IMPROV: An Architecture for Reliable Execution of Dynamic Service Compositions

Felipe Pontes Guimarães

Tese apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Doutor em Ciências

Programa: Pós-Graduação em Ciências da Computação
Orientador: Prof. Dr. Daniel Macêdo Batista
Coorientador: Prof. Dra. Genaína Nunes Rodrigues

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da
CAPES/CNPq/FAPESP

São Paulo, Janeiro de 2017

Acknowledgements

I would like to start thanking God that somehow kept me going in spite of all adversities thrown in my way. Secondly, to my loving wife Thaisa and my mother Teresa who supported me all the way. Finally, a special thanks to my advisor and co-advisor, Prof. Daniel and Genaina who were always there for exchanging ideas and maturing concepts and lead me all the way up to this point.

Abstract

SOBRENOME, A. B. C. **IMPROV: An architecture for distributed and reliable execution of dynamically built service choreographies**. 2017. 88 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2010.

Adaptivity is a capability that enables a system to choose amongst various alternatives to satisfy or maintain the satisfaction of certain requirements. The criteria of requirements satisfaction could be pragmatic and context-dependent. Contextual Goal Models (CGM) capture the power of context on banning or allowing certain alternatives to reach requirements (goals) and also deciding the quality of those alternatives with regards to certain quality measures (softgoals). It is used to depict facets of the decision making strategy and rationale of an adaptive system at the preliminary level of requirements. In this thesis we argue the case for *pragmatic requirements* and extend the CGM with additional constructs to capture them and allow their analysis. We also develop an automated analysis which aids the planning and scheduling of tasks execution to meet pragmatic goals. Moreover, we evaluate our modelling and analysis regarding correctness and performance. Such an evaluation showed the applicability of the approach and its usefulness in aiding sensible decisions. It has also shown its capability to do so in a time short enough to suit run-time adaptation decision making.

Keywords: Service Composition, Fault Tolerance, Context Aware.

Contents

List of Abreviations **ix**

List de Symbols **xi**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Motivação 1

1.1.1 Leading example - Assisted Living 2

2 Theoretical Background **5**

2.1 Web Services 5

2.1.1 Integration of Legacy Software 5

2.1.2 Service Compositions 6

2.2 Fault-Tolerance for web service compositions 7

2.3 Fault-Tolerance for web services *in-the-wild* 9

2.4 Self-Adaptive and Context-aware Systems 10

2.4.1 Goal Models 10

3 Pragmatic Runtime Goal Model - PRCGM **13**

3.1 Conceptualizing Pragmatism in Requirements 13

3.2 Achievability of Pragmatic Goals 15

3.3 Meta-Model 16

3.4 Goal Annotations 17

3.5 Task Fault-Tolerance Annotations 19

3.6 Quality Constraints 19

3.7 Goal Interpretation and Achievement Criteria 19

3.7.1 Goal Achievability Function and QoS-Constrained Planning Algorithm 20

4 IMPROV Architecture **25**

4.1 Using a Pragmatic GM to model actor behaviour 26

4.1.1 Requirements for Composition of Unmodified Web Services 27

4.2 MAPE cycle 29

4.2.1 Monitoring 30

4.2.2 Analysis . . . 31
4.2.3 Planning . . . 32
4.2.4 Execution . . . 32
4.3 IMPROV choreography agents . . . 33
4.3.1 Communication Architecture and WS Handling . . . 33
4.3.2 Communication Architecture and WS Handling . . . 34
4.3.3 Agent Interactions . . . 35
4.3.4 Internal Agent Architecture . . . 36
4.3.5 Internal Elements Interactions . . . 37
4.4 Dynamic Choreography Execution . . . 39
4.4.1 Tasks Execution Process . . . 40
4.4.2 Task Sequencing and Inter-Services Communication . . . 41
4.4.3 Actors Execution Coordination . . . 41
4.5 Fault Tolerance . . . 42
4.5.1 Fault Categorization . . . 42
4.5.2 Task-level Fault-Tolerance . . . 43
4.5.3 Actors' Fault-tolerance mechanism . . . 44
4.6 Compositions Lifecycle and Evolution . . . 45
4.7 Project Decisions . . . 45

5 Validation 49
5.1 Pragmatic Model and Achievability Algorithm Evaluation . . . 49
5.1.1 GQM Validation Plan . . . 49
5.1.2 Experiment Setup . . . 49
5.1.3 Goal 1: PPA's runtime usage capability in the MPERS case study . . . 50
5.1.4 Goal 2: PPA's runtime usage capability for larger models . . . 51
5.1.5 Goal 3: Algorithm's pinpointing unachievable context sets capability . . . 52
5.1.6 Discussion of the results . . . 52
5.1.7 Threats to validity . . . 55
5.2 Fault-Tolerance Mechanism effectiveness . . . 55
5.2.1 Goal 1: Enabling Service Choreographies . . . 56
5.2.2 Goal 2: Increase the composition's reliability with several FTTs . . . 58
5.2.3 Adaptation to different contexts . . . 60

6 Conclusions and Future Work 63

Bibliography 65

Index 68

List of Abreviations

List of Symbols

List of Figures

1.1 A goal model for dispatching an ambulance to assist a patient 2

2.1 Three possible approaches for integrating legacy software into an agent-based architecture (adapted from [Gen97]) 6
2.2 A Pragmatic GM for responding to emergencies in an assisted living environment (adapted from [MAR14]) 12

3.1 G_{loc} graphical representation as a pragmatic goal 15
3.2 Metamodel for the Pragmatic Goal Model with QoS-constrained planning 16
3.3 The goal-annotated Pragmatic Goal for MPERS 18
3.4 Workflow output for context [$c4$, $c5$, $c6$, $c7$, $c10$, $c11$] 22
3.5 MPERS model with tasks chosen by the QoS-Constrained Planning Algorithm algorithm highlighted in blue 22

4.1 Visão macro do processo de tratamento de requisições pela arquitetura IMPROV . . 26
4.2 MAPE cycle implemented by the agents and involved 30
4.3 Esquema de cooperação entre os agentes da coreografia. 34
4.4 Esquema de cooperação entre os agentes da coreografia. 35
4.5 Arquitetura Interna dos Agentes 36
4.6 MAPE cycle implemented by the agents and involved 38
4.7 Laço de execução do workflow 40
4.8 Fluxo para realização tolerante a falhas de tarefas 41
4.9 Processo para identificação e recuperação de agentes inacessíveis ou indisponíveis . . 44
4.10 Ciclo de vida de uma composição sensível a contexto 46

5.1 Assertions to verify the generated plan's adherence to the annotations' semantics . . 51
5.2 Algorithm's scalability over the model size, in number of nodes, and context amount 53
5.3 Time spent to run PPA algorithm for all the possible context sets (average execution time over 10 different models) 54
5.4 Goal 1 evaluation graphs 57
5.5 Comparison between observed faults and average response time for fail-stop scenarios with and without each of the built-in FTTs 59
5.6 Comparison between observed faults and average response time for faulty response scenarios with and without NVP FTT 60
5.7 Response time in a scenario with changes observed in the service pool 61

List of Tables

1.1 Involved agents for each context combination 3

2.1 Failures in non-FT compositions 7
2.2 Failures in FT compositions 7
2.3 Individual WS reliability aided by FTTs 8
2.4 Goal annotations syntax and meaning [DBHM13] 11

3.1 Goal annotations and respective evaluated QoS Functions) 17

4.1 IMPROV's Considered Faults Categorization 43

5.1 GQM devised plan 50
5.2 GQM devised plan 56
5.3 Observed errors and reliability under each of the considered contexts 61

Chapter 1

Introduction

Nowadays, the usage of web services as building-blocks for larger applications and the integration of different applications is already wide-spread in the industry. The BPEL language and execution engines allows for using and integrating web services into highly complex and specialized applications in a predetermined way.

However, with the advent of wearable computing, mobile devices and ubiquitous computing, the computational environment now transcends from a traditional computer-based system into an ever-present and ever-changing system with elements entering and leaving the vicinities of an actor many times over. In such a scenario, the service-oriented architecture also presents itself as a very interesting alternative to grasp and collaborate with adjacent resources to achieve greater goals, given its ability to describe functionalities over a registry via WSDL files. This means that the SOA architecture can play a major role in supporting self-adaptive systems, allowing them to collaborate in possibly unplanned manners in order to achieve a goal.

In this sense, we introduce a novel way to describe actors' roles. By combining state-of-the-art requirements engineering techniques to depict systems' behaviours and an underlying architecture to convert those into a service choreography, we enable dynamic, unforeseen, distributed and fault-tolerant collaboration between actors

1.1 Motivação

Nowadays, the usage of web services as building-blocks for larger applications and the integration of different applications is already wide-spread in the industry. The BPEL language and execution engines allows for using and integrating web services into highly complex and specialized applications in a predetermined way. The almost omnipresence of computational services and resources made possible for information to run freely to the point of the arrival of concepts such as wearable computing, mobile devices and ubiquitous computing and the Internet of Things (IoT). This means that the computational environment now transcends from a traditional computer-based system into an ever-present and ever-changing system with elements entering and leaving the vicinities of an actor many times over.

