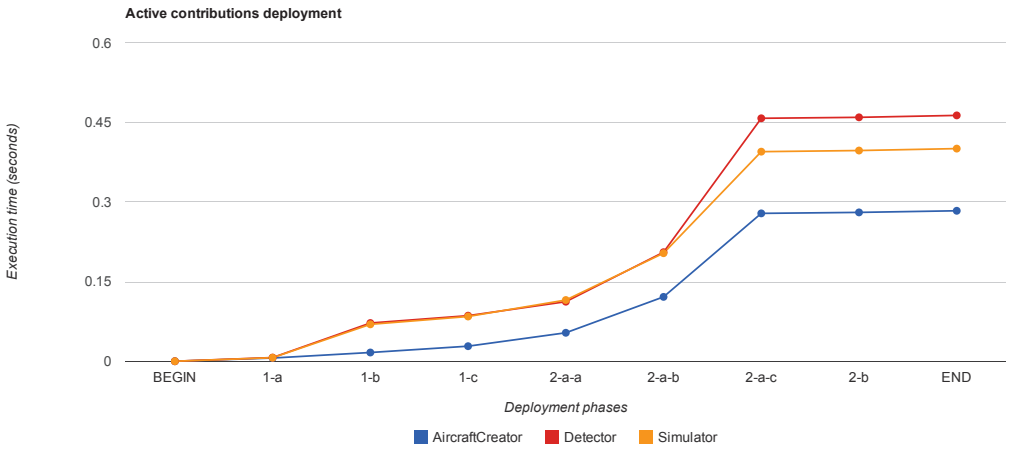

Figure 5.9: SCA:Deploy command: Contributions with active entities

Table 5.6 displays average execution times for active entities' deployment activities. Again, values are measured in seconds and were obtained by executing the `sca:deploy` command five times after one warm-up execution. Bundles providing services required by activity entities were already deployed in the platform.

5.3.5 Contribution undeployment analysis

Differently from its deployment counterpart, the `sca:undeploy` command works similarly for all contributions. The user must inform the id number of the contribution to be undeployed (it can obtained by means of the `sca:list` command). Three arrays are stored by the NaSCAr implementation:

	AircraftCreator	Detector	Simulator
Act. 1-a	0.006023	0.006888	0.006626
σ	0.002233	0.000294	0.001564
Act. 1-b	0.010413	0.065087	0.062870
σ	0.001668	0.005141	0.003535
Act. 1-c	0.028332	0.085827	0.084246
σ	0.003801	0.006567	0.005166
Act. 2-a-a	0.025258	0.026324	0.031065
σ	0.003307	0.002712	0.007273
Act. 2-a-b	0.067639	0.093269	0.088212
σ	0.011437	0.007091	0.009684
Act. 2-a-c	0.157339	0.252298	0.191312
σ	0.006406	0.026257	0.004649
Act. 2-b	0.001844	0.001798	0.002123
σ	0.000046	0.000434	0.000578

Table 5.6: Contribution deployment activity results for passive entities

1. An array containing contribution file paths. The array index is used as identifier for each deployed contribution;

2. An array containing objects of a class called `InstanceInformation`. This class contains an integer field indicating the ID of the contribution in which a given composition instance was declared. It also contains an iPOJO `ComponentInstance` object.

3. And an array containing objects of a class called `TaskInformation`. This class also contains a field that indicates the ID of the contribution in which a given task was declared. In addition, it stores the name of this task.

So when the `undeploy` command is executed, first, NaSCAr iterates over the `InstanceInformation` array and, for each item whose ID field value matches the ID passed as argument to the command, it disposes the iPOJO instance, releases the resource required by the associated implementation and removes the resources provided by them. Then, it iterates over the `TaskInformation` array and, for each object whose ID field value matches the command argument ID, it invokes the URSO Scheduler method `undeployTask` on the task name. In its turn, the scheduler implementation must retrieve the thread associated to that name, and interrupt it. Consequently, the time needed to undeploy a contribution depends on its number of declared component instances and tasks and on the number of components and tasks declared by a composite.
In order to analyse the undeployment performance, we have deployed and undeployed a same contribution several time in the platform. Table 5.7 presents the average task and instance undeployment execution times for DCD_j contributions and their standard deviation. Time was measured in seconds. Values were obtained by executing the `sca:undeploy` command five times after one warm-up execution.

5.3.6 Mode change analysis

In order to analyse the behaviour of the framework during mode transitions, it was measured the execution time of the `sca:changemode` command. Both tasks and instances may be affected during mode transitions: if a node must be disabled for a given mode but it was enabled in the previous mode, all its component instances as tasks must be undeployed.

	Instance undeployment	Task undeployment
Motionizer	0.094194	0.000009
σ	0.003087	0.000001
Reducer	0.098222	0.000009
σ	0.008644	0.000001
Aircraft	0.099668	0.000009
σ	0.005379	0.000001
FrameBuffer	0.095982	0.000009
σ	0.004910	0.000001
Detector	0.157883	0.000034
σ	0.006377	0.000004
AircraftCreator	0.138181	0.000038
σ	0.013795	0.000001
Simulator	0.145322	0.000035
σ	0.007571	0.000010

Table 5.7: Contribution deployment activity results for active entities

Similarly, if a node was not activated during a previous mode and is enabled in a new one, all declared tasks and instances mapped to this node must be deployed. Thus, it is expected that the mode change execution time is equal to the sum of the undeployment of the entities whose nodes must be disabled (instances and tasks) or which are not associated to the new mode (tasks) plus the deployment of entities whose nodes become enabled (instances and tasks) or which are associated to the new mode and were not associated to the previous one (tasks).
Three modes were defined in order to evaluate the `sca:changemode` command: a mode 'Mode1' in which all nodes and tasks are mapped, a mode 'Mode2' in which no tasks are declared, and a mode 'Mode3' in which only one of the two machines nodes ('Node1') is enabled. Bidirectional transitions between all three modes were declared. It is supposed that the association between a thread and a mode respects its node assignment, that is, that the node to which the task was mapped will be activated during its associated modes. Table 5.8 depicts the execution time and standard deviation obtained for all possible transitions. As mentioned before, in 'Mode1' there are seven composite instances and three tasks, in 'Mode2' there are seven component instances and in 'Mode3' there are four composite instances. Table rows indicate the transition origin mode, while columns indicate the transition destination mode. Table diagonal is not null due to the verifications performed by the platform to check whether the designated node exists and if it matches the name of the current mode. Values are slightly higher than those of expected by summing deployment and undeployment times, due to verifications and iterations over all deployed tasks and components to check whether they must remain in their actual state (either deployed or undeployed), be deployed or be undeployed. Based on the previous tables for deployment, undeployment and mode change, it is possible to state that:

- Both instance deployment and undeployment times are higher than task deployment and undeployment times in the framework;

- For both tasks and instances, deployment time tends to be higher than undeployment time.

	Mode1	Mode2	Mode3
Mode1	0.000045	0.001209	0.378603
σ	0.000010	0.000175	0.018017
Mode2	0.005663	0.000057	0.290199
σ	0.001351	0.000014	0.009670
Mode3	0.738378	0.680683	0.000066
σ	0.019279	0.055586	0.000013

Table 5.8: Mode change execution time

5.3.7 Execution timeliness analysis

Timeliness analysis has focused on two items:

- Comparison between expected services WCETs[9] and obtained worst execution time for services and service compositions.
- and real-time threads' timeliness and deadline miss ratio.

Figure 5.10 depicts the comparison between estimated WCETs and worst execution times obtained in a test suite with ten benchmark rounds for services provided by passive entities. No warm up round was provided, as worst case execution times may happen (and often do in Java, due to dynamic class loading costs) during this phase. The default value established for non-estimated methods has proven to be too pessimistic for most service operations. This test (and the tests hereafter) assumed that the maximum number of executions for each loop was 50, and considered recursive calls as methods with non-estimated WCETs (and consequently using the default value for calculations). We have limited the number of aircrafts to 50 as well (since most loops iterate over the list of aircrafts). For each service operation, the third bar (from left to right) represents the real worst time obtained (100%). The bar in the middle compares the estimated WCET with the worst time obtained in percentage terms. And the first of the three bars (from left to right) corresponds to the percentage of the estimated WCET formed by the addition of default values.

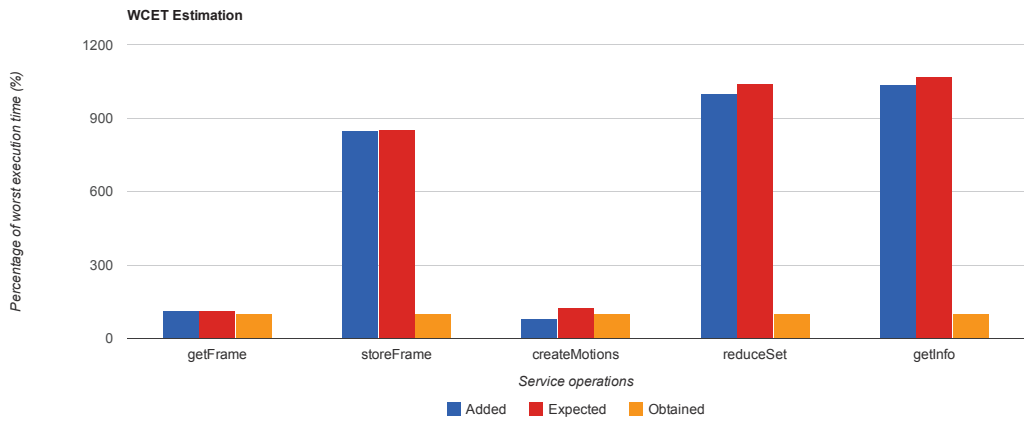

Figure 5.10: NaSCAr WCET estimation - passive entities services

[9]Since a non-open-source JVM was used, it was not possible to have access to all its byte-code beforehand to populate the WCET table. Consequently, for the timeliness analysis, we have established a default value for calls towards methods whose WCET could not be estimated: 100 micro-seconds.

	Highest execution time	Estimated WCET
FrameService.getFrame	452184	504878.75 (+11.6%)
FrameService.storeFrame	118196	1008906.25 (+753.5%)
MotionizerService.createMotion	878904	1091863.33 (+24%)
ReducerService.reduce	3197680	33198598.75 (+938.2%)
AircraftService.getInfo	57962	618297.917 (+966.7%)
AircraftCreator composition	4483979	14008385.417 (+212.4%)
Simulator composition.getInfo	89071491	139844182.913 (+57%)
Detector composition	15008467	33217411.25 (+121.3%)

Table 5.9: Estimated WCETs for services and service compositions implementations

Based on passive entities' services WCET, service compositions' WCET was calculated. Figure 5.11 presents the comparison between their estimated WCET and the highest obtained execution time among tasks releases. NaSCAr's WCET estimation was tighter for service compositions than for passive services. That is mainly due to the default value for unknown service calls and the difference in the nature of methods executed by them: basic services tend to make calls towards simple classes and initializer methods, for which the established value is too high compared to their execution time, whereas service compositions tend to use more complex services, whose execution time is closer to the default value.

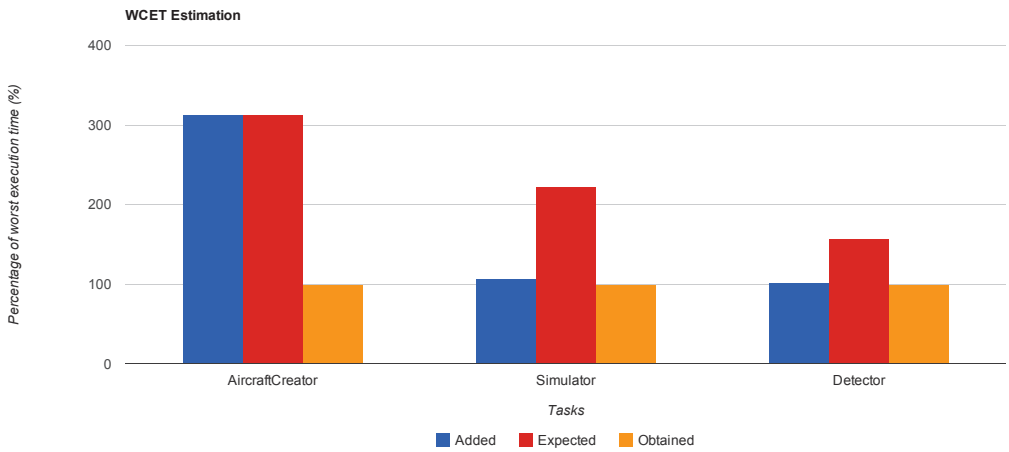

Figure 5.11: NaSCAr WCET estimation - service compositions

Table 5.9 presents the absolute values for both passive entities' services and service compositions WCET and worst execution time. Time is presented in nano-seconds. The percentage value beside the estimated WCET represents the overestimation rate.
As stated before, only one real-time thread is present at DCD_j: the detector thread. Based on the service compositions' estimated WCET, and on their properties, we have analysed its deadline miss ratio by varying its period (and consequently the deadline, since both have the same value in our application) of the detector task. The period of the simulator task was modified when necessary so that it remains always equal or higher than detector's period (otherwise, since our frame buffer implementation has a limited size, radar frames would be dropped). It was not necessary to perform schedulability test: since we have only one task with deadlines, the task set is schedulable as long as the real-time task WCET is higher than its deadline. Consequently, for periods higher than 140 ms the task set would

be considered as schedulable (although other tasks might not have the opportunity to be scheduled or executed until completion if the detector task has a higher priority). We have thus disabled the schedulability test in order to verify deadline's miss ratio for period values smaller than detector's WCET. The chart in Figure 5.12 depicts detector's thread deadline miss ratio for different period values (from 30 ms to 150 ms). In this interval, the miss ratio did not exceed 2.05%.

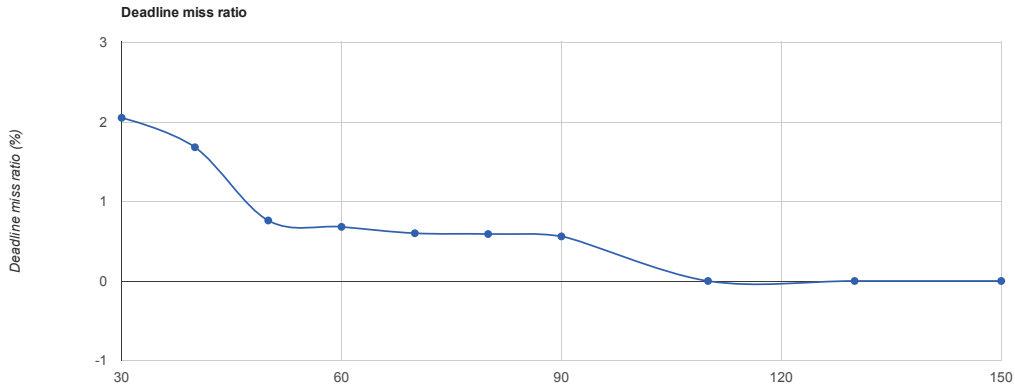

Figure 5.12: Detector task deadline miss ratio

5.4 Summary and Discussion

5.4.1 Overview

This chapter has presented NaSCAr, a Java framework based on a URSO-conform extended version of SCA. The framework was implemented in an OSGi platform and mapped URSO/SCA components to Apache Felix iPOJO component instances. URSO tasks were mapped to real-time threads, whose parameters were obtained through tasks' properties. By means of shell commands, users are able to deploy and undeploy SCA contributions, gather information about the actual state of the platform, its instances and tasks, and change the platform current operational mode.

This chapter has also presented an implementation of the Dynamic Collision Detection for Java, called DCD_j. DCD_j was separated into seven SCA contributions, four providing services through composite instances (Frame buffer, aircraft motionizer, collision set reducer, and aircrafts), and three defining service compositions and tasks (collision detection, traffic simulation and aircraft dynamic generation).

The time taken to deploy or undeploy a contribution in NaSCAr depends on the complexity of the implementation class. Classes with more dependencies and services take more time to be deployed due to instrumentation and validation phases. Similarly, complex composite instances take more time to be disposed by iPOJO. Platform mode transition time depends on the number of components and tasks that must be installed or uninstalled.

Concerning timeliness, NaSCAr's WCET estimation mechanism is very pessimistic. It assigns a high execution time value for the execution of methods whose WCET could not be estimated (for instance, JVM and native methods). Thus, it is important to populate NaSCAr's WCET tables conveniently with all used methods before executing applicative components, or many task sets may be considered as non-schedulable by its schedulability

test. In addition, DCD_j's detector thread was able to execute in NaSCAr with low levels of missed deadlines.

5.4.2 Discussion

NaSCAr has demonstrated to maintain good levels of predictability, despite being executed in a non-real-time OSGi framework. Better results could have been achieved by using a RTSJ-based OSGi implementation as the ones described in [Basanta-Val *et al.* 2013] and [Richardson & Wellings 2012]. NaSCAr could also benefit from some real-time JVM features on its implementation, such as native or ahead-of-time compilation. Tests on other virtual machines must be performed to verify if more predictable results can be attained.
A desirable feature for NaSCAr would be to enable users to interact with it by means other than shell commands. JMX (Java Management Extensions) may provide a way to monitor and interact with NaSCAr. It would require very little effort to implement it, as it could communicate through the same API that NaSCAr-shell module uses and iPOJO already contains a JMX handler to publish NaSCAr methods and field in MBeans. These fields and methods could then be observed and controlled through graphic consoles, such as VisualVM[10] or JConsole [11].
The manipulation of URSO-based structures is heavily based on the iteration of elements of potentially big data structures. A way to improve its performance could be to use more refined structures allowing the easy access to elements, such as hashes or caches.
As discussed in [Américo & Donsez 2012], despite the fact that iPOJO implements many of the concepts and mechanisms described in URSO and required for a service-oriented component model (such as hierarchical service registries, dynamic discovery and publication, service filtering), it contains one severe limitation: only one service may be provided by component (which may, in turn, provide several interfaces). Although this limitation did not impacted on the design of the DCD_j implementation, it may restrict the implementation of other real world applications.
A desirable feature for an OSGi-based implementation of URSO would be to integrate an implementation OSGi Remote Services specification in order to enable the design of real-time service-based distributed systems. Among the existing Remote Services implementation, there are Apache CXF [12] and R-OSGi [Rellermeyer *et al.* 2007].
An alternative to the use of a unique default value for non-estimated methods WCET would be to add some intelligence about the nature of the called methods and use diverse default values based on sampling. For instance, JVM constructors do not tend to have long execution times. Similarly, getters and setters often have a simple and short implementation. On the other hand, recursive call or a call towards methods with long lists of arguments tend to be longer. In addition, considering caches, pipelines and other computer architecture optimisation elements may highly improve WCET predictions.
Another alternative would be to associate a Java agent to the JVM in order to inspect every class during its loading phase and populate the WCET tables with the analysis result. If a warm-up phase is foreseen, the cost of this approach would be amortized, since used classes would be loaded and analysed by the agent during that phase and not during the real application execution. That would require a lot of space in memory for the WCET

[10] VisualVM is a graphic tool through which Java platforms can be monitored and controlled. More information about VisualVM can be found in `http://visualvm.java.net`.

[11] JConsole is a performance and resource monitoring tool for Java introduced with the JMX specification. More information about the JConsole can be found in `http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html`.

[12] More information about Apache CXF may be found at `http://cxf.apache.org/distributed-osgi.html`.

table though, in order to be able to store all JVM classes' WCET.
A `sca:update <contribution>` command could be added to NaSCAr. This command could be based on the architectural freezing described in [Américo *et al.* 2012]. On non-critical phases, it would undeploy all contributions' tasks and instances, fetch back the new version of the contribution file and redeploy its entities. A different version of the command implementation could verify the differences and perform only the needed modifications without redeploying entities not concerned by the update, keeping the component's state.

Chapter 6

Conclusions and Perspectives

"There are two kinds of people, those who finish what they start and so on."

Robert Byrne

Contents

6.1 Conclusions **157**
6.1.1 Summary 157
6.1.2 Conclusions 159
6.1.2.1 Designing real-time systems with URSO 159
6.1.2.2 URSO's service-oriented component model 160
6.1.2.3 URSO framework implementation 160
6.2 Perspectives **161**

6.1 Conclusions

6.1.1 Summary

Many techniques have been developed by software engineers to tackle the software increasing complexity problem. Most of them are based on the *separation of concerns* principle, which favours the modularization of software projects into smaller pieces according to their functions. Modular projects are easier to integrate and to maintain. They may also have a smaller time to market, due to the fact that modules can be produced independently by different development teams. In addition, their costs may be decreased by the reuse of old and third-party modules. The *component-based* design is a very popular approach for the development of modular systems. In this approach, *components* are software units with well-defined interfaces, which state which functionalities can be seen by other components. By means of these interfaces, components connectors can be defined and information can be exchanged. Component models are used in various application domains, from graphical user interfaces to automotive electrical and electronic systems.
Besides modularity, *flexibility* is an important feature for systems which interact with real-world entities. Flexibility is the ability to adapt a software's architecture at run-time, without downtime or with a minimized one. Flexibility is crucial for systems which must react to environmental changes, and for systems whose unavailability could lead to disasters. Updates, additional functionality and bug fixes for these systems must be installed through a mechanism which dynamically substitute remove, substitute and incorporate new system components. Service-oriented architectures are an architecture style which enables the development of flexible systems. This paradigm is based on the concept of service,

which designates a software functionality defined by a contract. Software units implementing this contract (called service providers) publish their services in a service registry, which is a catalogue associating service interfaces which software implementations. Software units requiring functions defined in a contract (called service consumers) query for software implementations in the service registry. The service registry returns the query with a proxy for the service provider, so that consumers may invoke their services directly. Upon the departure of a service provider, notifications are sent by the service registry to service consumers. This way, they may query for new implementations. The service-based approach may be incorporated with the component-based one, generating service-oriented component models, which benefit from the advantages of both approaches.
Real-time systems are systems whose correctness depend on both logical and temporal correctness. They present timing constraints, whose enforcement classify them onto soft, firm and hard real-time systems. In soft real-time systems, results obtained after the violation of a timing constraint are considered, but the system continues executing and its quality of service is seen as degraded. in firm real-time systems, these results may not be considered, but the system is not stopped either. However, in hard real-time systems, deadline violations may lead to catastrophic results, so the results are discarded and the system is put in a failure state. Thus, it is important for real-time systems to bound the execution time of all its sub-components, mastering the latency sources, to reduce processing time fluctuation and to provide means to guarantee that deadlines will not be transgressed.
The lack of predictability in the service-oriented architectures has impeded the adoption of service-oriented component models in real-time applications. The last decade has seen the emergence of the first works adapting service-based paradigm and technologies to real-time systems. Still, none of these works was dedicated to the technology-agnostic development and deployment of service-based real-time applications. URSO, the main contribution of this thesis, is a metamodel which relates concepts from service-based, component-based and real-time domains and provides a methodology for the design and development of modular and flexible real-time systems. URSO is organized around three main concerns:

- Deployment, which contains concepts associated to the underlying platform, its resources, operational modes and the mapping between software entities to execution platform entities;
- Assembly, whose concepts are related to the service-based software components composition; and
- Behaviour, which contains concepts cognate with the timeliness and internal functioning of service implementations.

We have shown mappings from URSO to other component models used in both service-oriented and real-time domains: SCA, AADL and the MARTE profile for UML. These models were chosen because of their extensibility, which enabled the development of extensions to add concepts which were lacking in order to establish URSO mappings. In SCA, the extensions were developed as a binding extension for dynamic systems, a implementation extension for real-time systems resource usage, an interface extension for real-time service descriptions, and a deployment extension consisting of a platform description and deployment plans. In AADL, we have developed special types of thread group components to represent provided and required services, URSO components and composites and service registries. In UML-MARTE, we have used the stereotypes provided by the SoaML profile to define URSO components and composites.
In order to validate our proposition, we have implemented a proof-of-concept framework based on our SCA extensions. This framework was developed on an OSGi platform and

mapped URSO-SCA components to Apache iPOJO components. Shell commands were implemented to provide a URSO platform description, to enable the hot deployment and undeployment of SCA contributions (which may contain composite instances and tasks), to change operational modes and to return information about the current tasks, instances and contributions deployed on the platform. We have tested the timeliness of our framework by adapting CD_x, a collision detection benchmark application initially proposed to evaluate real-time Java virtual machines. Our CD_x version, called DCD_x, enables the hot deployment of new aircrafts, which is used to dynamically increase the system charge and evaluate this impact on the logical and temporal correctness of our implementation. Our implementation has shown to respect most real-time deadlines and deliver a deterministic execution.

6.1.2 Conclusions

6.1.2.1 Designing real-time systems with URSO

URSO provides a way to model service-oriented component-based real-time systems. Since it targets real-time systems, it is mandatory to provide mechanisms to guarantee the hosted components execution timeliness. Among these mechanisms, it is common to use isolation (temporal and spatial), WCET analysis, model checking, resource usage enforcement and access to low-level and hardware facilities. Among these items, only the last one is not compatible with URSO: in view that it encourages technology-agnostic and portable applications, it is not recommended to rely on platform-dependent structures. It is possible though to use a service-based approach to do so, like wrapping these facilities in a service provider and publishing the interface to make it available to other components.
A big part of URSO's metamodel is based on descriptions provided by the platform administrator and the application deployer. A more reliable solution could be estimating resource usage, detecting implemented and required services, and automatically creating a suitable mapping according to the available resources. Despite being more code-intrusive than URSO's current approach (it inspects all implementation classes' object code to estimate methods' WCETs), deducting data from code inspection can be more dependable than relying on information manually input by human administrators and deployers, which can be error-prone.
Isolation in URSO is covered by the resource reservation mechanism (temporal isolation) and node mapping (spatial isolation). As stated above, for the moment, it is based on descriptions provided by administrators. For quantifiable resources, their availability is automatically managed by the platform as new components are deployed. A resource usage enforcement module can be deployed in the platform to ensure that component implementations' used resource amount is not superior than the one that was declared. Again, node partition is another information manually declared by the administrator. We assumed here that nodes' non-shared resources are isolated, so that faults can not be propagated. A resource verifier module can be installed in URSO as well to verify whether this property holds for the current set of nodes.
Tasks are the only active entities in URSO. Usually, real-time component models and design methodologies associate triggers to tasks, in order to specify their activation and release pattern. In URSO this information is obtained through task properties. For the moment, clocks ,clock expressions and clock specification languages are not integrated with the model, but they could be present in future version to enable the specification of complex activation patterns.
Mode transitions have also room for improvement in future versions. For the moment they

are triggered by a platform administrator through shell commands, but rules could be defined in a way that transitions may take place automatically.
WCET analysis in URSO follows a simple approach which has already been adopted in several works in the past. Control flow graphs are created through code inspection. The WCET of each path in graph is calculated by summing the WCET of each node in the graph, and the WCET of each method is the maximum WCET among all paths. Once a method WCET is evaluated, it is then put in a table which may be consulted to evaluate other methods. However, this estimation does not take into account platform processors' internal architecture, which could produce a tighter and less pessimist WCET, to avoid idle times and be able to schedule more tasks.

6.1.2.2 URSO's service-oriented component model

URSO's component model supports the design and implementation of real-time service-oriented components. Timing information is not present in the composition though; it is either informed at thread level, or inferred by the run-time after object code inspection. It makes sense, since the same composition may have different timing characteristics depending on the available component implementations. Still, it could be possible to add timing (WCET) restrictions at service selection level.
Despite URSO is technology-independent and could potentially be used to establish a communication between components developed in different technologies, the cost of this communication must be somehow informed to the platform. For this reason, special types of binding protocol could be used. In addition, intercommunications between machines could also be enriched with their own properties, such as latency, distance and link availability.
Differently from AADL, where even execution platform entities are components, in URSO only software and application entities can be componentized. That creates a gap in the levels of detail of software and hardware elements. Modelling platform entities as components could ease the work of the platform modeller. Their functionality could be modelled as services as well, just as in MARTE. Mapping instances could be seen as a special type of connection between software and hardware entities. As a consequence, this approach could make design and analysis simpler, as software and hardware elements are processed equally.
Indeed, the platform/hardware modelling in URSO is still very primitive, providing only a few abstractions to model resources, machines and machine partitions. An approach more based on MARTE's metamodel, which provides a full framework for modelling both hardware and software resources could be more fit for future versions.
Technical components are not fully specified in URSO. They are actually used by the framework implementation, so their interface and mechanisms used to switch among implementations at run-time may be technology-dependent. Given that, it is important that the framework provides a mechanism enabling technical components extensibility and substitutability.

6.1.2.3 URSO framework implementation

The proof-of-concept implementation presented in this work was developed in Java, on top of an OSGi platform. It is known that Java applications may be very unpredictable when it comes to execution time determinism. Although part our application was developed using the RTSJ [Bollella & Gosling 2000] classes and the whole run-time was executed on a real-time JVM, the underlying OSGi Service Platform and iPOJO component framework were not re-factored and re-implemented using real-time classes, what may have brought some unpredictability to the final result.

In addition, although some features needed for real-time systems are described in the RTSJ specification, they may not have been implemented (or present a mock implementation) in currently available JVMs. That is the case for resource usage enforcement and schedulability analysis. While the former can be done based on the works of Richardson *et al.* in [Richardson *et al.* 2009], the latter can be achieved by implementing any scheduling analysis algorithm in Java based on the information provided by URSO task properties.
Another possible improvement for our implementation could be to use OSGi's remote services communication with URSO frameworks in other OSGi platforms, potentially written in other programming languages (like Celix[1], in C, or Pelix [Calmant *et al.* 2012], in Python). A desirable feature as well would be a graphical monitor enabling both modelling and runtime monitoring of components and tasks. The modelling part could generate and display the equivalent model representation in extended versions of SCA, AADL and UML-MARTE, as presented in this thesis.

6.2 Perspectives

Service-oriented real-time systems is a relatively new research domain. We have presented in this thesis a contribution to the integration of some concepts from both paradigms for modelling and developing applications. We believe there is much yet to be done in integrating more complex real-time and performance-related aspects in service-based systems. Similarly, there are several open issues in the use and adaptation of service-based systems and their internal mechanisms to real-time systems. The next paragraphs list some points which can be considered as continuation of this work.

Parallelized service invocations URSO could be improved with annotations on service composition invocations to indicate parallelizable invocations. For instance, if a series of calls in performed in each item of a collection, and these invocations are stateful and independent from one another, URSO could create threads to invoke them in parallel depending on the quantity of schedulable units currently executing in the platform. This would different from using BPEL's parallel activity in the fact that it would be automatically handled by the platform itself, which could find the best way to divide the data set. This could improve the performance of our framework and reduce the WCET of service compositions. To this end, a URSO framework implementation could be based on component models for grids, like ProActive' [Baduel *et al.* 2006], in order to create parallelizable communication between URSO components.

Probability-based WCET estimation The use of probabilities to model real-time service-based systems' behaviour and perform quantitative model checking has been studied in [Calinescu *et al.* 2012]. URSO could improve the initial WCET expression with information extracted at run-time about the probability of taking a particular path in the control flow graph.

Deployment platform description refinement As stated before, the URSO platform model in the deployment concern could be improved based on UML's MARTE hardware resource model. This improvement could affect the description of resources, communication protocols, machines, nodes and machine interconnections.

[1] Apache Celix is an implementation of the OSGi specification written in C. More information in `http://incubator.apache.org/celix/`.

Integration of other communication and programming paradigms Although we target service-based applications, sometimes only communicating through services may not be fit for real-time applications. Indeed, when it comes to the exchange large pieces of data, a data flow or data service accessed by components may be a better solution. Likewise, event-based communication is often supported in service-based applications and may constitute a better solution for notifications between components than using the service approach. Event-driven SOA is also known as 'SOA 2.0' [Laliwala & Chaudhary 2008, Papazoglou & Heuvel 2007]. An implementation integrating the OMG DDS standard [Object Management Group 2007] or MQTT[2], which add low-latency and QoS features to the communication layer, would be interesting. DDS integration in SCA has already been studied by [Labejof *et al.* 2012] in the R-MOM framework. Another important integration would be that with asynchronous frameworks or programming languages to support asynchronous service calls [Gostelow & Plouffe 1978]. This programming model recommends using the Reactor design pattern [Schmidt *et al.* 2000] to avoid locks between applications and improve performance. Asynchronous programming is commonly found in application web and SIP servers. A solution to integrate it with NaSCAr could include using RTSJ or EJB asynchronous event handling for service calls, using Future objects and integrating Reactor[3] or Vert.x[4] frameworks.

Platform-based WCET estimation As mentioned before, our WCET is very pessimistic, as it does not take into account processors organization and architecture. Instructions pipeline, cache memories, hyper-threading, branch predictors and speculative execution are optimization techniques found in most general-purpose processors nowadays, which could greatly reduce the execution time of an instruction sequence. A too pessimistic WCET estimation could lead the framework to schedule a smaller set of tasks than it could, adding idle times to the scheduler. The integration of cache memories (for both instructions and data) and pipelined execution in the WCET estimation is investigated in many works, since they constitute well-mastered architectural elements. Besides internal processor features, multi-core aspects must also be considered. An overview on methods and tools for WCET estimation can be found in [Wilhelm *et al.* 2008]. Future URSO framework implementations could integrate third-party WCET analysis tools such as Heptane[5], SWEET[6] or AbsInt's aiT[7].

URSO Cloud extension URSO could be extended to address the cloud computing [Hayes 2008], incorporating features such as business models for pricing and the possibility to provide the platform as a service. Several other aspects would require special attention, such as security, elasticity and the need to include users in model itself.

Automatic resource-based instance and task placement As remarked before, URSO is based on manual declarations of resource capabilities and resource requirements. It also

[2]MQTT stands for MQ Telemetry Transport. It is a publish/subscribe protocol over TCP/IP networks conceived for sensor-based systems and devices. Its specification is available at `http://www.ibm.com/developerworks/webservices/library/ws-mqtt`.