In this new context, systems integration brings about a whole new set of challenges and opportunities with computational systems overcoming the computer systems' frontiers and being widely spread as a part of people's daily routines. This means that the elements of such systems are ever-more subject to an infinite variable environment, which may impact the system's functionalities.

Capturing such variability is a novell challenge for requirement engineers and several models have been proposed to grasp this variability. But capturing the variability is not enough if we cannot integrate the resources available and this is where the SOA architecture comes in handy. SOA provides the means for system interoperation without the need for modifications in the system's inner working.

In such a scenario, the service-oriented architecture also presents itself as a very interesting alternative to grasp and collaborate with adjacent resources to achieve greater goals, given its ability

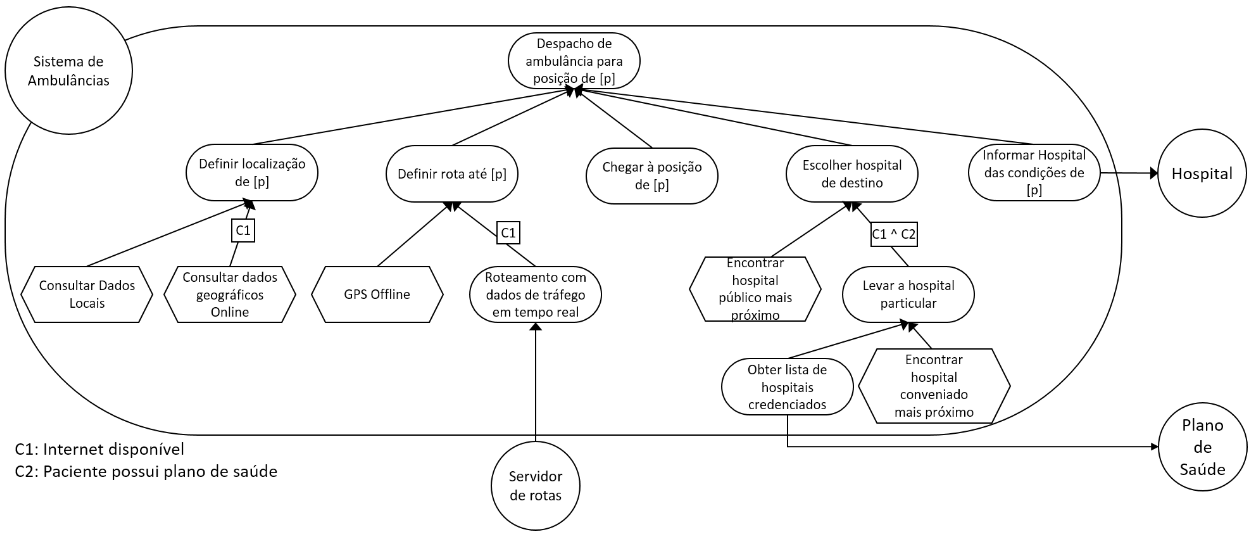

Sistema de Ambulâncias
Despacho de ambulância para posição de [p]
Definir localização de [p]
Definir rota até [p]
Chegar à posição de [p]
Escolher hospital de destino
Informar Hospital das condições de [p]
Hospital
C1
C1
C1 ^ C2
Consultar Dados Locais
Consultar dados geográficos Online
GPS Offline
Roteamento com dados de tráfego em tempo real
Encontrar hospital público mais próximo
Levar a hospital particular
Obter lista de hospitais credenciados
Encontrar hospital conveniado mais próximo
Plano de Saúde
Servidor de rotas
C1: Internet disponível
C2: Paciente possui plano de saúde

Figure 1.1: *A goal model for dispatching an ambulance to assist a patient*

to describe functionalities over a registry via WSDL files. This means that the SOA architecture can play a major role in supporting self-adaptive systems, allowing them to collaborate in possibly unplanned manners in order to achieve a goal.

In this sense, we introduce a novel way to describe actors' roles. By combining state-of-the-art requirements engineering techniques to depict systems' behaviours and an underlying architecture to convert those into a service choreography, we enable dynamic, unforeseen, distributed and fault-tolerant collaboration between actors

1.1.1 Leading example - Assisted Living

A possible use case scenario comes from the assisted living environment. Imagine a system for early emergency response for elderly people. This system would need to interact with several different actors (mobile device, hospital, ambulance dispatch, health-care insurance, among others). However, the manner through which such actors may interact will depend on the context they belong to. For instance, the actor "health-care insurance" would only be involved in the scenario if the patient is himself insured; the ambulance actor will be expected to arrive at the scene faster whenever a patient is suffering from, let's say, a cardiac arrhythmia and so on.

Let's take the example presented in Figure 1.1. The ambulance dispatch process to provide a patient with assistance starts up by defining the actual location of the patient, either through the already stored data or through geolocation data gathered from the patient's smartphone. Then, it must trace a route from the hospital to the patient, get to the patient's location, choose the hospital to take the patient in and finally report the patient's condition to the selected hospital so that it can prepare itself for receiving the patient.

However, take notice of all the contexts involved in this workflow. The amulance would depend on a mobile connection to access the internet that may not be available in remote locations. The "unavailable connection" context can impact the applicable alternatives making it impossible to look up the patient's location on an online geographic data source or to monitor real-time traffic. On another front, taking the patient to a nearer private hospital may be faster but it can only be done if the patient is insured, otherwise it is necessary to deliver the patient at a public hospital.

Another aspect to be considered is that the very same tasks may be implemented by very different services. Let's consider the "Online geographic data search" task: it can be provided both by Google (Google Maps) and by Microsoft (Bing Maps) and any provider can be used to achieve the goal. Such multiplicity of available providers allows us to implement fault-tolerance techniques that will enhance the overall quality of the composition.

The challenge however arises from the consequent variability of such scenarios. This rather small

Active Contexts	**Routing Server**	**Health-care Insurance**	**Hospital**
C1 ^ C2	✓	✓	✓
¬ C1 ^ C2			✓
C1 ^ ¬ C2	✓		✓
¬ C1 ^ ¬ C2			✓

Table 1.1: *Involved agents for each context combination*

scenario already contains two contexts and four involved actors. Also, it already presents:

1. topology and involved actors that depend directly on the current context;
2. Several environments and several distinct administrative domains and;
3. the possibility of using the multiplicity of implementing services to provide higher levels of provided quality.

Item (1) may be perceived because of the possible interactions between the ambulance actor and the other three agents (Routing Service, Hospital, Health-care Insurance) but, as illustrated in Table 1.1, only under certain contexts; Item (2) is due to the fact that most likely the ambulance dispatch, the health-care insurance, the routing server and the hospital are independently managed systems interacting among themselves but, despite pursuing a common goal, may not allow for external entities to overcome the limits imposed by their administrators.

Chapter 2

Theoretical Background

2.1 Web Services

The service-oriented computing (SOC) paradigm uses services to support the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications. Services are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways [PTDL07]. They are meant to be used as building blocks for rapid, low-cost, yet secure applications [WB10]. In [Pap03], some of the main characteristics of SOC are presented: its services must be invokable through standardized technologies (*technology neutral*), must not require knowledge about internal structures or conventions at the server side(*loosely coupled*) and have accessible information so that the invocation may happen irrespective of the service location (*location transparency*).

Services may be spread over several departments or organizations [LR12] and possibly provided *as-is*. Therefore we can divide them into two categories: the first category are the *in-house* developed services. This category includes all services that are developed by the same team that will actually use them. Services belonging to this category are easily adjustable and can be tweaked accordingly to arising needs in the business. The other category are services which are available for usage but not under the user's administrative control. These services can be referred to as web services (WS) *in-the-wild*. The concept of a web service *in-the-wild* tries to imply that they belong to a dynamic web service ecosystem[BDB05]. Important characteristics of services that fall within this category is that they are not prone to any adjustment and that their availability cannot be taken for granted, given that changes in their individual administrative domain, and therefore unpredictable, may impact their level of service or even take them offline.

2.1.1 Integration of Legacy Software

In [Gen97], the author points out three basic approaches to dealing with the integration of legacy software into an agent-based architecture: transducers, wrappers and rewrite.

In the transducer approach (Figure 2.1a), the agent mediates the existing programs and the other agents. The transducer agent receives requests from the other agents and translates them into the legacy software's native communication language. The main advantages of this approach is that it requires no knowledge of the underlying program except that of its communication behaviour.

A second approach consists of the implementation of a wrapper software that, in essence, injects code into the legacy software to allow it to communicate directly. This approach, illustrated in Figure 2.1b, assumes that the wrapper can directly examine and modify the data structures of the original software. It has the benefit of greater performance in comparison to the transducer due to the lower serial communication.