[3]Reactor is an open-source framework developed by Spring for asynchronous Java applications. More information about Spring Reactor is available at `https://github.com/reactor/reactor`.

[4]Vert.x is an open-source polyglot framework for concurrent, non-blocking and distributed applications. More information about Vert.X framework is available at `http://vertx.io/`.

[5]Heptane is a static WCET analysis tool developed at INRIA Rennes, in France. More information about Heptane is available at `http://bit.ly/1aufXFI`.

[6]SWEET is a static WCET analysis tool developed at Mälardalen University, in Sweden. More information about SWEET is available at `http://www.mrtc.mdh.se/projects/wcet/sweet/`.

[7]Available at `http://www.absint.com/ait/index.htm`.

depends on the node mappings expressed in deployment plans. Based on the current resource availability, the platform could automatically establish mappings towards execution platform entities for both instances and tasks. Task placement could take into account the placement of its service composition composite, just as the availability of processor units.

Automatic resource and service requirement detection As stated in the last section, the use of code inspection to estimate resource usage and service requirement and provision could improve the reliability of the component information used for WCET calculation and resource reservation mechanisms.

Implementation on other service-based technologies At the implementation level, it could be interesting to evaluate the effort and performance of URSO implementations based on other service-based technologies, like web services, or real-time technologies, like code generated by AADL-based tools. It would also be interesting to establish comparisons with an implementation based on extensions to complete SCA run-times, like Apache TuSCAny or OW2 FraSCAti.

Appendix A

Appendix

Contents

A.1 URSO name and logo **165**
A.2 Example: URSO Potential Use Cases Description **166**
A.3 Example: Mapping URSO Composite to AADL **170**
A.4 UML MARTE Simplified Metamodel **174**
A.4.1 Core Elements 174
A.4.2 Non-Functional Properties 174
A.4.3 Time 175
A.4.4 Generic Resource Model 177
A.4.5 Allocation 177
A.4.6 Generic Component Model 177
A.4.7 High Level Application Model 179
A.4.8 Generic Quantitative Analysis Model 179

A.1 URSO name and logo

'Urso' means *'bear'* in Portuguese; that is why URSO's logo (presented in Figure A.1) reminds a bear paw. In the back of the hand, the three circles bound by a white triangle shows the real-time and distributed aspect of URSO - three clocks synchronized as one. The fingers make an allusion to the UML notation for the provision (elliptic ball) and requirement (socket) of services, the main feature of service-oriented architectures.

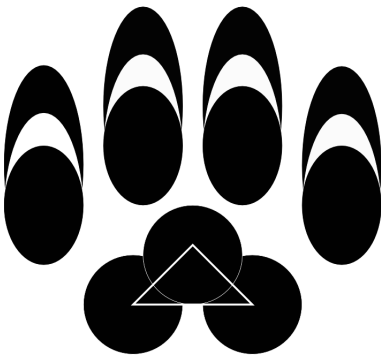

Figure A.1: URSO Component metamodel logo

A.2 Example: URSO Potential Use Cases Description

In this section, we will detail the URSO potential use-cases. For sake of simplicity, the corresponding UML use-case diagram is re-presented in Figure A.2.

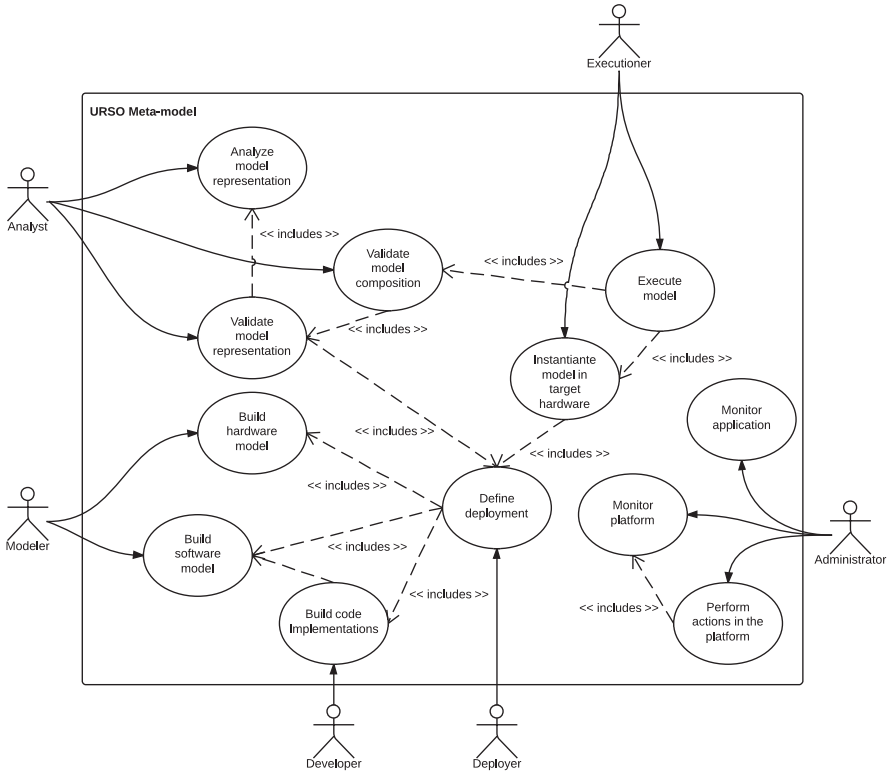

Figure A.2: Potential use cases for the URSO metamodel

Details of the use case "Build Software Model"

- Actor: Modeler
- Description: The modeler builds a URSO-compliant model of a software entity. The model can be built through the use of a model conform to URSO metamodel or through the use of URSO concepts directly. In the former case, the software description will be converted to a URSO representation of the entity. The resulting model is component-based, software-oriented and may or not have real-time properties and constraints.
- Deliverable: A URSO compliant model of the software system with its properties and restrictions.

Details of the use case "Build Hardware Model"

- Actor: Modeler
- Description: The modeler builds a URSO-compliant model of a hardware entity. The model can be built through the use of a model conform to the URSO metamodel or directly through the use of URSO concepts. In the former case, the hardware

description will be converted to a URSO representation of the entity. The resulting model must contain the system resources, connections, subcomponents and their real-time characterization.

- Deliverable: A URSO compliant model of the hardware system with its properties.

Details of the use case "Build Code Implementations"

- Actor: Developer
- Description: The developer builds an implementation of a software modeled beforehand in a specific technology. The implementation must take into account the properties and characteristics of the model it implements, that is, provide and consume the services specified by the model. The developer must also specify the resources needed for the execution and deployment of the implementation.
- Deliverable: An executable code implementing a URSO software model, along with its resource usage description.

Details of the use case "Define Deployment"

- Actor: Deployer
- Description: Based on the physical platform description (hardware model) and on the resource usage of the code implementations, the deployer creates a mapping between instances of these implementations and entities in the physical platform.
- Deliverable: A mapping between software entities grouped in a deployable unit and hardware entities.

Details of the use case "Analyze Model Representation"

- Actor: Analyst
- Description: The analyst verifies if all resources were correctly described (*e.g.* if resources in sub-components are part of the resources of their parent component). The analysis phase may statically generate meta-information for dynamic aspects (for instance, define an expression to estimate the WCET of an entity whose bindings will be defined at run-time).
- Deliverable: Error in the case of an invalid model representation, and meta-information for dynamic aspects. It may also output results of the analysis and store this information in the platform itself.

Details of the use case "Validate Model Representation"

- Actor: Analyst
- Description: This use case uses analysis results to say whether a model can be deployed or not in the platform.
- Deliverable: The analysis output.

Details of the use case "Validate Model Composition"

- Actor: Analyst

- Description: This use case uses the validate model representation use case to validate all the parts of the composition separately. Then it takes into account all the analysis result that depend on other parts of the system and verifies whether all model representations are still valid.
- Deliverable: It returns the analysis result for all the composition (error, if at least one element is invalid).

Details of the use case "Instantiate Model in Target Hardware"

- Actor: Executioner
- Description: In this use case, the executioner creates instances of the implementations on the target physical components of the platform, reserving all necessary resources for its correct execution.
- Deliverable: The platform with the entities instantiated.

Details of the use case "Execute Model"

- Actor: Executioner
- Description: For all executable entities (tasks), after the verification and (schedulability) analysis of the complete composition, the executioner instantiates them and adds them to its pool of tasks to be performed. If the executable entity has a dependency towards other entity, the executioner first verifies whether the task it depends on is already in the pool. If it is not, the task is not added to the pool. The scheduling policy used to execute tasks (and thus define their priority) may depend on the current state of the platform.
- Deliverable: The platform with the executable entity in the pool of the eligible tasks.

Details of the use case "Monitor Application"

- Actor: Administrator
- Description: The Administrator may access the state of all instantiated software entities and their compositions to monitor whether errors have been produced or constraints have violated. The monitor may also provide a model @ run-time view of the entities, making the correspondence between the URSO concepts and the instantiated elements.
- Deliverable: The state of the concerned software entities.

Details of the use case "Monitor Platform"

- Actor: Administrator
- Description: The Administrator may access the state of all platform entities and their compositions to monitor whether errors have been produced or constraints have violated. The monitor may also provide a model @ run-time view of the entities, making the correspondence between the URSO concepts, the platform elements and the software elements related to it.
- Deliverable: The state of the concerned platform elements.

Details of the use case "Perform Actions In The Platform"

- Actor: Administrator
- Description: The Administrator may execute commands on the platform and software entities. The state of the platform changes accordingly, and it may imply the revalidation of several elements.
- Deliverable: The platform with the updated state.

A.3 Example: Mapping URSO Composite to AADL

Listing A.1: Example: AADL Declaration for URSO Composite

```
process implementation URSOComposite.C1 extends URSO.i
  subcomponents
    C11: thread group URSOComponent.C11;
    C12: thread group URSOComponent.C12;
    LSR: thread ServiceRegistry.local;
  connections
    -- S2
    event data port C12.newService -> LSR.newService in modes (queryC12,
        int12);
    event port C12.actService -> LSR.actService in modes (int12, queryC12);
    event data port C12.removeService -> LSR.removeService in modes (
        queryC12, int12);
    -- R1
    event data port LSR.queryResult -> C11.queryResult in modes (queryC11);
    event data port C11.newQuery -> LSR.newQuery in modes (queryC11);
    -- Bidirectional communication between C11 and C12 (service invocation)
    -- A pair of port groups for every pair of urso components
    port group C11.interactOut -> C12.interactIn in modes (int12);
    port group C12.interactOut -> C11.interactIn in modes (int12);
    -- Exported Services
    -- S1
    event data port C11.newService -> GSR.newService in modes (queryGC11,
        intG11);
    event port C11.actService -> GSR.actService in modes (intG11, queryGC11)
        ;
    event port data C11.removeService -> GSR.removeService in modes (intG11,
         queryGC11);
    -- S3
    event data port C12.newService -> GSR.newService in modes (queryGC12,
        intG12);
    event port C12.actService -> GSR.actService in modes (intG12, queryGC12)
        ;
    event port C12.removeService -> GSR.removeService in modes (intG12,
        queryGC12);
    -- Imported Dependencies
    -- R3
    event data port GSR.queryResult -> C11.queryResult in modes (queryGC11);
    event data port C11.newQuery -> GSR.newQuery in modes (queryGC11);
    -- R2
    event data port GSR.queryResult -> C12.queryResult in modes (queryGC12);
    event data port C12.newQuery -> GSR.newQuery in modes (queryGC12);
    -- Data exchange with unknown entities
    port group C11.interactOut -> GMM.interactIn in modes (intG11);
    port group C12.interactOut ->GMM.interactIn in modes (intG12);
    port group GMM.interactOut -> C12.interactIn in modes (intG12);
    port group GMM.interactOut -> C11.interactIn in modes (intG11);
  modes
    initialize: initial mode;
    -- Internal query modes
    queryC12: mode;
    queryC11: mode;
    -- Global query modes
    queryGC12: mode;
    queryGC11: mode;
    -- Local interaction modes
    intC11 : mode;
    intC12 : mode;
    -- Mode for the interaction with multiplexer
    intGC11: mode;
    intGC12: mode;
    -- Initial transition
    initialize -[initialized]-> queryC11;
    initialize -[initialized]-> queryC12;
    initialize -[initialized]-> queryGC11;
    initialize -[initialized]-> queryGC12;
end URSOComposite.i;
```

Listing A.2: Example: AADL Declaration for URSO Component Implementation

```
thread group implementation URSOComponent.C11
  subcomponents
    -- They could also have been grouped into two thread groups
        ServiceRequest and ServiceProvision
    R1: thread group ServiceRequester.C11R1;
    R3: thread group ServiceRequester.C11R3;
    S1: thread group ServiceProvider.C11S1;
  connections
    -- In this case we only need to differentiate the local query mode from
        the remote one.
    -- In a general case, we would need to specify modes for different
        dependencies' queries
    -- R1
    event data port R1.newQuery -> newQuery in modes (queryC11);
    event data port queryResult -> R1.queryResult in modes (queryC11);
    port group interactIn -> R1.interactIn in modes (intC11);
    -- R3
    event data port R3.newQuery -> newQuery in modes (queryGC11);
    event data port queryResult -> R3.queryResult in modes (queryGC11);
    port group interactIn -> R3.interactIn in modes (intGC11);
    -- S1
    event data port S1.newService -> newService in modes (intC11, queryC11);
    event port interactOut.eventOut -> actService in modes (intC11);
    event port actService -> S1.actService in modes (intC11);
    port group  S1.interactOut -> interactOut in modes (intC11);
    port group interactIn -> S1.interactIn in modes (intC11);
  modes
    queryC11 : initial mode;
    intC11: mode;
    queryGC11: mode;
    intGC11: mode;
end URSOComponent.C11;

thread group ServiceProvider
  features
    actService: in event port;
    newService: out event data port URSOServiceRecord;
    interactOut: port group basic::output_PT;
    interactIn: port group basic::input_PT;
    removeService: out event data port URSOServiceRecord;
end ServiceProvider;

thread group ServiceRequester
  features
    newQuery: in out event data URSOServiceQueryRecord;
    queryResult: in out event data URSOServiceRecord;
    interactIn: port group basic::input_PT;
    interactOut: port group basic::output_PT;
end ServiceRequester;
```

Listing A.3: Example: AADL Declaration for URSO Services and Dependencies

```
thread group implementation ServiceProvider.C11S1
  subcomponents
    OP1: thread OperationOne in modes (op1);
    OP2: thread OperationTwo in modes (op2);
  connections
    event data port OP1.newService -> newService in modes(queryC11, intC11);
    event data port OP2.newService -> newService in modes(queryC11, intC11);
    event data port OP1.removeService -> removeService in modes(queryC11,
        intC11);
    event data port OP2.removeService -> removeService in modes(queryC11,
        intC11);
    event port interactOut.eventOut -> activeThread in modes (intC11);
    event port actService -> OP1.service in modes (intC11, op1);
    event port actService -> OP2.service in modes (intC11, op2);
    data port OP1.result -> interactOut.dataOut in modes (intC11);
    data port OP2.result -> interactOut.dataOut in modes (intC11);
  modes
    queryC11: initial mode;
    queryGC11: mode;
    intC11: initial mode;
    intGC11: mode;
    op1: mode;
    op2: mode;
    intC11 -[ interactIn.eventIn ]-> op1;
    intC11 -[ interactIn.eventIn ]-> op2;
end ServiceProvider.C11S1;

thread group implementation ServiceRequester.C11R1
  subcomponents
    Requester: thread Requester in modes (queryC11);
    Interaction: thread Interaction in modes (intC11);
  connections
    event data port Requester.newQuery -> newQuery in modes (queryC11);
    event data port queryResult -> Requester.queryResult in modes (queryC11)
        ;
    event port Requester.Activation -> Interaction.Activation in modes (
        intC11);
    port group Interaction.interactOut -> interactIn in modes (intC11);
    port group interactOut -> Interaction.interactIn in modes (intC11);
  properties
    Dispatch_Protocol => Periodic;
    Period => 30 Ms;
  modes
    queryC11 : initial mode;
    intC11: mode;
end ServiceRequester.C11R1;

thread group implementation ServiceRequester.C11R3
  subcomponents
    Requester: thread Requester in modes (queryGC11);
    Interaction: thread Interaction in modes (intGC11);
  connections
    event data port Requester.newQuery -> newQuery in modes (queryGC11);
    event data port queryResult -> Requester.queryResult in modes (queryGC11
        );
    event port Requester.Activation -> Interaction.Activation in modes (
        intGC11);
    port group Interaction.interactOut -> interactIn in modes (intGC11);
    port group interactOut -> Interaction.interactIn in modes (intGC11);
  properties
    Dispatch_Protocol => Periodic;
    Period => 30 Ms;
  modes
    queryGC11 : initial mode;
    intGC11: mode;
end ServiceRequester.C11R3;
```

We present an example of URSO component described in AADL. The composite is named C1, and it has two inner components: C11 and C12. The component C11 has one service, named S1, and two references, named R1 and R3. The component C12 has two services, S2 and S3, and one reference, R2. The services S1 and S3 are exported. The dependencies R2 and R3 are imported. The corresponding AADL declarations for that given architecture is depicted in the Listing A.1.
Listing A.2 and Listing A.3 exemplify the implementation of an URSO Component. We have decided to detail the implementation of the component C1/C11 described above. Each dependency or service correspond to an inner component of type Service Requester or Service Provider. Service requesters contain two periodic threads, one for interacting with services and another for querying for services. Service providers have one aperiodic thread and mode per service operation.

A.4 UML MARTE Simplified Metamodel

This section details the metamodel of the MARTE profile for UML, complementing the information presented in the section 4.3.1. The diagrams and explanations here presented depict a simplified version of the original metamodel. Due to the size and complexity of MARTE, we have created one UML metamodel diagram for each package, with some cross-cutting concerns. The original metamodel can be found in [Object Management Group 2008].
This section does not cover the metamodel of the Detailed Resource Modelling, Schedulability Analysis Model and Performance Analysis Model packages, as they are specializations of other concepts presented below and not essential to the understanding of the MARTE model.

A.4.1 Core Elements

The simplified metamodel of the core elements package is shown in Figure A.3. A model element may represent either a classifier or an instance. Models elements may contain other elements. Classifiers correspond to types, whereas instances represent run-time instantiations of the very same concepts. For example, in this metamodel, the pairs Behaviour - Behaviour Execution, Composite Behaviour - Comp Behaviour Execution, Action - Action Execution, Event - Event Occurrence, Start Event - Start Occurrence have a relationship classifier - instance. A Behavioured Classifier is an element type which possesses behaviour. A behaviour may be composed by an action or a composition of other behaviours. A special type of behaviour is the mode behaviour, that is, a behaviour that depends on the current operational mode. A mode corresponds to a configuration of the platform. Modes can be switched from one another, through mode transitions. Events can be actively generated by means of triggers in mode transitions and behavioural elements. They are automatically generated in case of invocation of other behaviours. This invocation mechanism generates first an invocation event occurrence, that results a request. Upon the reception of the on the target element generates a receive event occurrence and may execute a behaviour depending on the request content. The execution of a behaviour also generates an event for the beginning of the execution and an event for the end of the execution.

A.4.2 Non-Functional Properties

The metamodel presented in Figure A.4 depicts part of the metamodel of the Non-functional properties (NFP) package in UML Marte. Grey concepts represent concepts from other packages. It corresponds to two of the three packages in the NFP package, namely NFP_Nature and NFP_Annotation. The third package, NFP_Declaration, relates properties, their values and their types with the VSL specification and is not depicted in the Figure. NFPs are either quantitative or qualitative. Qualitative NFPs corresponds to high-level labels which may be parametrized with other NFPs. Quantitative NFPs are characterized by measures and sample realizations. The measure is often a statistical function applied to the domain of values in the sample realization. Measures have a measurement units and a quantity. This quantity may either be a base quantity (like length) or derived (like speed, which is length per time).
Annotated elements designate elements which may be annotated with NFPs. Mode-dependent constraints may be defined as well. Sets of NFPs may be grouped according to a modelling concern.

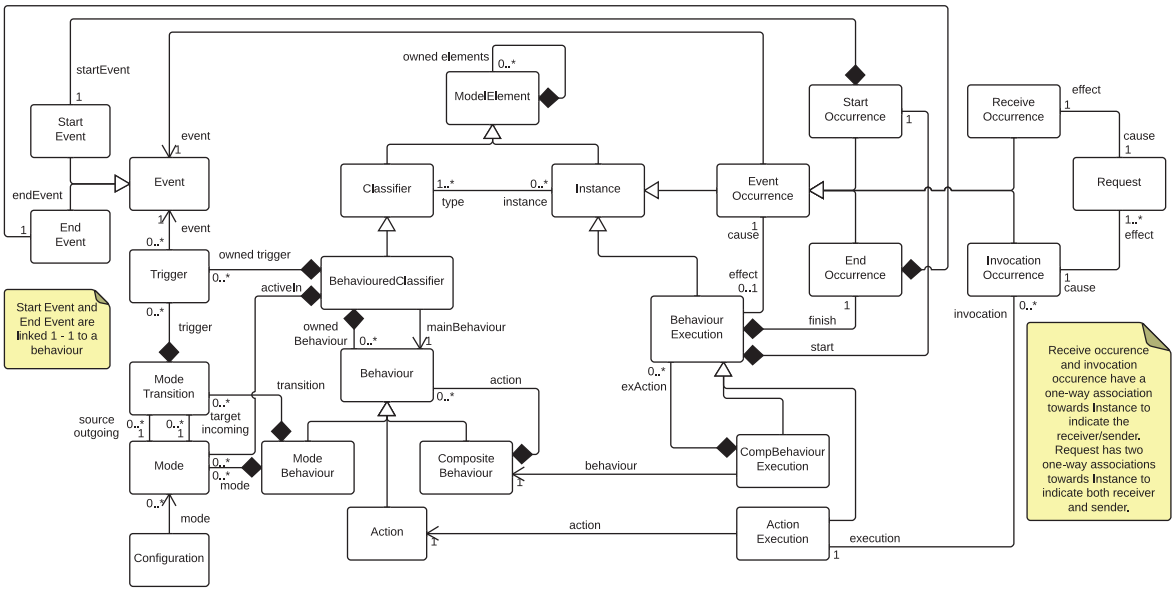

Figure A.3: UML MARTE - Core Elements

A.4.3 Time

Figure A.5 depicts a simplified metamodel for the Time package. A time base is the most basic concept, which gathers an ordered non-empty set of instants. It can be extended to a discrete time base when the nature of the instants is discrete. Time bases can be grouped into a multiple time base structure, and relations can be defined between these time bases and their instants. Relations between instants can be established as well. Instants are used to define interval boundaries. A clock is a structure that can be used to access time measures. A clock is linked to a time base and produces periodic events called ticks. A clock measures time on a determined unit of time. Time values refers to a time instant in a given clock. Instant values specialize time values to denote a set of junction instants. A time interval value uses two instant value references to denote time intervals. Duration

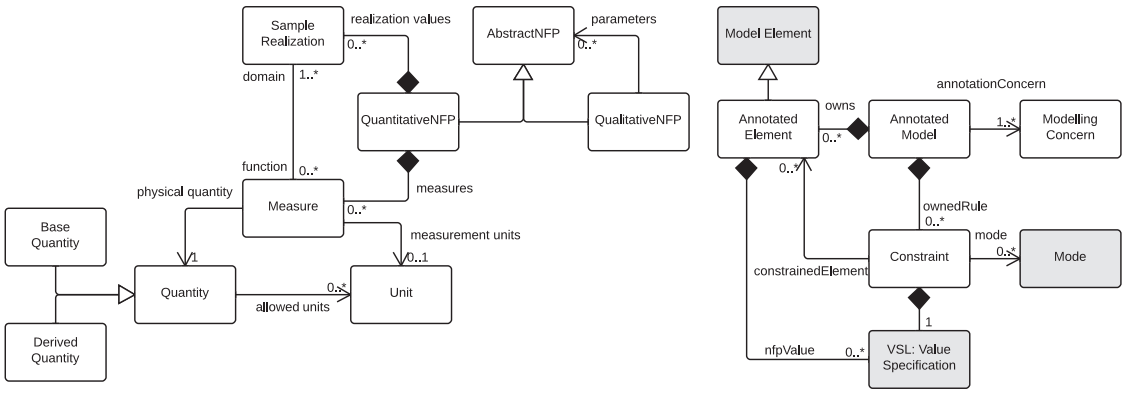

Figure A.4: UML MARTE - Non-Functional Properties

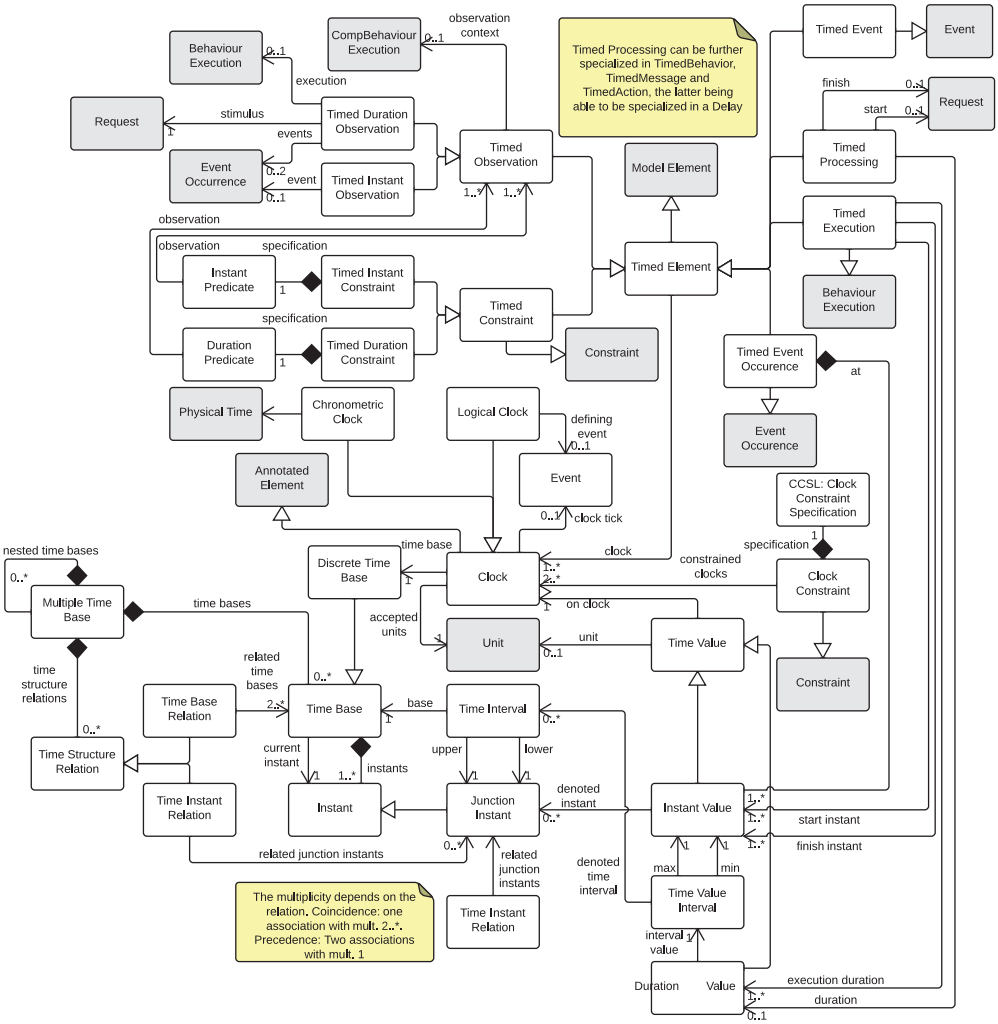

Figure A.5: UML MARTE - Time

values make reference to a time span. Constraints can be defined on two or more clocks on a set of clocks. These constraints can be expressed using a special language called Clock Constraint Specification Language (CCSL).
Timed elements are model elements associated with a non-empty set of clocks. There are different types of timed elements:

- Timed Observations, which observe the time associated to an event occurrence, or the duration of an execution, request or between two event occurrences. Timed observations are performed in an behaviour execution context;

- Timed Constraints, which imposes constraints on the occurrence of events or in the duration of an execution. The predicates used to specify the constraints may contain timed observations;

- Timed Event and Event Occurrences, which associates the occurrence and repetition of events to instant values and clock value specifications;

- Timed Execution, which associates the beginning and the end of an execution to instant values, and its execution duration to a duration value;
- and Timed Processing, which generalizes activities with known processing times (bound by a duration or start and finish events), and may be extended to timed messages (which also extends requests), timed behaviour (which extends behaviour) and timed action (which extends action). A particular timed action is called Delay, which is a null operation that lasts for a determined duration.

A.4.4 Generic Resource Model

Figure A.6 presents a simplified model for the Generic Resource Modelling (GRM) framework in MARTE. Resources are classifiers for Resource Instances, and they may own other resources. They provide services which at tun-time correspond behaviour executions. Examples of services are *acquire*, *release*, *get amount available* and *activate*. They can be annotated with NFPs and contain references towards clocks. Two abstract concepts, resource reference and resource amount, can be used to represent the dynamic creation of resources and the quantification of the amount of the resource, respectively. Resources can be extended to represent storage resources (*e.g.* memory), timing resources (*e.g.* clocks), synchronization resources (*e.g.* semaphores and locks), computing resources (*e.g.* CPU), concurrency resources (*e.g.* threads, processes), communication resources (*e.g.* bus) and device resources (external devices).
Two special types of resources are responsible for resource management mechanisms: resource brokers (responsible for allocation and de-allocation) and resource managers (which controls resource creation, maintenance and deletion). Both can be linked to control policies. It is worth mentioning that the GRM foresees a special sub-package which details scheduling mechanisms. Schedulers are in fact resource brokers, which allocate and de-allocation schedulable resources (an specialization of concurrency resources). Schedulers in URSO are seen as technical components whose implementation may vary, but the architecture follows the same structure: an entity which hosts several schedulable resources that are executed in a non-empty set of processing resources according to a scheduling policy. This package metamodel will be detailed here though, as we are showing the main concepts of the whole model.
Finally, the consumption of an amount of resource is represented in the model by the resource usage entity. It is associated to an usage amount and it may use an event-based mechanism to demand the reservation. The usage amount can be classified as either static or dynamic.

A.4.5 Allocation

Figure A.7 depicts the metamodel for the Allocation package. Allocation represents the mapping from application resources (source) to execution platform resources (target). This mapping can be restricted to some constraints. The package also describes refinement, that is, the mapping between elements in different level of abstractions. Refinements can be constrained as well and operate on both ends (source and target) of the allocation.

A.4.6 Generic Component Model

Figure A.8 presents a simplified metamodel for the Generic Component Model (GCM) package. Structured components have assembly parts and interaction ports. They can connect to other components by means of connectors. There are two types of ports: flow

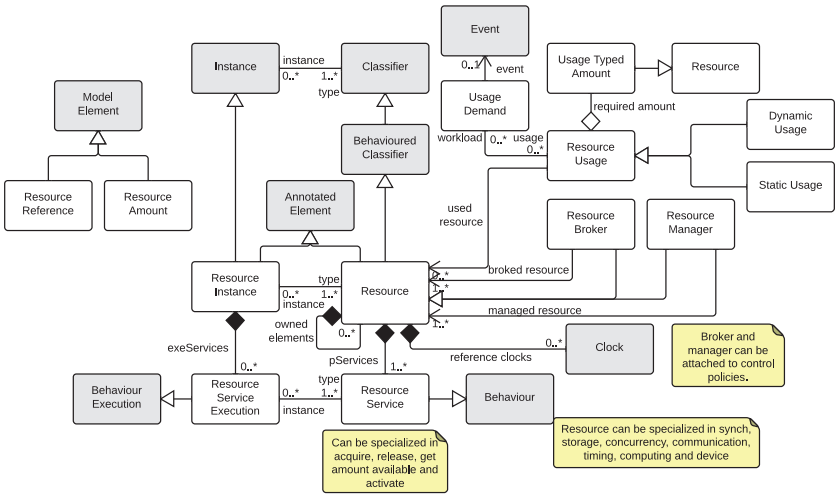

Figure A.6: UML MARTE - Generic Resource Modelling

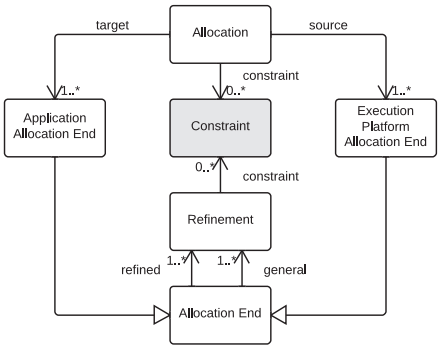

Figure A.7: UML MARTE - Allocation

ports, through which a directed data flow can be specified; and client-server ports through which remote operations can be called and received. Actions on interaction ports are called invocation actions, and may be used to send signal or data and call operations.

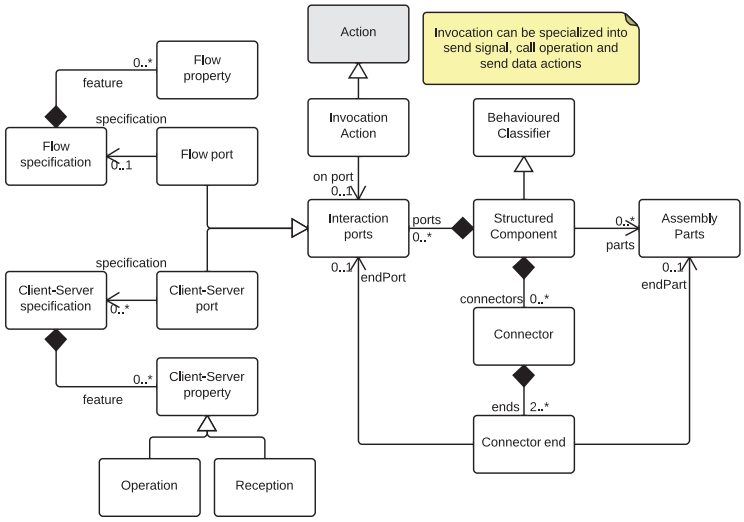

Figure A.8: UML MARTE - Generic Component Model

A.4.7 High Level Application Model

Figure A.9 illustrates a simplified version of the High-Level Application Model package. A real-time unit is a concurrency unit with services and behaviours. It also owns a message queue, which acts as an event and message pool. RtUnits may invoke services on other RtUnits and send message to them. Shared information is modelled through Passive protected units (PpUnits), whose access policy can be configured. Ppunits do not constitute a schedulable resource for themselves, and use that of the RtUnit that invoke their provided services. Scheduling policies can be defined to determine the order in which messages will be consumed from the in message queue. Real-time features are information used by the internal controllers of RtUnits to control their execution. The communication aspects of the interaction between RtUnits is captured by the concept of RtAction, which may contain real-time features as well.