The third and most drastic approach is the complete rewriting of the legacy code, thus enabling even further performance gains. However, this is also the most expensive alternative due to the workload of understanding and re-implementing the application from scratch.

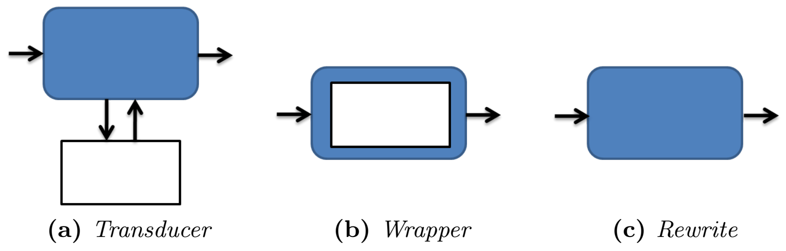


Figure 2.1: *Three possible approaches for integrating legacy software into an agent-based architecture (adapted from [Gen97])*

Although the SOA architecture is not agent-based, the same approaches can be used for exposing a legacy application as a web service.

2.1.2 Service Compositions

A service composition provides higher-level functionalities by consuming lower-level services. It may utilize any number of in-house developed and WS *in-the-wild* to achieve such functionalities. A composition that involves services *in-the-wild* cannot require any intrusive approach neither demand of the providing services any specific semantics . Much of the existing work [BDH+12, Dob06, FD02, LFCL03, MBHJ+11] does not consider this. This assumption may render them unsuitable for several real-world scenarios in which the WS provider is not under the user's control.

Services Orchestrations and Choreographies

There are two main ways of defining and performing a service composition: orchestrations and choreographies [Pel03]. In an orchestration, the whole composition is controlled by an entity referred to as the orchestrator. This entity coordinates and mediates all the interactions between the involved services. On the other hand, in a choreography the services do not have any kind of centralized control [BWR09] and rely on peer-to-peer (P2P) communication. In this scenario, each involved service or entity has the knowledge of how to perform its operations, with whom to interact and when to do it. This sequence of activities and interactions is commonly referred to as the `role` which the entity is playing out. Due to the lower need of communication and the decentralization of the data transfers, choreographies may achieve higher performance than orchestrations [GKB+12].

Reliability of service compositions:

One of the issues with the composition of services has to do with the reliability of the composed service. It depends completely on the reliability of the consumed web services, which may themselves be error-prone or even become unavailable.

Several fault-tolerance techniques have been proposed, each having a different impact on the observed reliability and performance of a service composition. In this Section, we study the impact for each of the considered FT techniques.

Since we are dealing with web services and their intrinsic encapsulation and loose coupling, the failure types prone to consideration are limited by the SOAP protocol's observable events and different failures may fall within the same categorization. For instance, an erroneous response may be due to corruption over the transmission or malicious components, but it is not possible to say which without actually inspecting the service and/or the network themselves. The failures that may be observed in non fault-tolerant compositions are depicted in Table 2.1. The considered failures within this environment are: (1) Request not replied - timeout after a service invocation; (2) incorrect reply - an incorrect response is received and; (3) service offline - a service that is no longer available and its WSDL interface is no longer accessible. Following the failure classification from Avižienis et al. [ALRL04], these failures are then categorized according to the component in

Table 2.1: *Failures in non-FT compositions*

Failure	Comp.	Domain	Detectab.	Conseq.
Request not replied	WS	Time	yes	Medium
Incorrect Reply	WS	Content	no	Major
Service Offline	WS	Time	yes	Catastrophic

Table 2.2: *Failures in FT compositions*

Failure	Comp.	Domain	Detectab.	Conseq.
Request not replied	WS	Time	yes	Medium
Incorrect Reply	WS	Content	yes	Major
Regular WS Offline	WS	Time	yes	Minor
Unique WS Offline	WS	Time	yes	Catastrophic

which they occur, the domain of the failure (time or content), their detection capability and finally the consequences of their occurrence (minor, medium, major or catastrophic)[1].

However, when we turn to compositions in fault-tolerant environments, the multiplicity of services makes it possible to identify and tackle with incorrect responses or to circumvent a service that has gone offline by replacing it with another candidate to play its role. As such, failures categorization falls into what is presented in Table 2.2.

The overall reliability of a service composition in non-FT environments can be estimated as the probability that all the tasks (service invocations) are completed, given the individual reliability of the underlying service providing such functionality. The dependencies are not considered because we define a successful invocation as the whole process of sending the request and receiving its reply.

Should we consider a generic service composition as a mathematical problem we may, without loss of generality, consider the the overall composition reliability - R_c - as the probability that all tasks performed by each of the i services, by the means of inv_{ws_i} invocations with R_{ws_i} chance of success, are correctly executed. This is given by Equation 2.1.

$$R_c = \prod_{i=1}^{n} (R_{\mathrm{ws}_i} * inv_{\mathrm{ws}_i}) \tag{2.1}$$

2.2 Fault-Tolerance for web service compositions

Several fault-tolerance techniques already exist. Zheng et al. [ZL10] uses four main techniques in the SOC context: retry, recovery block (RB), N-Version Programming (NVP) and active. Retry sends a new request to the same resource, hoping that the fault was transient. Recovery block retries the request in an standby service, until successful or all resources were attempted. The NVP technique sends the request to an even number of resources and determines the final result by majority voting. The active FTT invokes all services in parallel and selects the first returned value as the final result.

Fault-tolerance for web services has been largely discussed over the last years [BDH+12, Dob06, FD02, LV07, LFCL03, MBHJ+11, MGMS11, MRD08, ZL10]. Each of the cited references presents a slightly different approach. In [BDH+12], the authors propose the active replication of several BPEL-engines aided by a collection of input and output proxies collocated with the web services and BPEL engines; [Dob06] investigates the feasibility of using WS-BPEL as an implementation technique for isolated fault tolerant Web services without actually dealing with the service composition issue; [LV07] proposes a centralized architecture in which a central application server deals with proxy services which then deal with several alternative web services; [LFCL03] uses dynamically created services and replications, with supporting infrastructural components such as fault-detectors, to

[1]citação?

Retry	$R_{Retry} = 1 - (1 - R_{\mathrm{WS}})^{tries}$
Active	$R_{Active} = 1 - \prod_{i=1}^{n}(1 - R_{\mathrm{WS}_i})$
Recovery Block	$R_{RB} = 1 - \prod_{i=1}^{n}(1 - R_{\mathrm{WS}_i})$

Table 2.3: *Individual WS reliability aided by FTTs*

provide higher reliability to a service; [MBHJ+11] uses aspect-orientation programming (AOP) to intercept sent and received WS invocations for WS implemented in Java ® to evaluate the reliability and eventually act upon it; [MGMS11] describes a framework for self-architecting service-oriented systems that uses FT as a means of attend QoS constraints but does not focus on how to do it; [MRD08] creates architectural components within a JBoss application server with a BPEL processor so that the BPEL's service invocations are intercepted through AOP, its context is modified and the SOAP call is finally made invoking either the original WS or an alternative one. [ZL10] uses an approach similar to the one used by [BDH+12] in the sense of creating coordinators for service communities to enhance the overall reliability.

In an FT environment, the R_{ws_i} factor need to take into consideration the gains in terms of reliability due to the FTTs being used. This may also add parameters to it, such as the maximum allowed amount of tries for the Retry FTT, the amount of replicas used for Active FTT and the amount of candidate services for RB.

The expressions for evaluating the R_{ws_i} factor when using the FTTs are described in Table ??. These formulae model the expected behavior for a given service when using one of these techniques. The task will succeed unless all of the attempts fail. For Retry, the probability that any given try will fail is given by $1 - R_{ws}$. The probability that all tries will fail is then given by $(1 - R_{ws})^{tries}$. Finally, the probability of success can be calculated as $1 - (1 - R_{ws})^{tries}$. The reliability for Active is similar but instead of always using the same reliability R_{ws} it will use the different reliabilities for all n services being used (R_{ws_i}). Since it will succeed unless all replicas fail, the final reliability can be calculated as 1 minus the product of all i terms in the form $(1 - R_{ws_i})$. The Recovery Block Reliability calculation is similar since it also places n attempts on n different services.

Note that the above FTTs (Retry, Active and Recovery Block) do not account for content failures, only time-domain failures. To deal with failures in the content domain, one may use the NVP technique whose reliability is given by Equation 2.2.

$$R_{\mathrm{NVP}} = 1 - \sum_{i=res/2+1}^{res} R_{\mathrm{WS}_i} \tag{2.2}$$

Though somewhat similar to the expressions presented in [ZL10], these expressions differ from those because Zheng evaluated the failure rate f while we evaluate the observed reliability R_c. Both Zheng's as well as our equations consider that an invocation is successful as long as one of all the posed requests is replied.

Finally, we model the composition as a whole as a Kubat failure model. This model considers the case of a software composed of M modules designed for K different tasks. Each task may require several modules and the same module can be used for different tasks. Since we are dealing with service-oriented systems, the choreography agents may be perceived as the modules M, and the tasks are the provided functionalities described in the composition's WSDL. For simplicity, we will assume that a composition provides a single functionality.

Similarly to the work [GCRB13], the architecture of this model is given by a DTMC in which, after an agent i, with α_i reliability, is invoked to perform task k, there is a probability $p_{ij}(k)$ that the agent j will be invoked after the execution of k after its execution. The sojourn time during the execution of agent i has the probability distribution function $g_i(k, t)$.

Thus, the probability that no failure occurs while agent i is executing task k is given by:

$$R_i(k) = \int_0^{\infty} e^{-\alpha_i t} g_i(k,t)dt \tag{2.3}$$

The expected number of visits in module i by task k, denoted by $a_i(k)$ can be obtained by solving

$$a_i(k) = q_i(k) + \sum_{j=1}^{M} a_j(k)p_{ji}(k) \tag{2.4}$$

The probability that no failure will occur while executing task k is given by

$$R(k) = \prod_{i=1}^{M} [R_i(k)]^{a_i(k)} \tag{2.5}$$

And the overall composition failure rate can be calculated as $\lambda_s = \sum_{k=1}^{K} r_k[1 - R(k)]$ where r_k is the arrival rate of task k.

By combining these equations with Equation 2.1, it is possible to formulate the overall composition reliability by replacing the R_{WS_i} with the observed reliability for the service using the chosen FTT. Thus it is possible to evaluate, *a priori*, the expected reliability for the composition and choose a suitable FTT for each choreography agent so that the composition's reliability QoS requirement is met.

2.3 Fault-Tolerance for web services *in-the-wild*

When dealing with fault-tolerance for services *in-the-wild*, it is essential to emphasize that SOC differs from computational-intensive applications. It is usual for a service's semantics to overcome computational borders, *i.e.*, a service invocation may not only consume computational power but also trigger real-world effects. For instance, a credit card service invocation may incur in transferring an amount of money between parties, a delivery service may incur in some goods being transported across the country and so on.

Much of the available research regarding web service fault-tolerance neglect to acknowledge this characteristic and consider that only computational resources are consumed by a service invocation. As an example, the FT-SOAP fault-tolerance mechanism proposed in Liang et al. [LFCL03] always retries a failed request on a different resource until it succeeds or until all the resources have failed. While perfectly reasonable for a credit card service, for a large set of unreliable arithmetical services it may be far less efficient than simply sending the request in parallel to all of them.

The idea of using an architecture in which the interaction with the components is mediated by an specialized entity (proxy) is not new and has been used for Multi-Agent Systems back in 2002 [FD02] as well as for web services, as seen on the last paragraph [BDH+12, Dob06, LV07, LFCL03, MBHJ+11, MRD08]. However, some proposals require that additional software is installed either in the client's or the server's end [LFCL03, MBHJ+11, MRD08], some are exclusive to a programming language [MBHJ+11, MRD08] or deal only with BPEL orchestrations [BDH+12, Dob06, MGMS11], some depend on an application server that becomes a single point of failure [LV07], and most do not provide multiple fault-tolerance techniques in order to accommodate real-world services' semantics [BDH+12, FD02, MBHJ+11, MRD08]. Most notably, none of the approaches enable the composition of generic web services into a choreography, since they do not enable the P2P communication that characterizes it.

2.4 Self-Adaptive and Context-aware Systems

Self-adaptive systems are computational systems designed to enjoy a degree of flexibility in meeting their requirements and maintaining them. Adaptivity requires variability so that the system can choose amongst alternatives and optimize certain performance indicators. In requirements engineering literature, a main-stream model to capture and analyse such variability is Goal-Model (GM) [YM98]. It provides the requirements (goals) for which the system is designed and the various possible ways to reach those goals (alternatives) and their quality (soft-goals). It also allows breaking down the system to a set of autonomous entities (actors) who can also decide adaptively how to depend on each others to reach requirements.

Adaptivity is triggered by contextual factors which could be internal, e.g. errors and availability of computational resources, or external, e.g., newly available services and packages, physical and social environment of the user, their skills and computing device. In [ADG10], traditional goal models [BPG^{+}04, CKM02, Yu11] were extended to capture the notion of context and its relation with requirements. The proposed Contextual Goal Model (CGM) treats context as an adaptation driver which can help filtering the space of applicable alternatives to reach goals and dependencies between actors and deciding the quality of such alternatives and dependencies with regards to certain quality measures (softgoals). The Runtime Goal Models proposed in Dalpiaz et al. [DBHM13] elaborate on specifying the possible valid alternatives to reach goals and the possible sequences for that achievement.

2.4.1 Goal Models

Goal-Models (GM) are well established requirements engineering tools to depict and break-down systems using socio-technical concepts [YM98]. In other words, it provides the goals for which the system should be designed and the various possible ways to reach those goals. However, as pointed out in [ADG10] and [DBHM13] two aspects of real-life cannot be captured in the traditional goal model [BPG^{+}04, CKM02, Yu11]: the notion of contexts [ADG10], the determination whether a task sequencing is valid within the model and the exploration of alternative system configurations to restore the system to a valid state [DBHM13].

Goal-Models (GM) are well established requirements engineering tools to depict and break-down systems using socio-technical concepts [YM98]. In other words, it provides the goals for which the system should be designed and the various possible ways to reach those goals. However, as pointed out in [ADG10] and [DBHM13] two aspects of real-life cannot be captured in the traditional goal model [BPG^{+}04, CKM02, Yu11]: the notion of contexts [ADG10], the determination whether a task sequencing is valid within the model and the exploration of alternative system configurations to restore the system to a valid state [DBHM13].

Runtime Goal-Model

In [DBHM13] Dalpiaz et al. present a framework for bridging the gap between design-time goal-models and runtime behaviour. Starting with an early requirements model representing stakeholder goals (*Design-time Goal Model* or *DGM*) they refine it with additional behavioural detail about how goals are to be achieved. Specifically, they add constraints on valid orderings for pursuing sub-goals thereby creating a *Runtime Goal-Model* (*RGM*).

Dalpiaz argues that DGMs are in some ways too abstract to enable runtime monitoring of requirements. In this sense, they pose some questions:(1) Is observed behaviour compliant with the system specification?; (2) How does system behaviour relate to the fulfilment of stakeholders (root) goals? and; (3) Can the system switch to an alternative behaviour to restore fulfilment?

Furthermore, they state that such questions may be projected over a time frame, similarly to the Awareness Requirements proposed by [SLRM11]. In this sense, one would also be able to question: (4) What is the percentage of success for a given goal during the last month? and ; (5) What is the trend for failure of a given goal in the last week?

Annotation	**Meaning**
$R_1; R_2$	Sequential execution of R_1 then R_2
$R_1\#R_2$	Parallel execution of R_1, R_2 or both
$R^{k\#}$	Parallel execution of k instances of R, with $k > 0$
R^{k+}	Sequential execution of k iterations of R, with $k > 0$
$try(R_1)?R_2 : R_3$	Attempt R_1. If R_1 is fulfilled, R_2; otherwise R_3
$R_1\|R_2$	Alternative execution of R_1 or R_2, not both
$opt(R)$	R is optional
skip	do nothing;

Table 2.4: *Goal annotations syntax and meaning [DBHM13]*

Dalpiaz's work on runtime goal models focus on the first two questions: providing algorithms for deciding whether a given execution trace is compliant with the specification and with the relationship of the system behaviour with the stakeholders' goals.

To do so, Dalpiaz proposed the goal annotations depicted in Table 2.4. In addition to the traditional AND- and OR- decomposition semantics, Dalpiaz's goal annotations guarantee it is fulfilled in accordance with the runtime rules associated with R for sequential and parallel executions.

Suppose a goal R is decomposed into R_1 and R_2. While the AND/OR-decomposition rules define if all of R subgoals must be achieved, the goal annotations define the order in which they may be achieved. As such, an $(R_1; R_2)$ annotation requires the fulfillment of R_1 prior to R_2, while an $(R_1\#R_2)$ does not impose any order on the fulfillment of R_1 and R_2 allowing for parallel execution of both.

For the other rules, the fulfillment of the node follows runtime rules. If R must be repeated k times, k instances of R must be executed serially (R^{k+}) or in parallel $(R^{k\#})$. The ternary try $(try(R_1)?R_2 : R_3)$ is interpreted as the attempt of R_1 followed by R_2 (if R_1 succeeds) or R_3 (should R_1 fail). Finally, alternative rule $(R_1|R_2)$ is interpreted of either R_1 or R_2, but not both.