A.4.8 Generic Quantitative Analysis Model

Figure A.10 depicts a simplified version of the General Quantitative Analysis Model package. This package is centred around the concept of analysis context. An analysis context possesses a workload behaviour (a group of end-to-end system operations considered for the analysis) and the set of resources of the platform.
A workload behaviour is characterized by a set of triggering events (Workload Event), which may be generated by timed events, workload generators (which may be represented as a finite state machine) or event traces (from log files). These events are responded by a behaviour scenario from the set of scenarios of the workload behaviour. Sub-operations in the behaviour scenario are called steps, which can be refined by another behaviour scenario. Acquiring and Releasing resources are examples of steps. The precedence relation between

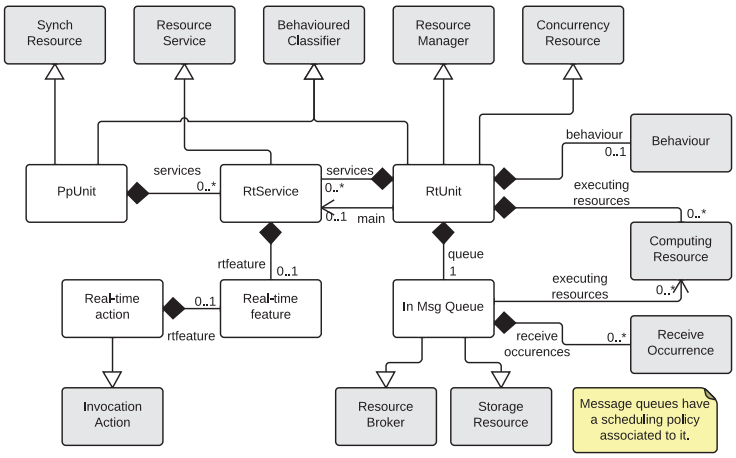

Figure A.9: UML MARTE - High Level Application Modelling

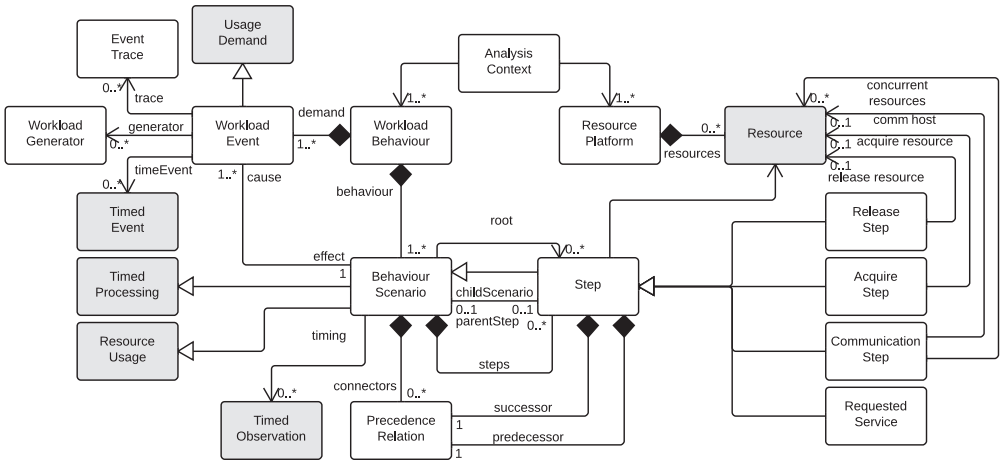

Figure A.10: UML MARTE - General Quantitative Analysis Model

steps establish a sequence of steps. Steps have time- and scheduling-related attributes such as end-to-end response time and execution priority. These measures are obtained by means of timed observations. Behaviour Scenarios are also associated to resource usages.

List of Figures

2.1 Dynamic Software Evolution 9
2.2 Service-Oriented Architecture: Actors and Interactions 13
2.3 Dynamic Service-Oriented Architecture: Actors and Interactions 14

3.1 A taxonomy of potential URSO users. 41
3.2 Potential use cases for the URSO metamodel 42
3.3 URSO Component metamodel - Deployment concern concepts 48
3.4 URSO Component metamodel - Platform definition 49
3.5 Service Registries in a Service-Oriented Hierarchical Component Model . . 55
3.6 URSO Component Model - Assembly concern 59
3.7 URSO Component Model - Component Development 60
3.8 Taxonomy of Java byte-code instructions 63
3.9 URSO Component Model - Behaviour Layer 70
3.10 URSO Component Metamodel: Deployment Concern 73
3.11 Deployment unit development process 74
3.12 Deployment Process Activity 76
3.13 Mode Change Activity 77
3.14 A Service-oriented Architecture for the Collision Detector Application . . . 82
3.15 Dynamic Collision Detector Application Architecture 83

4.1 SCA Metamodel 98
4.2 Extended SCA Metamodel: Dynamic Binding Extension 100
4.3 SCA Real-time Service Interface Extension 101
4.4 SCA Real-time Component Implementation Extension 102
4.5 SCA Real-time Deployment Extension - Platform Description 103
4.6 SCA Real-time Deployment Extension - Deployment Plan 103
4.7 SCA Assembly with Real-time and Dynamic Extensions 104
4.8 AADL Simplified Metamodel 108
4.9 SOA model in AADL 109
4.10 AADL URSO Extension: Service-related concepts 113
4.11 AADL URSO Extension: Component-related concepts 114
4.12 URSO Extensions for MARTE, based on SoaML profile 119

5.1 URSO+NaSCAr framework architecture 133
5.2 Control flow graph for the Detector Implementation 140
5.3 WCET Expression for Node L0 141
5.4 WCET Expression for Node L1 142
5.5 DCDj Service Architecture on NaSCAr 144
5.6 SCA:PlatformDesc command: XML parsing and platform creation 145
5.7 SCA:PlatformInfo command: Information retrieval and string composition . 146
5.8 SCA:Deploy command: Contributions with passive entities 148

5.9 SCA:Deploy command: Contributions with active entities 149
5.10 NaSCAr WCET estimation - passive entities services 152
5.11 NaSCAr WCET estimation - service compositions 153
5.12 Detector task deadline miss ratio 154

A.1 URSO Component metamodel logo 165
A.2 Potential use cases for the URSO metamodel 166
A.3 UML MARTE - Core Elements 175
A.4 UML MARTE - Non-Functional Properties 175
A.5 UML MARTE - Time 176
A.6 UML MARTE - Generic Resource Modelling 178
A.7 UML MARTE - Allocation 178
A.8 UML MARTE - Generic Component Model 179
A.9 UML MARTE - High Level Application Modelling 180
A.10 UML MARTE - General Quantitative Analysis Model 180

List of Tables

3.1 Mapping towards BPEL activities 57
3.2 Estimated WCET for Service Composition Activities 58

4.1 Mapping between URSO and SCA Assembly 105
4.2 Overview on the URSO extensions to SCA, AADL and MARTE 125

5.1 NaSCAr project metrics 135
5.2 Contributions' project metrics 144
5.3 Platform description parsing and validation results 146
5.4 Platform description information display results 147
5.5 Contribution deployment activity results for passive entities 148
5.6 Contribution deployment activity results for passive entities 150
5.7 Contribution deployment activity results for active entities 151
5.8 Mode change execution time 152
5.9 Estimated WCETs for services and service compositions implementations . 153

List of Algorithms

3.1 Data structures in Byte-code Control Flow Graph 64
3.2 Byte-code Control Flow Graph Construction Structure 65
3.3 Byte-code Control Flow Graph Construction - Labels 66
3.4 Byte-code Control Flow Graph Construction - Jumps 67
3.5 Byte-code Control Flow Graph Construction - Finalization 68
3.6 Byte-code Control Flow Graph Construction - Edge Update 69
3.7 Byte-code Control Flow Graph Construction - Reachability 69

List of Listings

4.1 URSO AADL Process and Data Declarations 112
4.2 URSO AADL Registry Thread Declaration 113
5.1 Test platform description 136
5.2 Test platform node description 137
5.3 Detector service composition and deployment plan 139
5.4 WCET Estimation of Detector Implementation Nodes 140
A.1 Example: AADL Declaration for URSO Composite 170
A.2 Example: AADL Declaration for URSO Component Implementation 171
A.3 Example: AADL Declaration for URSO Services and Dependencies 172

Bibliography

[Aeronautical Radio, INC 1997] Aeronautical Radio, INC. *ARINC Specification 653: Avionics Application Software Standard Interface*, 1997.

[Aghajani & Jawawi 2012] M. Aghajani and D. N A Jawawi. *An Implementation of Embedded Real Time System Framework in Service Oriented Architecture*. In Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on, pages 334–340, 2012.

[Åkerholm *et al.* 2007] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson and Massimo Tivoli. *The SAVE approach to component-based development of vehicular systems*. Journal of Systems and Software, vol. 80, no. 5, pages 655–667, May 2007.

[Al-Ali *et al.* 2002] Rashid J. Al-Ali, Omer F. Rana, David W. Walker, Sanjay Jha and Shaleeza Sohail. *G-QoSM: Grid Service Discovery Using QoS Properties*. Computers and Artificial Intelligence, vol. 21, no. 4, 2002.

[Al-Turki & Meseguer 2007] Musab Al-Turki and José Meseguer. *Real-time rewriting semantics of orc*. In Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP '07, pages 131–142, New York, NY, USA, 2007. ACM.

[Albert *et al.* 2007] E. Albert, P. Arenas, S. Genaim, G. Puebla and D. Zanardini. *Cost analysis of java bytecode*. In Proceedings of the 16th European conference on Programming, ESOP'07, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag.

[Alhaj & Petriu 2010] Mohammad Alhaj and Dorina C. Petriu. *Approach for generating performance models from UML models of SOA systems*. In Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '10, pages 268–282, Riverton, NJ, USA, 2010. IBM Corp.

[Allen & Garlan 1997] Robert Allen and David Garlan. *A formal basis for architectural connection*. ACM Trans. Softw. Eng. Methodol., vol. 6, no. 3, pages 213–249, July 1997.

[Allen *et al.* 1998] Robert Allen, Rémi Douence and David Garlan. *Specifying and analyzing dynamic software architectures*. In Egidio Astesiano, editor, Fundamental Approaches to Software Engineering, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[Alvarez *et al.* 2003] José M. Alvarez, Manuel Diaz, Luis Llopis, Ernesto Pimentel and José M. Troya. *Integrating Schedulability Analysis and Design Techniques in SDL*. Real-Time Syst., vol. 24, no. 3, pages 267–302, May 2003.

[Américo & Donsez 2012] João Claudio Américo and Didier Donsez. *Service component architecture extensions for dynamic systems*. In Proceedings of the 10th international conference on Service-Oriented Computing, ICSOC'12, pages 32–47, Berlin, Heidelberg, 2012. Springer-Verlag.

[Américo *et al.* 2012] João Claudio Américo, Walter Rudametkin and Didier Donsez. *Managing the dynamism of the OSGi Service Platform in real-time Java applications.* In Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12, pages 1115–1122, New York, NY, USA, 2012. ACM.

[Aminpour *et al.* 2011] Raziyeh Aminpour, Vahid Rafe and Mohsen Rahmanei. *Modeling and non-functional analysis of service- oriented architectures using AADL.* Scientific Research and Essays, vol. 6, no. 16, pages 3504–3513, August 2011.

[Anastasi *et al.* 2011] Gaetano F. Anastasi, Tommaso Cucinotta, Giuseppe Lipari and Marisol Garcia-Valls. *A QoS registry for adaptive real-time service-oriented applications.* In Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications, SOCA '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[André & Mallet 2009] Charles André and Frédéric Mallet. *Specification and verification of time requirements with CCSL and Esterel.* In Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '09, pages 167–176, New York, NY, USA, 2009. ACM.

[Andrieux *et al.* 2007] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke and Ming Xu. *Web Services Agreement Specification (WS-Agreement).* Technical report, Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG, 2007.

[Aniketos Project Consortium 2011] Aniketos Project Consortium. *D3.1: Design-Time Support Techniques for Secure Composition and Adaptation*, 2011.

[Arnold 1999] Ken Arnold. *The Jini architecture: dynamic services in a flexible network.* In Proceedings of the 36th annual ACM/IEEE Design Automation Conference, DAC '99, pages 157–162, New York, NY, USA, 1999. ACM.

[Atkinson *et al.* 2008] Colin Atkinson, Philipp Bostan, Daniel Brenner, Giovanni Falcone, Matthias Gutheil, Oliver Hummel, Monika Juhasz and Dietmar Stoll. *The Common Component Modeling Example.* Chapter " Modeling Components and Component-Based Systems in KobrA", pages 54–84. Springer-Verlag, Berlin, Heidelberg, 2008.

[Avizienis *et al.* 2004] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr. *Basic concepts and taxonomy of dependable and secure computing.* Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 1, pages 11–33, 2004.

[Aziz *et al.* 2013] Muhammad Waqar Aziz, Radziah. Mohamad and Dayang N. A. Jawawi. *Critical evaluation of two UML profiles for Distributed Embedded Real-time Systems design.* International Journal of Software Engineering and its Applications, vol. 7, no. 3, pages 137–146, may 2013.

[Baduel *et al.* 2006] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel and Romain Quilici. Grid Computing: Software Environments and Tools, Chapter " Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[Basanta-Val *et al.* 2013] P. Basanta-Val, M. García-Valls and I. Estévez-Ayres. *Enhancing OSGi with real-time Java support.* Softw. Pract. Exper., vol. 43, no. 1, pages 33–65, January 2013.

[Baskiyar & Meghanathan 2005] Sanjeev Baskiyar and Natarajan Meghanathan. *A survey of contemporary real-time operating systems.* INFORMATICA-LJUBLJANA-, vol. 29, no. 2, page 233, 2005.

[Basu *et al.* 2006] Ananda Basu, Marius Bozga and Joseph Sifakis. *Modeling Heterogeneous Real-time Components in BIP.* In Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM '06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[Batista *et al.* 2008] Thais Batista, Antônio T. Gomes, Geoff Coulson, Christina Chavez and Alessandro Garcia. *On the Interplay of Aspects and Dynamic Reconfiguration in a Specification-to-Deployment Environment.* In Proceedings of the 2nd European conference on Software Architecture, ECSA '08, pages 314–317, Berlin, Heidelberg, 2008. Springer-Verlag.

[Becker *et al.* 2010] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck and Arian Treffer. *Model-Based extension of AUTOSAR for architectural online reconfiguration.* In Proceedings of the 2009 international conference on Models in Software Engineering, MODELS'09, pages 83–97, Berlin, Heidelberg, 2010. Springer-Verlag.

[Becker 2008] Steffen Becker. *Quality of Service Modeling Language.* In Irene Eusgeld, FelixC. Freiling and Ralf Reussner, editors, Dependability Metrics, volume 4909 of *Lecture Notes in Computer Science*, pages 43–47. Springer Berlin Heidelberg, 2008.

[Beisiegel *et al.* 2005] Michael Beisiegel, Henning Blohm, Dave Booz, Jean-Jacques Dubray, Mike Edwards, Bill Flood, Bruce Ge, Oisin Hurley, Dave Kearns, Mike Lehmann, Jim Marino, Martin Nally, Greg Pavlik, Michael Rowley, Adi Sakala, Chris Sharp and Ken Tam. *SCA Service Component Architecture Specification - Assembly Model Specification Version 0.9.* Nov 2005.

[Benghazi *et al.* 2010] Kawtar Benghazi, Manuel Noguera, Carlos Rodríguez-Domínguez, Ana Belén Pelegrina and José Luis Garrido. *Real-time web services orchestration and choreography.* In Proceedings of the 6th International Workshop on Enterprise & Organizational Modeling and Simulation, EOMAS '10, pages 142–153, Aachen, Germany, Germany, 2010. CEUR-WS.org.

[Bergstra & Klop 1982] Jan A. Bergstra and Jan Willem Klop. *Algebraic Specifications for Parametrized Data Types with Minimal Parameter and Target Algebras.* In Proceedings of the 9th Colloquium on Automata, Languages and Programming, pages 23–34, London, UK, UK, 1982. Springer-Verlag.