In the remainder of this paper, we have tackled the concept brought about by Dalpiaz's third question: the search, within the acceptable system configurations, of some set up to allow it to restore fulfilment of the root goal.

Contextual Goal-Model

Traditional goal modelling tools and techniques depict the intentionality behind the goals and the possible several different ways of achieving those goals. However, they do not explicitly state the triggers or drivers that motivate such adaptation. Contextual goal-models seeks to collaborate with requirements engineering by adding the application's context as an explicit adaptation driver. Under some enviromental variables, a set of alternatives may be rendered impractible, inadequate or, on the other hand, may be made obligatory. In this sense, context itself and its eventual changes become a very important adaptation trigger, that respond to internal (e.g. errors, computational resources availability) or external (e.g. newly found services, weather conditions, time of day) changes.

The Contextual Goal Model, proposed in [ADG10], is meant to capture the relation between requirements and their dynamic environment. Context can guide adaptation and support the decision in the goals to activate and filter the space of alternative strategies - subgoal, task or delegation - which can be applied to achieve activated goals. Context can also have an effect on the quality of those alternatives and this is captured through the notion of contextual contribution from goals and tasks to softgoals.

Context is the description that concretize relevant factors in the system's environment, *i.e.*, the surrounding in which it operates [FS01]. In goal modeling terminology, context is a specification of a partial state of the world relevant to an actor's goals [ADG10]. Actors are social entities, or software representing them, in an organization. Actors exist to have and be responsible of achieving goals and they have degree of flexibility how to achieve them. A context that affects that choice could be the time of the day, a weather condition, patient's chronic cardiac problem, etc.

Fig. 2.2 present a CGM that depicts the requirements of a Mobile Personal Emergency Response System meant to respond to emergency situations for people in an assisted living environment. The root goal is "respond to emergency", which is performed by the actor `Mobile Personal Emergency Response`. The root goal is divided into 4 subgoals: "emergency is detected", "[p] is notified about emergency", "central receives [p] info" and "medical care reaches [p]" ([p] stands for "patient"). Such goals are then further decomposed, within the boundary of an actor, to finally reach executable tasks or delegations to other actors. A task is a process performed by the actor and a delegation is the act of passing a goal on to another actor that can perform it.

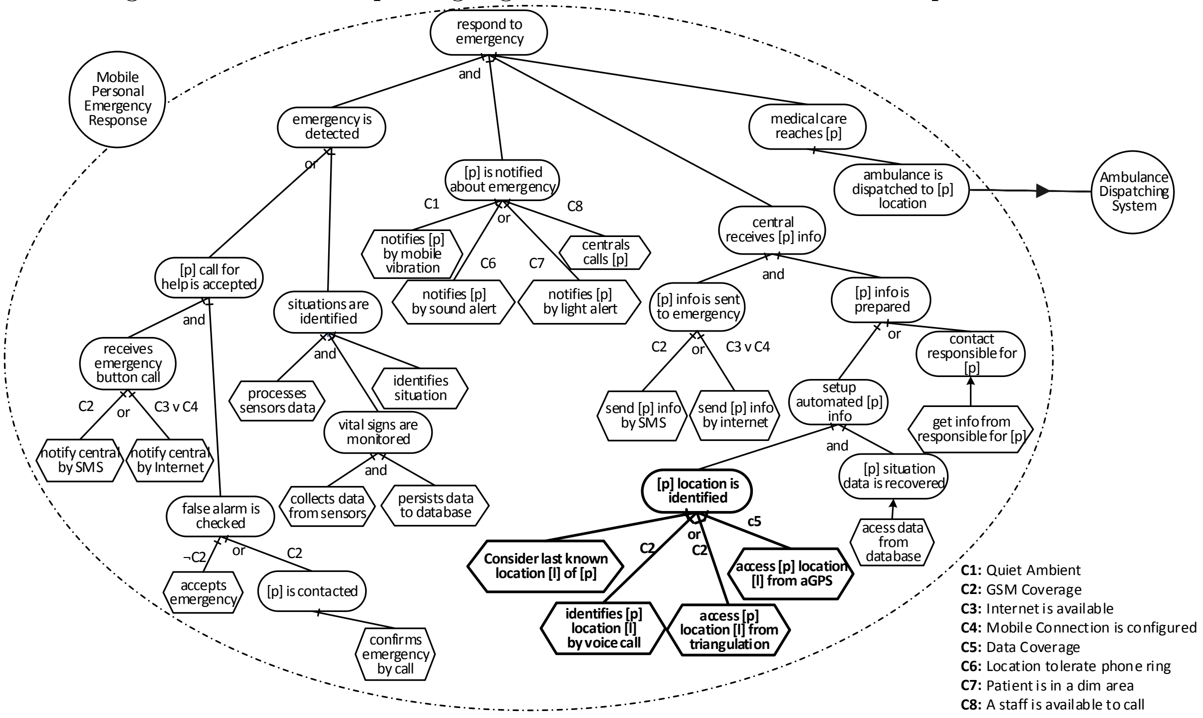

respond to emergency
and
Mobile Personal Emergency Response
emergency is detected
or
medical care reaches [p]
ambulance is dispatched to [p] location
Ambulance Dispatching System
[p] is notified about emergency
C1
or
C8
notifies [p] by mobile vibration
C6
C7
centrals calls [p]
notifies [p] by sound alert
notifies [p] by light alert
central receives [p] info
and
[p] call for help is accepted
and
situations are identified
and
[p] info is sent to emergency
[p] info is prepared
or
contact responsible for [p]
receives emergency button call
C2
or
C3 v C4
processes sensors data
identifies situation
vital signs are monitored
and
C2
or
C3 v C4
send [p] info by SMS
send [p] info by internet
setup automated [p] info
and
get info from responsible for [p]
notify central by SMS
notify central by Internet
collects data from sensors
persists data to database
[p] situation data is recovered
[p] location is identified
false alarm is checked
¬C2
or
C2
c5
or
C2
C2
acess data from database
Consider last known location [l] of [p]
access [p] location [l] from aGPS
accepts emergency
[p] is contacted
confirms emergency by call
identifies [p] location [l] by voice call
access [p] location [l] from triangulation
C1: Quiet Ambient
C2: GSM Coverage
C3: Internet is available
C4: Mobile Connection is configured
C5: Data Coverage
C6: Location tolerate phone ring
C7: Patient is in a dim area
C8: A staff is available to call

Figure 2.2: *A Pragmatic GM for responding to emergencies in an assisted living environment (adapted from [MAR14])*

It is important to highlight that not all the subgoals, delegations and/or tasks are always applicable. Some of them depend on certain contexts whether they hold.

Chapter 3

Pragmatic Runtime Goal Model - PRCGM

As we have introduced in Chapter 2, in a goal model the relations between sub-nodes and parent nodes are supposed to be causal. Achieving one (OR-Decomposition) or all (AND- Decomposition) of the subgoals is seen as a satisfactory precondition for achieving the parent goal. However, in [PGNRMBA15] we argued that the achievement of goals would in certain cases need to be seen in a pragmatic fashion and not as a direct causation of the achievement of other goals or the execution of certain tasks. The decision whether a goal is achieved could be context-dependent. Our work advocated the need for a more flexible definition of goals to accommodate their contextual interpretation and achievement measures.

Take the example of Fig. 2.2: in general, let's consider that the ambulance may take up to ten minutes to arrive. For a patient with a minor discomfort it can take its time and arrive in 20 minutes without suffering any penalty. On the other hand, if the patient is having a heart attack, one cannot say the goal was achieved. In these situations, the mere fact that the ambulance has reached the event site is not sufficient to render the goal as achieved. If the time between the need for assistance was detected and the ambulance's actual arrival is unacceptably long, the goal would be considered failed in spite of the arrival. In other words, the delivered level of quality may not be a separate part from the boolean answer of whether a goal is achieved or not, but is actually an integral part of it.

The Pragmatic Goal Model (PGM) was initially presented in [PGNRMBA15], with the concepts of pragmatic requirements and pragmatic goals. In this paper we have stated that the achievement of a goal may depend on the provided level of quality and that both the provided and the required level of quality may be dependent on the context. Later, we have enhanced the PGM into what we called the Pragmatic Runtime Goal Model - PRGM presented in [GRAB17]. The PRCGM extends the PGM model with Dalpiaz's Runtime Annotations in order to allow the PRCGM to specify the system's runtime behavior and engineer, on the fly, a system configuration that would satisfy the user needs under the current context.

In the following Sections we are going to dig deeper into the need for pragmatic goals and the PRCGM model. We will also introduce some newly added modifications to the model so that it can also be used to decide on the fault-tolerance technique to be used. This model is itself one of the contributions of this work to requirements engineering *state-of-the-art* by combining the CGM [ADG10], PGM [PGNRMBA15] and RGM [DBHM13], while adding to them the ability to explicitly include fault-tolerance specifications. It is also the conceptual basis on which the IMPROV architecture implementation lies.