[Bernat *et al.* 2000] Guillem Bernat, Alan Burns and Andy Wellings. *Portable worst-case execution time analysis using Java byte code.* In Proceedings of the 12th Euromicro conference on Real-time systems, Euromicro-RTS'00, pages 81–88, Washington, DC, USA, 2000. IEEE Computer Society.

[Bernstein & Harter 1981] Arthur Bernstein and Paul K. Harter Jr. *Proving real-time properties of programs with temporal logic.* In Proceedings of the eighth ACM symposium on Operating systems principles, SOSP '81, pages 1–11, New York, NY, USA, 1981. ACM.

[Bézivin 2005] Jean Bézivin. *Model driven engineering: an emerging technical space.* In Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering, GTTSE'05, pages 36–64, Berlin, Heidelberg, 2005. Springer-Verlag.

[Bihari & Schwan 1991] Thomas E. Bihari and Karsten Schwan. *Dynamic adaptation of real-time software*. ACM Trans. Comput. Syst., vol. 9, no. 2, pages 143–174, May 1991.

[Bini & Buttazzo 2004] Enrico Bini and Giorgio C. Buttazzo. *Schedulability Analysis of Periodic Fixed Priority Systems*. IEEE Trans. Comput., vol. 53, no. 11, pages 1462–1473, November 2004.

[Bini *et al.* 2009] Enrico Bini, Thi Huyen Châu Nguyen, Pascal Richard and Sanjoy K. Baruah. *A Response-Time Bound in Fixed-Priority Scheduling with Arbitrary Deadlines*. IEEE Trans. Comput., vol. 58, no. 2, pages 279–286, February 2009.

[Birman & Joseph 1987] K. Birman and T. Joseph. *Exploiting virtual synchrony in distributed systems*. In Proceedings of the eleventh ACM Symposium on Operating systems principles, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[Bohn *et al.* 2006] Hendrik Bohn, Andreas Bobek and Frank Golatowski. *SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains*. In Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, ICNICONSMCL '06, pages 43–, Washington, DC, USA, 2006. IEEE Computer Society.

[Bollella & Gosling 2000] Gregory Bollella and James Gosling. *The Real-Time Specification for Java*. IEEE Computer, vol. 33, no. 6, pages 47–54, 2000.

[Boreale *et al.* 2006] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos and G. Zavattaro. *SCC: a service centered calculus*. In Proceedings of the Third international conference on Web Services and Formal Methods, WS-FM'06, pages 38–57, Berlin, Heidelberg, 2006. Springer-Verlag.

[Bottaro & Hall 2007] André Bottaro and Richard S. Hall. *Dynamic contextual service ranking*. In Proceedings of the 6th international conference on Software composition, SC'07, pages 129–143, Berlin, Heidelberg, 2007. Springer-Verlag.

[Boulanger *et al.* 2012] Frédéric Boulanger, Ayman Dogui, Cécile Hardebolle, Christophe Jacquet, Dominique Marcadet and Iuliana Prodan. *Semantic adaptation using CCSL clock constraints*. In Proceedings of the 2011th international conference on Models in Software Engineering, MODELS'11, pages 104–118, Berlin, Heidelberg, 2012. Springer-Verlag.

[Box 1997] Don Box. Essential com. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition, 1997.

[Boyer *et al.* 2013] Fabienne Boyer, Olivier Gruber and Damien Pous. *Robust reconfigurations of component assemblies*. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 13–22, Piscataway, NJ, USA, 2013. IEEE Press.

[Bozga *et al.* 2002] Marius Bozga, Susanne Graf and Laurent Mounier. *IF-2.0: A Validation Environment for Component-Based Real-Time Systems*. In Proceedings of the 14th International Conference on Computer Aided Verification, CAV '02, pages 343–348, London, UK, UK, 2002. Springer-Verlag.

[Brennan *et al.* 2002] Robert W. Brennan, Martyn Fletcher and Douglas H. Norrie. *A Holonic Approach to Reconfiguring Real-Time Distributed Control Systems.* In Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications II-Selected Revised Papers, pages 323–335, London, UK, UK, 2002. Springer-Verlag.

[Brogi *et al.* 2006] Antonio Brogi, Carlos Canal and Ernesto Pimentel. *Software Adaptation.* In L'Objet - Special Issue on Coordination and Adaptation Techniques for Software Entities, volume 12, pages 9–31, 2006.

[Brugali *et al.* 2012] Davide Brugali, Luca Gherardi, Markus Klotzbücher and Herman Bruyninckx. *Service Component Architectures in Robotics: The SCA-Orocos Integration.* In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner and Bernhard Steffen, editors, Leveraging Applications of Formal Methods, Verification, and Validation, Communications in Computer and Information Science, pages 46–60. Springer Berlin Heidelberg, 2012.

[Bruneton *et al.* 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma and Jean-Bernard Stefani. *The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems.* Softw. Pract. Exper., vol. 36, no. 11-12, pages 1257–1284, September 2006.

[Bruni *et al.* 2006] Roberto Bruni, Hernán Melgratti and Emilio Tuosto. *Translating orc features into petri nets and the join calculus.* In Proceedings of the Third international conference on Web Services and Formal Methods, WS-FM'06, pages 123–137, Berlin, Heidelberg, 2006. Springer-Verlag.

[Bruni 2009] Roberto Bruni. *Formal Methods for Web Services.* Chapter " Calculi for Service-Oriented Computing", pages 1–41. Springer-Verlag, Berlin, Heidelberg, 2009.

[Bucci & Vicario 1995] Giacomo Bucci and Enrico Vicario. *Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets.* IEEE Trans. Softw. Eng., vol. 21, no. 12, pages 969–992, December 1995.

[Burke & Monson-Haefel 2006] Bill Burke and Richard Monson-Haefel. Enterprise javabeans 3.0 (5th edition). O'Reilly Media, Inc., 2006.

[Burns & Wellings 1994] A. Burns and A. J. Wellings. *HRT-HOOD: a structured design method for hard real-time systems.* Real-Time Syst., vol. 6, no. 1, pages 73–114, January 1994.

[Burns & Wellings 2001] A. Burns and A.J. Wellings. Real time systems and their programming languages: Ada 95, real-time java and real-time posix. International computer science series. Addison-Wesley, 2001.

[Calinescu *et al.* 2012] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska and Raffaela Mirandola. *Self-adaptive software needs quantitative verification at runtime.* Commun. ACM, vol. 55, no. 9, pages 69–77, September 2012.

[Calmant *et al.* 2012] Thomas Calmant, João Claudio Américo, Olivier Gattaz, Didier Donsez and Kiev Gama. *A dynamic and service-oriented component model for python long-lived applications.* In Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering, CBSE '12, pages 35–40, New York, NY, USA, 2012. ACM.

[Cervantes & Hall 2004] Humberto Cervantes and Richard S. Hall. *Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model.* In Proceedings of the 26th International Conference on Software Engineering, ICSE '04, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[Chaize *et al.* 1999] J.M. Chaize, A. Götz, W.D. Klotz, J. Meyer, M. Perez and E. Taurel. *TANGO - an object oriented control system based on CORBA.* In Proceedings of the 7th International Conference on Accelerator and Large Experimental Physics Control Systems, ICALEPCS '99, pages 475–479, 1999.

[Chan *et al.* 2005] S. Chan, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, , J. Schlimmer, H. Sekine, D. Walter, J. Weast, D. Whitehead and Wright D. *Devices Profile for Web Services*, May 2005.

[Cheng *et al.* 2009] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns and Jon Whittle. *Software Engineering for Self-Adaptive Systems.* Chapter " Software Engineering for Self-Adaptive Systems: A Research Roadmap", pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[Chiang *et al.* 1994] Su-Hui Chiang, Rajesh K. Mansharamani and Mary K. Vernon. *Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies.* SIGMETRICS Perform. Eval. Rev., vol. 22, no. 1, pages 33–44, May 1994.

[Christensen *et al.* 2001] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Specification v1.1*, 2001.

[Christos *et al.* 2009] Kareliotis Christos, Costas Vassilakis, Efstathios Rouvas and Panayiotis Georgiadis. *QoS-Driven Adaptation of BPEL Scenario Execution.* In Proceedings of the 2009 IEEE International Conference on Web Services, ICWS '09, pages 271–278, Washington, DC, USA, 2009. IEEE Computer Society.

[Clements 2001] Paul C. Clements. *Component-based software engineering.* Chapter " From subroutines to subsystems: component-based software development", pages 189–198. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Cooling 2003] J.E. Cooling. Software engineering for real-time systems. Addison-Wesley, 2003.

[Crnkovic *et al.* 2011] Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta and Michel R. V. Chaudron. *A Classification Framework for Software Component Models.* IEEE Trans. Softw. Eng., vol. 37, no. 5, pages 593–615, September 2011.

[Crnkovic 2001] Ivica Crnkovic. *Component-based Software Engineering - New Challenges in Software Development.* In Software Development - Software Focus, pages 127–133. John Wiley Sons, 2001.

[Crnkovic 2005] Ivica Crnkovic. *Component-based software engineering for embedded systems.* In Proceedings of the 27th international conference on Software engineering, ICSE '05, pages 712–713, New York, NY, USA, 2005. ACM.

[Cucinotta *et al.* 2009] T. Cucinotta, A. Mancina, G.F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo and F. Rusina. *A Real-Time Service-Oriented Architecture for Industrial Automation*. Industrial Informatics, IEEE Transactions on, vol. 5, no. 3, pages 267–277, 2009.

[Dai & Wang 2010] Changying Dai and Zhibin Wang. *A Flexible Extension of WSDL to Describe Non-Functional Attributes*. In e-Business and Information System Security (EBISS), 2010 2nd International Conference on, pages 1–4, 2010.

[D'Ambrogio 2006] Andrea D'Ambrogio. *A Model-driven WSDL Extension for Describing the QoS of Web Services*. In Proceedings of the IEEE International Conference on Web Services, ICWS '06, pages 789–796, Washington, DC, USA, 2006. IEEE Computer Society.

[Davis & Burns 2011] Robert I. Davis and Alan Burns. *A survey of hard real-time scheduling for multiprocessor systems*. ACM Comput. Surv., vol. 43, no. 4, pages 35:1–35:44, October 2011.

[Deacon 2005] J. Deacon. Object-oriented analysis and design: A pragmatic approach. Pearson Addison Wesley, 2005.

[Deng *et al.* 2005] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt and Aniruddha Gokhale. *DAnCE: a qos-enabled component deployment and configuration engine*. In Proceedings of the Third international working conference on Component Deployment, CD'05, pages 67–82, Berlin, Heidelberg, 2005. Springer-Verlag.

[Díaz *et al.* 2008] M. Díaz, D. Garrido, L. Llopis, F. Rus and J. M. Troya. *UM-RTCOM: An analyzable component model for real-time distributed systems*. J. Syst. Softw., vol. 81, no. 5, pages 709–726, May 2008.

[Dijkstra 1968] Edsger W. Dijkstra. *The structure of the THE-multiprogramming system*. Commun. ACM, vol. 11, no. 5, pages 341–346, May 1968.

[Dittrich *et al.* 1995] Klaus R. Dittrich, Stella Gatziu and Andreas Geppert. *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. In Proceedings of the Second International Workshop on Rules in Database Systems, RIDS '95, pages 3–20, London, UK, UK, 1995. Springer-Verlag.

[Dong *et al.* 2006] Jin Song Dong, Yang Liu, Jun Sun and Xian Zhang. *Verification of computation orchestration via timed automata*. In Proceedings of the 8th international conference on Formal Methods and Software Engineering, ICFEM'06, pages 226–245, Berlin, Heidelberg, 2006. Springer-Verlag.

[Douglass 2002] Bruce Powell Douglass. Real-time design patterns: Robust scalable architecture for real-time systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Dubey *et al.* 2011] Abhishek Dubey, Gabor Karsai and Nagabhushan Mahadevan. *A component model for hard real-time systems: CCM with ARINC-653*. Softw. Pract. Exper., vol. 41, no. 12, pages 1517–1550, November 2011.

[Emmerich 2000] Wolfgang Emmerich. *Software engineering and middleware: a roadmap*. In Proceedings of the Conference on The Future of Software Engineering, ICSE '00, pages 117–129, New York, NY, USA, 2000. ACM.

[Escoffier *et al.* 2007] C. Escoffier, R.S. Hall and Philippe Lalanda. *iPOJO: an Extensible Service-Oriented Component Framework.* In Services Computing, 2007. SCC 2007. IEEE International Conference on, pages 474–481, 2007.

[Etienne *et al.* 2006] Jean-Paul Etienne, Julien Cordry and Samia Bouzefrane. *Applying the CBSE paradigm in the real time specification for Java.* In Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, JTRES '06, pages 218–226, New York, NY, USA, 2006. ACM.

[Faulk *et al.* 1992] Stuart Faulk, John Brackett, Paul Ward and James Kirby Jr. *The CoRE Method for Real-Time Requirements.* IEEE Softw., vol. 9, no. 5, pages 22–33, September 1992.

[Feiler *et al.* 2006] P.H. Feiler, Bruce A. Lewis and S. Vestal. *The SAE Architecture Analysis x00026; Design Language (AADL) a standard for engineering performance critical systems.* In Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE, pages 1206–1211, 2006.

[Feljan *et al.* 2009] Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petricic and Ivica Crnkovic. *Classification and survey of component models.* Technical Report ISSN 1404-3041 ISRN MDH-MRTC-242/2009-1-SE, Mälardalen University, December 2009.

[Florin *et al.* 1991] G. Florin, C. Fraize and S. Natkin. *Stochastic Petri nets: Properties, applications and tools.* Microelectronics Reliability, vol. 31, no. 4, pages 669 – 697, 1991.

[Fox & Clarke 2009] Jorge Fox and Siobhán Clarke. *Exploring approaches to dynamic adaptation.* In Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction, MAI '09, pages 19–24, New York, NY, USA, 2009. ACM.

[Friedenthal *et al.* 2008] Sanford Friedenthal, Alan Moore and Rick Steiner. A practical guide to sysml: Systems modeling language. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Frølund & Koistinen 1998] Svend Frølund and Jari Koistinen. *Quality of services specification in distributed object systems design.* In Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4, COOTS'98, pages 1–1, Berkeley, CA, USA, 1998. USENIX Association.

[Frost *et al.* 2011] Christian Frost, Casper Svenning Jensen, Kasper Søe Luckow and Bent Thomsen. *WCET analysis of Java bytecode featuring common execution environments.* In Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '11, pages 30–39, New York, NY, USA, 2011. ACM.

[Gamma *et al.* 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Garlan & Perry 1995] David Garlan and Dewayne E. Perry. *Introduction to the Special Issue on Software Architecture.* IEEE Trans. Softw. Eng., vol. 21, no. 4, pages 269–274, April 1995.

[Garlan & Schmerl 2004] David Garlan and Bradley Schmerl. *Using Architectural Models at Runtime: Research Challenges.* In Flavio Oquendo, BrianC. Warboys and Ron Morrison, editors, Software Architecture, volume 3047 of *Lecture Notes in Computer Science*, pages 200–205. Springer Berlin Heidelberg, 2004.

[Garlan & Shaw 1994] David Garlan and Mary Shaw. *An Introduction to Software Architecture.* Technical report CMU/SEI-94-TR-21, Pittsburgh, PA, USA, 1994.

[Gen*et al.* 2002] Thomas Gen, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller and Chris Stich. *Components for embedded software: the PECOS approach.* In Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '02, pages 19–26, New York, NY, USA, 2002. ACM.

[Gomaa 1994] Hassan Gomaa. *Software design methods for the design of large-scale real-time systems.* J. Syst. Softw., vol. 25, no. 2, pages 127–146, May 1994.

[Gostelow & Plouffe 1978] K.P. Gostelow and W. Plouffe. An asynchronous programming language and computing machine. Technical report. University of California, 1978.

[Group 2003] IGRS Working Group. *Intelligent Grouping & Resource Sharing protocol*, 2003.

[Gui *et al.* 2008] Ning Gui, Vincenzo De Flori, Hong Sun and Chris Blondia. *A framework for adaptive real-time applications: the declarative real-time OSGi component model.* In Proceedings of the 7th workshop on Reflective and adaptive middleware, ARM '08, pages 35–40, New York, NY, USA, 2008. ACM.

[Guidi *et al.* 2006] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi and Gianluigi Zavattaro. *SOCK: a calculus for service oriented computing.* In Proceedings of the 4th international conference on Service-Oriented Computing, ICSOC'06, pages 327–338, Berlin, Heidelberg, 2006. Springer-Verlag.

[Håkansson *et al.* 2008] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson and Davor Slutej. *Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT.* In Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis, ATVA '08, pages 252–257, Berlin, Heidelberg, 2008. Springer-Verlag.

[Hanninen *et al.* 2008] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback and K.-L. Lundback. *The Rubus component model for resource constrained real-time systems.* In Industrial Embedded Systems, 2008. SIES 2008. International Symposium on, pages 177–183, 2008.

[Hansson *et al.* 2004] Hans Hansson, Mikael Akerholm, Ivica Crnkovic and Martin Torngren. *SaveCCM - A Component Model for Safety-Critical Real-Time Systems.* In Proceedings of the 30th EUROMICRO Conference, EUROMICRO '04, pages 627–635, Washington, DC, USA, 2004. IEEE Computer Society.

[Harbour *et al.* 2001] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez and J. M. Drake Moyano. *MAST: Modeling and Analysis Suite for Real Time Applications.* In Proceedings of the 13th Euromicro Conference on Real-Time Systems, ECRTS '01, pages 125–, Washington, DC, USA, 2001. IEEE Computer Society.

[Harel & Politi 1998] David Harel and Michal Politi. Modeling reactive systems with statecharts: The statemate approach. McGraw-Hill, Inc., New York, NY, USA, 1st édition, 1998.

[Harel 1987] David Harel. *Statecharts: A visual formalism for complex systems.* Sci. Comput. Program., vol. 8, no. 3, pages 231–274, June 1987.

[Hayes 2008] Brian Hayes. *Cloud computing.* Commun. ACM, vol. 51, no. 7, pages 9–11, July 2008.

[Heam *et al.* 2007] P.-C. Heam, O. Kouchnarenko and J. Voinot. *How to Handle QoS Aspects in Web Services Substitutivity Verification.* In Enabling Technologies: Infrastructure for Collaborative Enterprises, 2007. WETICE 2007. 16th IEEE International Workshops on, pages 333–338, 2007.

[Hermosillo *et al.* 2010] Gabriel Hermosillo, Lionel Seinturier and Laurence Duchien. *Creating Context-Adaptive Business Processes.* In PaulP. Maglio, Mathias Weske, Jian Yang and Marcelo Fantinato, editors, Service-Oriented Computing, volume 6470 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin Heidelberg, 2010.

[Higuera-Toledano & Wellings 2012] T. Higuera-Toledano and A.J. Wellings. Distributed, embedded and real-time java systems. Electrical engineering. Springer, 2012.

[Hissam 2005] S. Hissam. Pin component technology (v1.0) and its c interface. Technical note. Carnegie Mellon University, Software Engineering Institute, 2005.

[Hoare 1978] C. A. R. Hoare. *Communicating sequential processes.* Commun. ACM, vol. 21, no. 8, pages 666–677, August 1978.

[Hopcroft 2007] John E. Hopcroft. Introduction to automata theory, languages, and computation. Pearson Addison Wesley, 3rd édition, 2007.

[Hošek *et al.* 2010] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka and Michal Malohlava. *Comparison of component frameworks for real-time embedded systems.* In Proceedings of the 13th international conference on Component-Based Software Engineering, CBSE'10, pages 21–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hu *et al.* 2007] Jie Hu, Shruti Gorappa, Juan A. Colmenares and Raymond Klefstad. *Compadres: a lightweight component middleware framework for composing distributed real-time embedded systems with real-time Java.* In Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Middleware '07, pages 41–59, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[Huang *et al.* 2005] Jinfeng Huang, Jeroen Voeten, Oana Florescu, Piet Putten and Henk Corporaal. *Predictability in Real-Time System Development.* In Pierre Boulet, editor, Advances in Design and Specification Languages for SoCs, pages 123–139. Springer US, 2005.

[Hürsch & Lopes 1995] Walter L. Hürsch and Cristina Videira Lopes. *Separation of Concerns.* Technical report NU-CCS-95-03, Northeastern University, Boston, USA, 1995.

[IEEE & Electronics Engineers 1993] Institute of Electrical IEEE and CORPORATE Electronics Engineers Inc. Staff. Ieee standard for information technology - portable operating system interface (posix): System application program interface (api), amendment 1: Realtime extension (c language), ieee std 1003.1b-1993. IEEE Standards Office, New York, NY, USA, 1993.

[IEEE Standards Comittee 1990] IEEE Standards Comittee. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, pages 1–84, 1990.

[ISIS a] Institute for Software Integrated Systems ISIS. *The ACE ORB (TAO).* http://www.dre.vanderbilt.edu/TAO.

[ISIS b] Institute for Software Integrated Systems ISIS. *Component-Integrated ACE ORB (CIAO).* http://www.dre.vanderbilt.edu/CIAO.

[ISO/IEC/ (IEEE) 2007] ISO/IEC/ (IEEE). *ISO/IEC 42010 (IEEE Std) 1471-2000 : Systems and Software engineering - Recomended practice for architectural description of software-intensive systems*, 07 2007.

[Jackson 1975] M. A. Jackson. Principles of program design. Academic Press, Inc., Orlando, FL, USA, 1975.

[Jackson 1983] M. A Jackson. System development (prentice-hall international series in computer science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1983.

[Jaeger & Mühl 2006] Michael C. Jaeger and Gero Mühl. *Soft real-time aspects for service-oriented architectures.* In E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on, pages 5–5, 2006.

[Jaeger *et al.* 2005] Michael C. Jaeger, Gero Mühl and Sebastian Golze. *QoS-Aware composition of web services: an evaluation of selection algorithms.* In Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Part I, OTM'05, pages 646–661, Berlin, Heidelberg, 2005. Springer-Verlag.

[Jin & Nahrstedt 2004] Jingwen Jin and Klara Nahrstedt. *QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy.* IEEE MultiMedia, vol. 11, no. 3, pages 74–87, July 2004.

[Jouault *et al.* 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin and Ivan Kurtev. *ATL: A model transformation tool.* Sci. Comput. Program., vol. 72, no. 1-2, pages 31–39, June 2008.

[Kaisler 2002] Stephen H. Kaisler. Real-time languages. John Wiley Sons, Inc., 2002.

[Kalibera *et al.* 2009] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer and Jan Vitek. *CDx: a family of real-time Java benchmarks.* In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM.

[Kazhamiakin *et al.* 2006] Raman Kazhamiakin, Paritosh Pandya and Marco Pistore. *Timed Modelling and Analysis in Web Service Compositions.* In Proceedings of the First International Conference on Availability, Reliability and Security, ARES '06, pages 840–846, Washington, DC, USA, 2006. IEEE Computer Society.

[Ke *et al.* 2007] Xu Ke, Krzysztof Sierszecki and Christo Angelov. *COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems.* In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '07, pages 199–208, Washington, DC, USA, 2007. IEEE Computer Society.

[Kephart & Chess 2003] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing.* Computer, vol. 36, no. 1, pages 41–50, January 2003.

[Kiczales *et al.* 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-oriented programming.* In Mehmet Akşit and Satoshi Matsuoka, editors, ECOOP'97 — Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[Kim *et al.* 1999] Taehyoun Kim, Naehyuck Chang, Namyun Kim and Heonshik Shin. *Scheduling garbage collector for embedded real-time systems.* SIGPLAN Not., vol. 34, no. 7, pages 55–64, May 1999.

[Kim *et al.* 2009] Ji Eun Kim, O. Rogalla, S. Kramer and A. Hamann. *Extracting, specifying and predicting software system properties in component based real-time embedded software development.* In Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, pages 28–38, 2009.

[Kopetz 2011] Hermann Kopetz. Real-time systems: Design principles for distributed embedded applications. Springer Publishing Company, Incorporated, 2nd édition, 2011.

[Koudri *et al.* 2011] Ali Koudri, Arnaud Cuccuru, Sebastien Gerard and François Terrier. *Designing heterogeneous component based systems: evaluation of MARTE standard and enhancement proposal.* In Proceedings of the 14th international conference on Model driven engineering languages and systems, MODELS'11, pages 243–257, Berlin, Heidelberg, 2011. Springer-Verlag.

[Kramer & Magee 1990] Jeff Kramer and Jeff Magee. *The Evolving Philosophers Problem: Dynamic Change Management.* IEEE Trans. Softw. Eng., vol. 16, no. 11, pages 1293–1306, November 1990.

[Kübert *et al.* 2011] Roland Kübert, Georgina Gallizo, Theodoros Polychniatis, Theodora Varvarigou, Eduardo Oliveros, Stephen C. Phillips and Karsten Oberle. *Achieving Real-Time in Distributed Computing: From Grids to Clouds.* Chapter " Service Level Agreements for Real-Time Service-Oriented Infrastructures. Igi Global, 2011.

[Kwiatkowska 2007] Marta Kwiatkowska. *Quantitative verification: models, techniques and tools.* In The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers, ESEC-FSE companion '07, pages 449–458, New York, NY, USA, 2007. ACM.

[Kyriazis *et al.* 2010] Dimosthenis Kyriazis, Andreas Menychtas, Karsten Oberle, Thomas Voith, Alcatel Lucent, Michael Boniface, Eduardo Oliveros, Tommaso Cucinotta and Sören Berger. *A Real-time Service Oriented Infrastructure.* In Proceedings of the Annual International Conference on Real-time and Embedded Systems, RTES 2010, 2010.

[Labejof *et al.* 2012] Jonathan Labejof, Antoine Leger, Philippe Merle, Lionel Seinturier and Hugues Vincent. *R-MOM: A Component-Based Framework for Interoperable and Adaptive Asynchronous Middleware Systems.* In Proceedings of the 2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops, EDOCW '12, pages 204–213, Washington, DC, USA, 2012. IEEE Computer Society.

[Laliwala & Chaudhary 2008] Z. Laliwala and S. Chaudhary. *Event-driven Service-Oriented Architecture.* In Service Systems and Service Management, 2008 International Conference on, pages 1–6, 2008.

[Lambert & Power 2008] Jonathan M. Lambert and James F. Power. *Platform Independent Timing of Java Virtual Machine Bytecode Instructions.* Electron. Notes Theor. Comput. Sci., vol. 220, no. 3, pages 97–113, December 2008.

[Lankes *et al.* 2001] S. Lankes, M. Pfeiffer and T. Bemmerl. *Design and implementation of a SCI-based real-time CORBA.* In Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on, pages 23–30, 2001.

[Lapadula *et al.* 2007] Alessandro Lapadula, Rosario Pugliese and Francesco Tiezzi. *A calculus for orchestration of web services.* In Proceedings of the 16th European conference on Programming, ESOP'07, pages 33–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[Laplante & Ovaska 2011] Phillip A. Laplante and Seppo J. Ovaska. Real-time systems design and analysis: Tools for the practitioner. Wiley-IEEE Press, 4th édition, 2011.

[Lau & Wang 2007] Kung-Kiu Lau and Zheng Wang. *Software Component Models.* IEEE Trans. Softw. Eng., vol. 33, no. 10, pages 709–724, October 2007.

[Leavens *et al.* 1999] Gary T. Leavens, Albert L. Baker and Clyde Ruby. *JML: A Notation for Detailed Design.* In Bernhard Rumpe Haim Kilov and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer, 1999.

[Liu & Layland 1973] C. L. Liu and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* J. ACM, vol. 20, no. 1, pages 46–61, January 1973.

[Liu *et al.* 2006] D. Liu, X.S. Hu, M.D. Lemmon and Qiang Ling. *Firm real-time system scheduling based on a novel QoS constraint.* Computers, IEEE Transactions on, vol. 55, no. 3, pages 320–333, 2006.

[Lopez *et al.* 2006] Patricia Lopez, Julio L. Medina and Jose M. Drake. *Real-Time Modelling of Distributed Component-Based Applications.* In Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO '06, pages 92–99, Washington, DC, USA, 2006. IEEE Computer Society.

[Luo *et al.* 2008] Chenguang Luo, Shengchao Qin and Zongyan Qiu. *Verifying BPEL-Like Programs with Hoare Logic.* In Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE '08, pages 151–158, Washington, DC, USA, 2008. IEEE Computer Society.

[Magee & Kramer 1996a] Jeff Magee and Jeff Kramer. *Dynamic structure in software architectures*. In Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '96, pages 3–14, New York, NY, USA, 1996. ACM.

[Magee & Kramer 1996b] Jeff Magee and Jeff Kramer. *Self Organising System Structuring*. In Joint proceedings of the second international software architecture workshop (ISAW) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, ISAW '96, pages 35–38, New York, NY, USA, 1996. ACM Press.

[Mallet & Andre 2009] Frédéric Mallet and Charles Andre. *On the Semantics of UML/-MARTE Clock Constraints*. In Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC '09, pages 305–312, Washington, DC, USA, 2009. IEEE Computer Society.

[Mallet & de Simone 2008] Frédéric Mallet and Robert de Simone. *MARTE: a profile for RT/E systems modeling, analysis–and simulation?* In Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops, Simutools '08, pages 43:1–43:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Mallet *et al.* 2009] Frédéric Mallet, Marie-Agnès Peraldi-Frati and Charles Andre. *Marte CCSL to Execute East-ADL Timing Requirements*. In Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC '09, pages 249–253, Washington, DC, USA, 2009. IEEE Computer Society.

[Marcos *et al.* 2011] Marga Marcos, Estévez Elisabet, Christophe Jouvray and Antonio Kung. *An Approach to Use MDE in Dynamically Reconfigurable Networked Embedded SOAs*. In Proceedings of the 18th IFAC World Congress, pages 14946–14951, 2011.

[Marienfeld *et al.* 2012] Florian Marienfeld, Edzard Höfig, Michele Bezzi, Matthias Flügge, Jonas Pattberg, Gabriel Serme, AchimD. Brucker, Philip Robinson, Stephen Dawson and Wolfgang Theilmann. *Service Levels, Security, and Trust*. In Alistair Barros and Daniel Oberle, editors, Handbook of Service Description, pages 295–326. Springer US, 2012.

[Martínez *et al.* 2009a] Enrique Martínez, Maria Emilia Cambronero, Gregorio Diaz and Valentin Valero. *Design and Verification of Web Services Compositions*. In Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services, ICIW '09, pages 395–400, Washington, DC, USA, 2009. IEEE Computer Society.

[Martínez *et al.* 2009b] Enrique Martínez, Gregorio Díaz, Carmen Rosa Martínez, M. Emilia Cambronero and Valentín Valero. *Time Ordering Architecture in SCA*. In Proceedings of the 2009 conference on Techniques and Applications for Mobile Commerce: Proceedings of TAMoCo 2009, pages 117–126, Amsterdam, The Netherlands, The Netherlands, 2009. IOS Press.

[Martinez *et al.* 2010] Patricia Lopez Martinez, Cesar Cuevas and Jose M. Drake. *RT-D&C: Deployment Specification of Real-Time Component-Based Applications*. In

Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA '10, pages 147–155, Washington, DC, USA, 2010. IEEE Computer Society.

[Martínez *et al.* 2013] Patricia López Martínez, Laura Barros and José M. Drake. *Design of component-based real-time applications.* Journal of Systems and Software, vol. 86, no. 2, pages 449 – 467, 2013.

[Matsumoto 2010] S. Matsumoto. *Echonet: A Home Network Standard.* Pervasive Computing, IEEE, vol. 9, no. 3, pages 88–92, 2010.

[McGregor & Eklund 2008] Carolyn McGregor and J. Mikael Eklund. *Real-Time Service-Oriented Architectures to Support Remote Critical Care: Trends and Challenges.* In Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, COMPSAC '08, pages 1199–1204, Washington, DC, USA, 2008. IEEE Computer Society.

[McKinley *et al.* 2004] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten and Betty H. C. Cheng. *Composing Adaptive Software.* Computer, vol. 37, no. 7, pages 56–64, July 2004.

[Medvidovic 1996] Nenad Medvidovic. *ADLs and dynamic architecture changes.* Joint proceedings of the second international software architecture workshop (ISAW) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops -, pages 24–27, 1996.

[Mendes *et al.* 2009] J. Marco Mendes, Axel Bepperling, João Pinto, Paulo Leitao, Francisco Restivo and Armando W. Colombo. *Software Methodologies for the Engineering of Service-Oriented Industrial Automation: The Continuum Project.* In Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01, COMPSAC '09, pages 452–459, Washington, DC, USA, 2009. IEEE Computer Society.

[Merlin & Farber 1976] P.M. Merlin and David J. Farber. *Recoverability of Communication Protocols–Implications of a Theoretical Study.* Communications, IEEE Transactions on, vol. 24, no. 9, pages 1036–1043, 1976.

[Meyer 1997] Bertrand Meyer. Object-oriented software construction (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[Minora *et al.* 2012] Leonardo Minora, Jérémy Buisson, Flávio Oquendo and Thaís Vasconcelos Batista. *Issues of Architectural Description Languages for Handling Dynamic Reconfiguration.* In 6ème Conférence Internationale Francophone sur les Architectures Logicielles, Montpellier, France, 2012.

[Mitchell 1990] R.J. Mitchell. Managing complexity in software engineering. Iee Computing Series No 17. Peter Peregrinus Limited, 1990.

[Mok 1996] Al Mok. *Firm real-time systems.* ACM Comput. Surv., vol. 28, no. 4es, December 1996.

[Moreland 2013] James D. Moreland. *Experimental Research and Future Approach on Evaluating Service-Oriented Architecture (SOA) Challenges in a Hard Real-Time Combat System Environment.* Systems Engineering, pages n/a–n/a, 2013.

[Morin *et al.* 2009] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey and Arnor Solberg. *Models@ Run.time to Support Dynamic Adaptation.* Computer, vol. 42, no. 10, pages 44–51, October 2009.

[Morrison 1997] Michael Morrison. Presenting javabeans. Sams, Indianapolis, IN, USA, 1997.