3.1 Conceptualizing Pragmatism in Requirements

We will now present the need and the concept of a pragmatic goal as a modelling tool to fully capture the nuances of a real-world scenario. Next, we concretize these concepts and present the

Pragmatic Goal Model or PGM. The PGM is an extension to the CGM and was initially proposed in [PGNRMBA15] to depict the goal's pragmatic aspect and the goal's context-dependent interpretation. In [GRAB17], it has been improved to depict also the task sequencing and to allow the engineering of an execution plan. Mainly, we have enhanced the PGM model with goal annotations for specification of valid decomposition orderings, context-dependent goal interpretations, the expected delivered QoS for tasks - which are also context-dependent, in order to reason about the achievability of goals and to engineer a suitable execution plan to achieve them.

In Goal Modelling, the relations between sub-nodes and parent nodes are supposed to be causal. Achieving one (OR-Decomposition) or all (AND- Decomposition) of the subgoals is seen as a satisfactory precondition for achieving the parent goal. However, in [PGNRMBA15] we argued that the achievement of goals would in certain cases need to be seen in a pragmatic fashion and not as a direct causation of the achievement of other goals or the execution of certain tasks. The decision whether a goal is achieved could be context-dependent. Our work advocated the need for a more flexible definition of goals to accommodate their contextual interpretation and achievement measures.

It is paramount to note that the representation of the perceived quality of a goal as a softgoal is innately different from the pragmatism of the goal achievement. From a pragmatic point of view, a proposition is true if it works satisfactorily. Similarly, a pragmatic goal is achieved if a satisfactorily high level of service is provided. It is not a matter of achieving it with higher or lower quality, but achieving it at all. Also, the satisfactory quality level for the pragmatic goal may even depend on the context of the system at runtime not just on the model itself - making it more strict or even more relaxed when some contexts apply.

Take the example of Fig. 2.2: in general, let's consider that the ambulance may take up to ten minutes to arrive. For a patient with a minor discomfort it can take its time and arrive in 20 minutes without suffering any penalty. On the other hand, if the patient is having a heart attack, one cannot say the goal was achieved. In these situations, the delivered level of quality may not be a separate part from the boolean answer of whether a goal is achieved or not, but an integral part of it.

A pragmatic goal model, just like a regular goal-model, describes the means to achieve it but it also describes the interpretation of such achievement. This interpretation, which depicts the goal's pragmatic fulfilment criteria, can be expressed as a set of quality constraints (QCs). Unlike softgoals, which are a special type of goal with no clear-cut satisfaction criteria[SLRM11], these QCs are mandatory and crisp, therefore quantifiable, constraints needed for the fulfilment of a goal and an inherent part of its definition. For instance, take goal "[p] location is identified" from Fig. 2.2($\mathbf{G}_{loc}$ for short): it could be defined as "in order to reach $\mathbf{G}_{loc}$, the location must be identified within an error radius of maximum 500 meters and in less than 2 minutes".

But then again, this would not suffice as a radius of 500m and 2 minutes might be an over-relaxed condition for patients under critical conditions.This brings into light another aspect to be taken into account for the pragmatic requirements: the fact that the interpretation for the achievement of a goal is itself context-dependent. We consider that there is a default condition for achieving a goal. However, for specific contexts, we could relax or further strengthen the condition which interprets whether a goal is achieved. We propose that the contextual QCs on the achievement of a goal should be captured together with the other effects of context in the CGM. One advantage of capturing the pragmatic goals within the CGM is to enable reasoning on the possibility of achieving a goal under the current context and QCs. We differentiate these interpretations in the sense that a relaxation condition is not mandatory but a requirement that further strengthen the QC must necessarily be considered.

In the previous example, a QC of getting a location within 500m in less than 2 minutes is a default constraint. However, if the user has access to mobile data (context C5) then a much preciser location can be obtained from the GPS. Under these circumstances, a lock within 500m may seem like an over-relaxed constraint. For a patient with cardiac arrhythmia (context C10), a more strict QC is needed. Suppose that the system has to ensure that an ambulance reaches the patient's home within 5 minutes. Possibly, in this case, a faster but less precise location would be better suited.

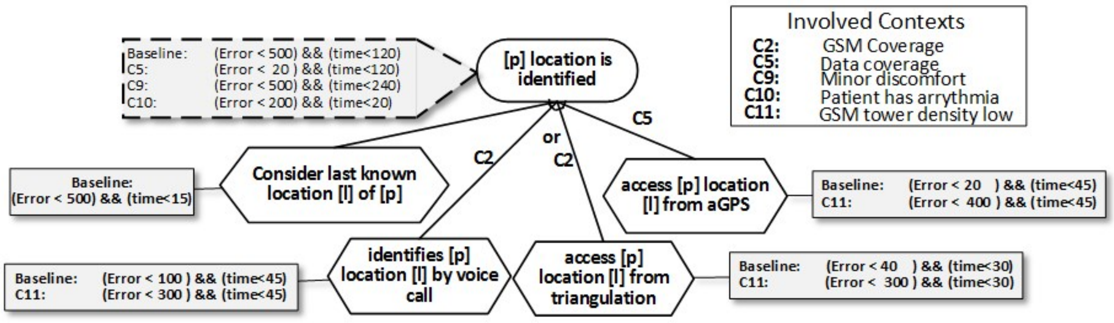

Involved Contexts
C2: GSM Coverage
C5: Data coverage
C9: Minor discomfort
C10: Patient has arrythmia
C11: GSM tower density low
Baseline: (Error < 500) && (time<120)
C5: (Error < 20) && (time<120)
C9: (Error < 500) && (time<240)
C10: (Error < 200) && (time<20)
[p] location is identified
or
C2
C2
C5
Consider last known location [l] of [p]
Baseline: (Error < 500) && (time<15)
access [p] location [l] from aGPS
Baseline: (Error < 20) && (time<45)
C11: (Error < 400) && (time<45)
identifies [p] location [l] by voice call
Baseline: (Error < 100) && (time<45)
C11: (Error < 300) && (time<45)
access [p] location [l] from triangulation
Baseline: (Error < 40) && (time<30)
C11: (Error < 300) && (time<30)

Figure 3.1: *G_{loc} graphical representation as a pragmatic goal*

The requirements for a minor discomfort (context C9) are also more flexible than those for an arrhythmia (C10). In the three specific contexts, the interpretation must differ from the original baseline.

Figure 3.1 sums up $\mathbf{G}_{loc}$'s interpretation criteria and presents it as a box connected to the goal itself. However, not only the goal's interpretation may vary according to the context but also the delivered quality of service (QoS) levels. When executed in different contexts, a task may provide different QoS levels, represented by the boxes linked to the tasks in Figure 3.1. Furthermore, provided that $\mathbf{G}_{loc}$ is annotated with `G1;G2;G3;G4`, the consequent task sequencing - derived from the goal's annotation - will too impact on the goal's perceived QoS. For instance, for $\mathbf{G}_{loc}$ this means that, although no individual task has a time constraint of over 120 seconds, unless context C9 is active at least one of such tasks will obligatorily not be chosen so that the total time stays under 120 seconds.

The set of QCs that represent the pragmatic aspects of a goal and the variable delivered QoS levels extend the traditional CGM into a Pragmatic CGM and allow for the reasoning over the achievability of a goal under a certain context.

3.2 Achievability of Pragmatic Goals

Pragmatic goals can only be achieved if their provided QoS levels comply with the QCs specified for them, both of which are context-dependent. This means that we extended the basic effect of context on a CGM to cover success and achievement criteria. Such expressiveness enables further analysis for a key adaptation decision: how to, under the current context, pick a task set and arrange them into a valid task sequence in order to reach our goals while respecting its interpretation when the goals' interpretations, the space of applicable alternatives to reach them and the QoS levels provided by the tasks are all context-dependent?

Such adaptation decision may also lead down a dire path. It is inevitable for one to wonder: "what to do when such a sequence cannot be found? ". Situations where it may not be possible to meet the goal's interpretation QoS standards through any of the applicable sub goals, tasks and/or delegations are also considered in our approach. This is a rather likely scenario considering that the tasks deliver not a static but a context-dependent QoS level. Whenever there is no possible task sequencing able to deliver the goal's required QoS we classify the goal as unachievable. This is expected to happen and the proper way of dealing with such goals is explained in the reasoning part.

Let's get back to the example in Figure 2.2: if we consider the goal $\mathbf{G}_{loc}$ and the contexts impacting on its interpretation, the conclusion is that under a certain context the system may not be able to determine the patient's location with the required precision. This, in practice, does not mean doing nothing. The motivation of doing this analysis and deeming the goal unachievable is simply to have such knowledge beforehand. At design-time, this allows the goal model designer to add new strategies for reaching the goal with that particular context in mind, thus covering a larger range of contexts. At runtime, this early conclusion would lead to search for a better variant at a higher-level goal by choosing another task sequencing, which is able to deliver the required quality standard. Therefore, our analysis is both meant for design-time - reasoning to evaluate and validate the comprehensiveness of the solution - and for runtime - devising the suitable planning strategy to reach goals in a specific context under a certain set of quality constraints.

3.3 Meta-Model

Figure 3.2 presents a conceptual model of our extension to the CGM. For the focus of this paper, the CGM could be seen as an aggregation of `Requirements`. `Requirements` may be specialized into several types: `Tasks`, `Delegations` and `Goals`. A `Delegation` represents when the `Goal` or `Task` (*dependum*) is pursued not by the current but by an external actor (*delegatee*). `Tasks` are performed by the actor in order to achieve a goal. `Tasks` may report the expected delivered quality for each metric through the `providedQuality` method.

A `Goal` is a useful abstraction to represent stakeholders' needs and expectations and they offer a very intuitive way to elicit and analyse requirements[ADG10]. Goals have a refinements set which define the `Requirements` (subgoals, tasks and/or delegations) that can be used for achieving it as well as a method to distinguish AND- from OR-compositions. Each goal is annotated with a `Goal Annotation` which describes the required behaviour of the goal's requirement instances. Goal annotations are thoroughly discussed in Section 3.4. Context is also strongly related to goals, for it changes the current goals of a stakeholder and the possible ways to satisfy them[ADG10]. In this sense, the goal's decomposition may vary according to the context and this is modelled in the meta-model as the method `getApplicableRefinements`.

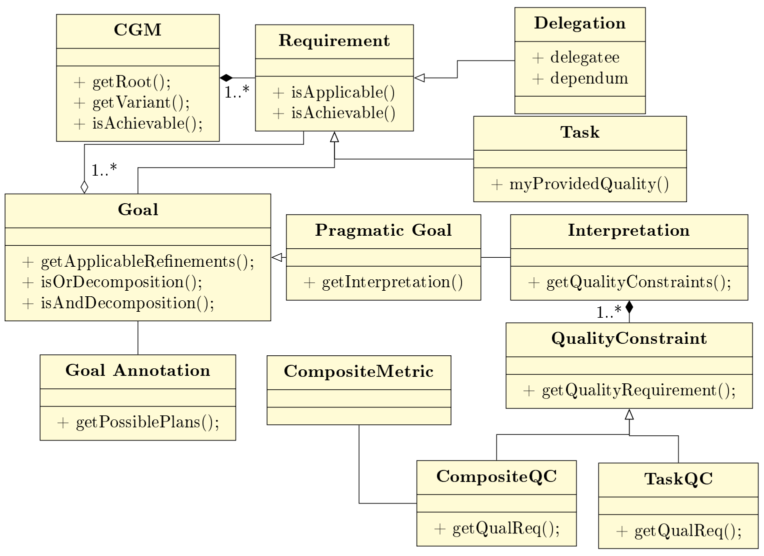


Figure 3.2: *Metamodel for the Pragmatic Goal Model with QoS-constrained planning*

`Pragmatic Goals` extend the `Goal` concept with the `Interpretation` of its achievement. A goal's `Interpretation` is an abstract concept that has the function of cross-referencing a context and the appropriate `QualityConstraints` for that given context. The `Pragmatic Goal` is said to be achieved if, and only if, such requirements are met. Otherwise, the goal's delivered QoS is considered inappropriate and the goal is not achieved regardless of achieving one or all of its `Requirements`.

The `Quality Constraints` are expressed in terms of the `applicableContext` in which it holds, the metric that should be considered, the threshold which is a numerical value that represents the scalar value for such metric and the comparison which defines whether the threshold described is the maximum allowed value or the minimum. For instance, to state a quality requirement of at most 250ms for the execution time when context $C1$ holds, the metric would be "ms", `threshold` would be 250, condition would be "Less or Equal" and `applicableContext` would be $C1$. `Quality constraints` may have task (`TaskQC`) or workflow scope (`CompositeQC`). `TaskQCs` impose a restriction on the applicable tasks whereas `CompositeQCs` restrictions apply to the workflow as a whole. The composition rules for the cal-

Annotation		**QoS Function Time**	**QoS Function Reliability**
$R_1; R_2$		$t(R_1) + t(R_2)$	$rel(R_1) * rel(R_2)$
$R_1 \# R_2$		$max(t(R_1), t(R_2))$	$rel(R_1) * rel(R_2)$
$R^{k\#}$		$t(R)$	$rel(R)^k$
R^{k+}		$t(R) * k$	$rel(R)^k$
$R_1 \vert R_2$	R_1 was chosen:	$t(R_1)$	$rel(R_1)$
	R_2 was chosen:	$t(R_2)$	$rel(R_2)$
$try(R)?R_1 : R_2$	R is achievable:	$t(r) + t(R_1)$	$rel(R) * rel(R_1)$
	R is unachievable:	$t(r) + t(R_2)$	$(1 - rel(R) * rel(R_2)$

Table 3.1: *Goal annotations and respective evaluated QoS Functions)*

culation of Composite QCs are expressed via the `CompositeMetric` objects, which are able to estimate the resulting metric depending on the goal annotation being considered (Section 3.4). The difference between these two kinds of quality constraints (QCs) is further discussed in Section 3.6 whereas their usage in the QoS-Constrained Planning Algorithm algorithm is explained in Section **??**.

Every `Requirement` inherits the `isAchievable` method. This method can be used either by the final users or by the higher level goals to define whether a particular goal can be achieved for a given quality requirement under the current context. While this is obviously necessary for the root goal, as the ultimate objective, we also allow certain subgoals to be defined as pragmatic, *i.e.*, they may also have their own predefined interpretation. In principle, actors should be able to impose further constraints on the criteria for achieving any goal within their boundary. The importance of the subgoals quality requirement becomes obvious when dealing with delegation of goals where the external actor may have itself a different, more relaxed or more strict, quality constraint, not necessarily compatible with what the delegator intends.

In this model, the goal annotations, the expectation of delivered quality by the tasks, the quality constraints for the goals, subgoals or delegations and the metrics compositional rules are added to the traditional CGM. This is meant to be done by the requirements expert or the domain experts due to the need for specialized knowledge to define such metrics.

3.4 Goal Annotations

As previously discussed in Section 2.4.1, goal annotations were first defined in [DBHM13]. They are intended to express the expected behaviour of a goal, regarding the manners in which its refinements may be sequenced. Following the meaning, allowed set of traces for goal annotations and annotation creation rules, the interleaved and sequential annotations are applicable to AND-Goals whereas try and alternative annotations are applicable to OR-Goals. The remaining annotations are unary therefore indifferently applicable to AND and OR decompositions.

In our case, goals are to be fulfilled by the system by taking into account (1) contextual specifications and (2) the possible valid configurations for goals and tasks, following RGM runtime rules, extended from [DBHM13]. Table 3.1 synthesizes a textual description of each RGM rule and its corresponding semantics.

In our model, the goal annotations are used to determine, together with the goal decomposition type, the different possible approaches for dealing with the goal's underlying requirements. It is the goal's annotation that first envisions the possible plans for achieving the goal. It is also the goal annotation, together with the Composite Quality metrics' QoS functions, that estimate the overall plan quality measures for each possibility. Only then, does the `Qos Constrained Planning` algorithm (see Section **??**) compare the possible plans and chooses an appropriate one.

As depicted in Table 3.1, the runtime annotations we have used are:

Sequential ($R_1; R_2$) Serial execution of two refinements, with composed time equals to the sum of their individual execution times ($t(R_1) + t(R_2)$) and reliability equals to the chance of successfully executing R_1 and R_2 $(rel(R_1) * rel(R_2))$;

Interleaved ($R_1\#R_2$) Refinement executions performed in parallel, with composed time equals to that of the most time consuming one $(MAX(t(R_1), t(R_2)))$and reliability equals to the chance of successfully executing R_1 and R_2 successfully $(rel(R_1) * rel(R_2))$;

Sequential iterations (R^{k+}) serial execution of k instances of the plan R, with composed time equal to the time spent for a single execution multiplied by the amount of repetitions $(t(R) * k)$ and reliability equals to the chance of successfully executing all the k instances $(rel(R)^k)$;