[Moussa *et al.* 2010] Hachem Moussa, Tong Gao, I-Ling Yen, Farokh Bastani and Jun-Jang Jeng. *Toward effective service composition for real-time SOA-based systems.* Service Oriented Computing and Applications, vol. 4, no. 1, pages 17–31, 2010.

[Muhammad *et al.* 2012] W.A. Muhammad, M. Radziah and N.A.J. Dayang. *SOA4DERTS: A Service-Oriented UML profile for Distributed Embedded Real-Time Systems.* In Computers Informatics (ISCI), 2012 IEEE Symposium on, pages 64–69, 2012.

[Murata 1989] T. Murata. *Petri nets: Properties, analysis and applications.* Proceedings of the IEEE, vol. 77, no. 4, pages 541–580, 1989.

[Muskens *et al.* 2005] Johan Muskens, Michel R. V. Chaudron and Johan J. Lukkien. *Component-Based Software Development for Embedded Systems.* Chapter " A component framework for consumer electronics middleware", pages 164–184. Springer-Verlag, Berlin, Heidelberg, 2005.

[Oberle *et al.* 2013] Daniel Oberle, Alistair Barros, Uwe Kylau and Steffen Heinzl. *A unified description language for human to automated services.* Information Systems, vol. 38, no. 1, pages 155 – 181, 2013.

[Object Management Group 2006a] Object Management Group. *CORBA Component Model 4.0 Specification.* Specification Version 4.0, Object Management Group, April 2006.

[Object Management Group 2006b] Object Management Group. *Deployment and Congu-ration of Component-based Distributed Applications v4.0*, 2006.

[Object Management Group 2006c] Object Management Group. *UML Specification, Version 2.0*, 2006.

[Object Management Group 2007] Object Management Group. *Data Distribution Service v1.2*, 2007.

[Object Management Group 2008] Object Management Group. *Modeling and Analysis of Real-time and Embedded Systems, Version 1.1*, 2008.

[Object Management Group 2011] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation, Version 1.1*, 2011.

[Object Management Group 2012a] Object Management Group. *Object Constraint Language Specification, Version 2.3.1*, 2012.

[Object Management Group 2012b] Object Management Group. *Service oriented architecture Modelling Language, Version 1.0.1*, 2012.

[Ölveczky *et al.* 2010] Peter Csaba Ölveczky, Artur Boronat and José Meseguer. *Formal semantics and analysis of behavioral AADL models in real-time maude.* In Proceedings of the 12th IFIP WG 6.1 international conference and 30th IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems, FMOODS'10/FORTE'10, pages 47–62, Berlin, Heidelberg, 2010. Springer-Verlag.

[Oreizy *et al.* 2008] Peyman Oreizy, Nenad Medvidovic and Richard N. Taylor. *Runtime software adaptation: framework, approaches, and styles.* In Companion of the 30th international conference on Software engineering, ICSE Companion '08, pages 899–910, New York, NY, USA, 2008. ACM.

[Panahi *et al.* 2010] Mark Panahi, Weiran Nie and Kwei-Jay Lin. *RT-Llama: Providing Middleware Support for Real-Time SOA.* Int. J. Syst. Serv.-Oriented Eng., vol. 1, no. 1, pages 62–78, January 2010.

[Panahi *et al.* 2011] Mark Panahi, Weiran Nie and Kwei-Jay Lin. *The Design of Middleware Support for Real-Time SOA.* In Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC '11, pages 117–124, Washington, DC, USA, 2011. IEEE Computer Society.

[Papaioannou *et al.* 2006] Ioannis V. Papaioannou, Dimitrios T. Tsesmetzis, Ioanna G. Roussaki and Miltiades E. Anagnostou. *A QoS Ontology Language for Web-Services.* In Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 01, AINA '06, pages 101–106, Washington, DC, USA, 2006. IEEE Computer Society.

[Papazoglou & Heuvel 2007] MikeP. Papazoglou and Willem-Jan Heuvel. *Service oriented architectures: approaches, technologies and research issues.* The VLDB Journal, vol. 16, no. 3, pages 389–415, 2007.

[Papazoglou *et al.* 2007] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar and Frank Leymann. *Service-Oriented Computing: State of the Art and Research Challenges.* Computer, vol. 40, no. 11, pages 38–45, November 2007.

[Parnas 1972] D. L. Parnas. *On the criteria to be used in decomposing systems into modules.* Commun. ACM, vol. 15, no. 12, pages 1053–1058, December 1972.

[Parnas 1978] David L. Parnas. *Designing software for ease of extension and contraction.* In Proceedings of the 3rd international conference on Software engineering, ICSE '78, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[Parsons & Walsh 2011] P. Parsons and A. Walsh. *SOA4GDS : Evaluating the Suitability of Emerging Service-based Technologies in Ground Data Systems.* In Proceedings of the European Ground System Architecture Workshop, ESAW '11, 2011.

[Perry & Wolf 1992] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the study of software architecture.* SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, pages 40–52, October 1992.

[Pfeffer & Ungerer 2004] M. Pfeffer and T. Ungerer. *Dynamic real-time reconfiguration on a multithreaded Java-microcontroller.* In Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on, pages 86–92, 2004.

[Pitter 2008] Christof Pitter. *Time-predictable memory arbitration for a Java chip-multiprocessor*. In Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems, JTRES '08, pages 115–122, New York, NY, USA, 2008. ACM.

[Plásil *et al.* 1998] F. Plásil, D. Bálek and R. Janecek. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. In Proceedings of the International Conference on Configurable Distributed Systems, CDS '98, pages 43–, Washington, DC, USA, 1998. IEEE Computer Society.

[Plšek *et al.* 2012] Ales Plšek, Frederic Loiret and Michal Malohlava. *Component-Oriented Development for Real-Time Java*. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, Distributed, Embedded and Real-time Java Systems, pages 265–292. Springer US, 2012.

[Raibulet & Massarelli 2008] C. Raibulet and M. Massarelli. *Managing Non-functional Aspects in SOA through SLA*. In Database and Expert Systems Application, 2008. DEXA '08. 19th International Workshop on, pages 701–705, 2008.

[Raman *et al.* 2005] Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, Raymond Klefstad and Trevor Harmon. *RTZen: highly predictable, real-time java middleware for distributed and embedded systems*. In Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware, Middleware '05, pages 225–248, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[Rasche & Polze 2005] Andreas Rasche and Andreas Polze. *Dynamic Reconfiguration of Component-based Real-time Software*. In Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS '05, pages 347–354, Washington, DC, USA, 2005. IEEE Computer Society.

[Real & Crespo 2004] Jorge Real and Alfons Crespo. *Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal*. Real-Time Syst., vol. 26, no. 2, pages 161–197, March 2004.

[Rellermeyer *et al.* 2007] Jan S. Rellermeyer, Gustavo Alonso and Timothy Roscoe. *R-OSGi: distributed applications through software modularization*. In Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Middleware '07, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[Richardson & Wellings 2012] Thomas Richardson and Andy J. Wellings. *RT-OSGi: Integrating the OSGi Framework with the Real-Time Specification for Java*. In M. Teresa Higuera-Toledano and Andy J. Wellings, editors, Distributed, Embedded and Real-time Java Systems, pages 293–392. Springer US, 2012. 10.1007/978-1-4419-8158-5_12.

[Richardson *et al.* 2009] T. Richardson, A. J. Wellings, J. A. Dianes and M. Díaz. *Providing temporal isolation in the OSGi framework*. In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '09, pages 1–10, New York, NY, USA, 2009. ACM.

[Rouvoy *et al.* 2008] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen and Erlend Stav. *Composing components and services using a planning-based adaptation middleware*. In Proceedings of the 7th international conference on Software composition, SC'08, pages 52–67, Berlin, Heidelberg, 2008. Springer-Verlag.

[Samaras *et al.* 2010] I.K. Samaras, J.V. Gialelis and G.D. Hassapis. *A service oriented-based system for real time industrial applications.* In Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on, pages 1–8, 2010.

[Schlenoff *et al.* 1999] Craig Schlenoff, Michael Gruninger, Mihai Ciocoiu and Jintae Lee. *The essence of the process specification language.* Trans. Soc. Comput. Simul. Int., vol. 16, no. 4, pages 204–216, December 1999.

[Schmidt & Kuhns 2000] Douglas C. Schmidt and Fred Kuhns. *An Overview of the Real-Time CORBA Specification.* Computer, vol. 33, no. 6, pages 56–63, June 2000.

[Schmidt *et al.* 2000] Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. Pattern-oriented software architecture: Patterns for concurrent and networked objects. John Wiley & Sons, Inc., New York, NY, USA, 2nd édition, 2000.

[Schmidt 2002] Douglas C. Schmidt. *Middleware for real-time and embedded systems.* Commun. ACM, vol. 45, no. 6, pages 43–48, June 2002.

[Schoeberl & Pedersen 2006] Martin Schoeberl and Rasmus Pedersen. *WCET analysis for a Java processor.* In Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems, JTRES '06, pages 202–211, New York, NY, USA, 2006. ACM.

[Schoeberl *et al.* 2007] Martin Schoeberl, Hans Sondergaard, Bent Thomsen and Anders P. Ravn. *A Profile for Safety Critical Java.* In Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC '07, pages 94–101, Washington, DC, USA, 2007. IEEE Computer Society.

[Schoeberl *et al.* 2010] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen and Benedikt Huber. *Worst-case execution time analysis for a Java processor.* Softw. Pract. Exper., vol. 40, no. 6, pages 507–542, May 2010.

[Seinturier *et al.* 2009] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni and Jean-Bernard Stefani. *Reconfigurable SCA Applications with the FraSCAti Platform.* In Proceedings of the 2009 IEEE International Conference on Services Computing, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[Sentilles *et al.* 2008] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson and Ivica Crnković. *A Component Model for Control-Intensive Distributed Embedded Systems.* In Proceedings of the 11th International Symposium on Component-Based Software Engineering, CBSE '08, pages 310–317, Berlin, Heidelberg, 2008. Springer-Verlag.

[Shaw *et al.* 1995] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik. *Abstractions for Software Architecture and Tools to Support Them.* IEEE Trans. Softw. Eng., vol. 21, no. 4, pages 314–335, April 1995.

[Shin & Ramanathan 1994] K.G. Shin and P. Ramanathan. *Real-time computing: a new discipline of computer science and engineering.* Proceedings of the IEEE, vol. 82, no. 1, pages 6–24, 1994.

[Stachtiari *et al.* 2012] Emmanouela Stachtiari, Anakreon Mentis and Panagiotis Katsaros. *Rigorous Analysis of Service Composability by Embedding WS-BPEL into the BIP Component Framework*. 2012 IEEE 19th International Conference on Web Services, vol. 0, pages 319–326, 2012.

[Stallings 2011] William Stallings. Operating systems: Internals and design principles. Prentice Hall Press, Upper Saddle River, NJ, USA, 7th édition, 2011.

[Stankovic & Rajkumar 2004] John A. Stankovic and R. Rajkumar. *Real-Time Operating Systems*. Real-Time Syst., vol. 28, no. 2-3, pages 237–253, November 2004.

[Stankovic & Ramamritham 1990] John A. Stankovic and Krithi Ramamritham. *What is predictability for real-time systems?* Real-Time Syst., vol. 2, no. 4, pages 247–254, October 1990.

[Stankovic 1992] John A. Stankovic. *Real-time computing*. BYTE, vol. 17, no. 8, pages 155–ff., August 1992.

[Stoyenko 1992] Alexander D. Stoyenko. *The evolution and state-of-the-art of real-time languages*. Journal of Systems and Software, vol. 18, no. 1, pages 61 – 83, 1992.

[Subramonian *et al.* 2004] V. Subramonian, Guoliang Xing, C. Gill, Chenyang Lu and Ron Cytron. *Middleware specialization for memory-constrained networked embedded systems*. In Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE, pages 306–313, 2004.

[Sun *et al.* 2008] Wei Sun, Xin Zhang, Chang Jie Guo, Pei Sun and Hui Su. *Software as a Service: Configuration and Customization Perspectives*. In Congress on Services Part II, 2008. SERVICES-2. IEEE, pages 18–25, 2008.

[Szyperski 2002] Clemens Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002.

[Tan *et al.* 2011] Tian Huat Tan, Yang Liu, Jun Sun and Jin Song Dong. *Verification of orchestration systems using compositional partial order reduction*. In Proceedings of the 13th international conference on Formal methods and software engineering, ICFEM'11, pages 98–114, Berlin, Heidelberg, 2011. Springer-Verlag.

[Taylor *et al.* 2009] R.N. Taylor, N. Medvidovic and P. Oreizy. *Architectural styles for runtime software adaptation*. In Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on, pages 171–180, 2009.

[The OSGi Alliance 2012] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 5*. http://www.osgi.org/Specifications, June 2012.

[Tsai *et al.* 2006] W. T. Tsai, Yann-Hang Lee, Zhibin Cao, Yinong Chen and Bingnan Xiao. *RTSOA: Real-Time Service-Oriented Architecture*. In Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering, SOSE '06, pages 49–56, Washington, DC, USA, 2006. IEEE Computer Society.

[UPnP Forum 2000] UPnP Forum. *UPnP Device Architecture*. http://www.upnp.org/download/UPnPDA10_20000613.htm, 2000.

[van Ommering *et al.* 2000] Rob van Ommering, Frank van der Linden, Jeff Kramer and Jeff Magee. *The Koala Component Model for Consumer Electronics Software.* Computer, vol. 33, no. 3, pages 78–85, March 2000.

[Vera *et al.* 1999] James Vera, Louis Perrochon and David C. Luckham. *Event-Based Execution Architectures for Dynamic Software Systems.* In Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA), WICSA, pages 303–318, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[Verma 1999] Dinesh Verma. Supporting service level agreements on ip networks. Macmillan Technical Publishing, 1999.

[Wang *et al.* 2004] Nanbor Wang, Christopher D. Gill, Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz and Craig Rodrigues. Middleware for communications. J. Wiley & Sons, 2004.

[Wang *et al.* 2005] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati and Wei Zhao. *Real-Time Component-Based Systems.* In Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, RTAS '05, pages 428–437, Washington, DC, USA, 2005. IEEE Computer Society.

[Wang *et al.* 2010] Hanbo Wang, Xingshe Zhou, Yunwei Dong and Lei Tang. *Timing properties analysis of real-time embedded systems with AADL model using model checking.* In Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on, volume 2, pages 1019–1023, 2010.

[Ward & Mellor 1991] Paul T. Ward and Stephen J. Mellor. Structured development for real-time systems. Prentice Hall Professional Technical Reference, 1991.

[Weerawarana *et al.* 2005] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey and Donald F. Ferguson. Web services platform architecture: Soap, wsdl, ws-policy, ws-addressing, ws-bpel, ws-reliable messaging and more. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[Wehrman *et al.* 2008] Ian Wehrman, David Kitchin, William R. Cook and Jayadev Misra. *A timed semantics of Orc.* Theor. Comput. Sci., vol. 402, no. 2-3, pages 234–248, July 2008.

[Weiser 1993] M. Weiser. *Ubiquitous Computing.* Computer, vol. 26, no. 10, pages 71–72, October 1993.

[Wellings & Schoeberl 2011] Andy Wellings and Martin Schoeberl. *User-defined clocks in the real-time specification for Java.* In Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '11, pages 74–81, New York, NY, USA, 2011. ACM.

[Wermelinger 1997] Michel Wermelinger. *A Hierarchic Architecture Model for Dynamic Reconfiguration.* In Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems, PDSE '97, pages 243–, Washington, DC, USA, 1997. IEEE Computer Society.

[Wieringa 2003] R.J. Wieringa. Design methods for reactive systems: Yourdon, statemate, and the uml. The Morgan Kaufmann Series in Software Engineering and Programming. Elsevier Science, 2003.

[Wilhelm *et al.* 2008] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat and Per Stenström. *The worst-case execution-time problemoverview of methods and survey of tools.* ACM Trans. Embed. Comput. Syst., vol. 7, no. 3, pages 36:1–36:53, May 2008.

[Williams 2005] R. Williams. Real-time systems development. Elsevier Science, 2005.

[Yang *et al.* 2009] Zhibin Yang, Kai Hu, Dianfu Ma and Lei Pi. *Towards a formal semantics for the AADL behavior annex.* In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, pages 1166–1171, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[Yourdon 1989] Edward Yourdon. Modern structured analysis. Yourdon Press, Upper Saddle River, NJ, USA, 1989.

[Zhai *et al.* 2009] Yanlong Zhai, Jing Zhang and Kwei-Jay Lin. *SOA Middleware Support for Service Process Reconfiguration with End-to-End QoS Constraints.* In Web Services, 2009. ICWS 2009. IEEE International Conference on, pages 815–822, 2009.

[Zhang *et al.* 2009] Yuanfang Zhang, Christopher Gill and Chenyang Lu. *Real-Time Performance and Middleware for Multiprocessor and Multicore Linux Platforms.* In Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '09, pages 437–446, Washington, DC, USA, 2009. IEEE Computer Society.

[Zuberek 1991] W.M. Zuberek. *Timed Petri nets definitions, properties, and applications.* Microelectronics Reliability, vol. 31, no. 4, pages 627 – 644, 1991.