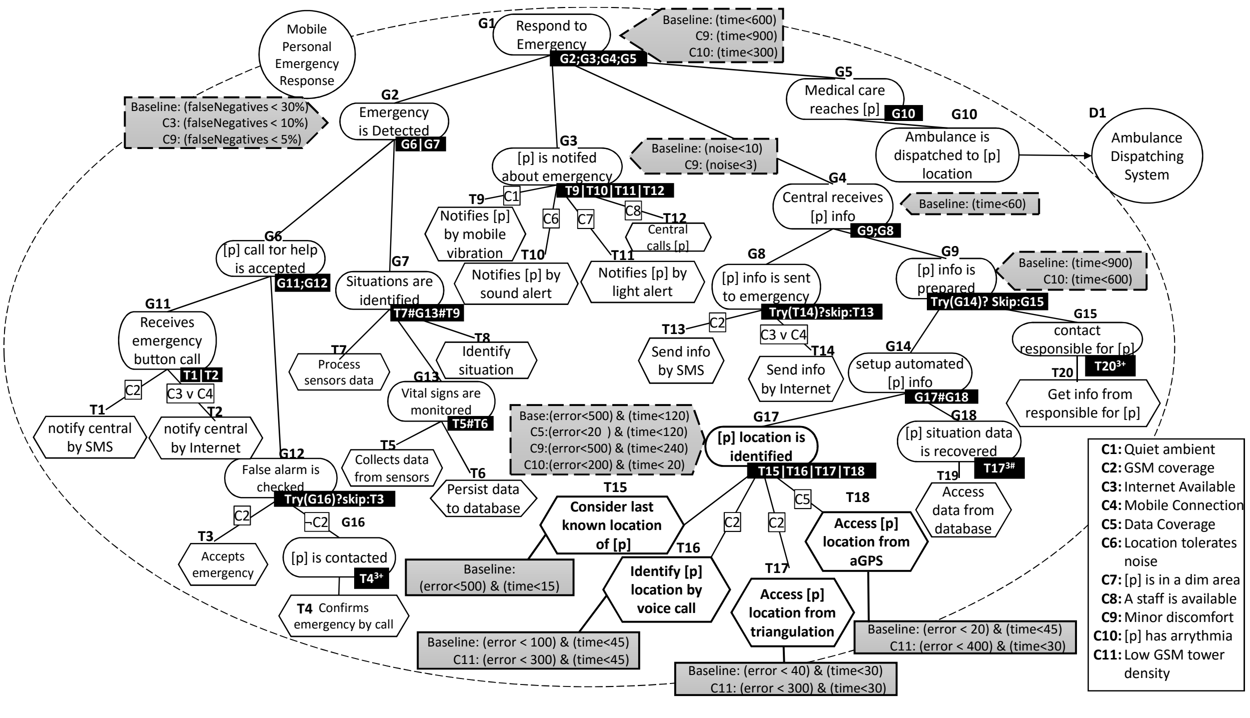
Interleaved iterations ($R^{k\#}$) Parallel execution of k instances of the plan R, with total time estimate equivalent to that of a single task $(t(R))$, since the model allows them all to be executed in parallel and reliability equals to the chance of successfully executing all the k instances $(rel(R)^k)$;

Alternative ($R_1|R_2$) Alternative execution of plan R_1 or R_2, with the same time and reliability QoS levels as that of the chosen alternative;

Try ($try(R)?R_1 : R_2$) Annotation that depends on the achievability of R. If it is achievable, performs R and R_1 sequentially, otherwise, performs R - which will fail - and R_2. The time QoS is then evaluated similarly to the sequential annotation and the reliability becomes the chance of successfully executing R and R_1 (R achievable) or just R_2 (since R will fail per design).

In Dalpiaz's original work, there was also the $opt(G)$ annotation which meant G was optional and applicable only under certain circumstances. Accordingly, we map the original annotation as a context-dependency on the goal itself.

Figure 3.3 presents the Pragmatic GM from Figure 2.2 but now with annotated goals (the goal annotations are represented as the black boxes on top of the goals). Because of such annotations, the possible behaviours of the system are now thoroughly specified and the dependencies between tasks can now be represented and taken into consideration when planning the system adaptation.



Mobile Personal Emergency Response
G1 Respond to Emergency
G2;G3;G4;G5
Baseline: (time<600) C9: (time<900) C10: (time<300)
Baseline: (falseNegatives < 30%) C3: (falseNegatives < 10%) C9: (falseNegatives < 5%)
G2 Emergency is Detected
G6|G7
G3 [p] is notifed about emergency
T9|T10|T11|T12
Baseline: (noise<10) C9: (noise<3)
G5 Medical care reaches [p]
G10
G10 Ambulance is dispatched to [p] location
D1 Ambulance Dispatching System
G4 Central receives [p] info
G9;G8
Baseline: (time<60)
T9 Notifies [p] by mobile vibration
C1
C6
C7
C8
T10 Notifies [p] by sound alert
T11 Notifies [p] by light alert
T12 Central calls [p]
G6 [p] call for help is accepted
G11;G12
G7 Situations are identified
T7#G13#T9
G8 [p] info is sent to emergency
Try(T14)?skip:T13
G9 [p] info is prepared
Try(G14)? Skip:G15
Baseline: (time<900) C10: (time<600)
G11 Receives emergency button call
T1|T2
C2
C3 v C4
T1 notify central by SMS
T2 notify central by Internet
T7 Process sensors data
T8 Identify situation
G13 Vital signs are monitored
T5#T6
T13 Send info by SMS
T14 Send info by Internet
G14 setup automated [p] info
G17#G18
G15 contact responsible for [p]
T20 Get info from responsible for [p]
T20^3+
Base:(error<500) & (time<120) C5:(error<20) & (time<120) C9:(error<500) & (time<240) C10:(error<200) & (time< 20)
G17 [p] location is identified
T15|T16|T17|T18
G18 [p] situation data is recovered
T17^3#
T19 Access data from database
G12 False alarm is checked
Try(G16)?skip:T3
T5 Collects data from sensors
T6 Persist data to database
T3 Accepts emergency
¬C2
G16 [p] is contacted
T4^3+
T4 Confirms emergency by call
T15 Consider last known location of [p]
Baseline: (error<500) & (time<15)
T16 Identify [p] location by voice call
Baseline: (error < 100) & (time<45) C11: (error < 300) & (time<45)
C5
T17 Access [p] location from triangulation
Baseline: (error < 40) & (time<30) C11: (error < 300) & (time<30)
T18 Access [p] location from aGPS
Baseline: (error < 20) & (time<45) C11: (error < 400) & (time<30)
C1: Quiet ambient
C2: GSM coverage
C3: Internet Available
C4: Mobile Connection
C5: Data Coverage
C6: Location tolerates noise
C7: [p] is in a dim area
C8: A staff is available
C9: Minor discomfort
C10: [p] has arrythmia
C11: Low GSM tower density

Figure 3.3: *The goal-annotated Pragmatic Goal for MPERS*

3.5 Task Fault-Tolerance Annotations

3.6 Quality Constraints

To enhance goals with context-dependent interpretation, we must revisit the classical concept of achievability of a goal to fit the nature of Pragmatic CGMs. On top of the basic context effect on a CGM, we enable a higher model expressiveness through the goal interpretation.

The goal interpretation, as stated in Section 3.1, is a set of mandatory and crisp, therefore quantifiable, quality constraints needed for the fulfilment of a goal. Since we are dealing with a goal annotated model and considering the tasks interdependencies are accurately modelled, we must also further divide the quality constraints into two classes based on their scope: individual task-wise constraints and composite workflow-wise constraints. Their concepts are as follows:

Task-wise Constraints
Many constraints deal with the individual quality provided by the task itself. These constraints can include measures like “level of precision”, “margin of error”, “noise” or “false negatives”. Any task that can isolatedly reach the level of quality stated in such constraints can be used, regardless of the remainder task choices. All constraints from [PGNRMBA15] were task-wise constraints, given that no sequencing was made. These constraints can be added to the model directly, simply by giving them a unique label.

Workflow-wise Constraints
Differently from the *task-wise constraints*, workflow-wise or composite constraints are applicable to the quality level of the overall chosen tasks sequencing that satisfy the root goal. In this sense, these constraints have a direct dependency on the way the tasks cooperate to fulfil a certain goal.

Provided that the QoS level composition is heavily dependent on the type of QoS being considered, each composite metric must implement a `CompositeMetric` object (from our meta model in Figure 3.2) defining the compositional rules for the calculation of CompositeQCs for each goal annotation. For this paper, the implemented `CompositeMetrics` were `Time` and `Reliability` but the approach as well as the developed algorithm are able to deal with any composite metric, given a `CompositeMetrics` object implementing all the compositional rules.

3.7 Goal Interpretation and Achievement Criteria

As stated previously in Section 3.1, a pragmatic goal’s achievement criteria is based on the user’s expected QoS levels for a given scenario or context.

These criteria are specified via a set of quality-constraints which are also context-dependent. Thus, a goal can be interpreted as achieved if, and only if, its provided QoS is within the QCs acceptable scope. This also means that the goal’s decompositions are not all equivalent. The applicability of each alternative decomposition now depends on both the applicable contexts but also on the level of QoS they are able to deliver. In turn, the levels they deliver also depend on the delivered QoS levels of their decompositions and so on.

The expressiveness power added by the dynamic, context-aware, task- or workflow-wise quality constraints enable richer adaptation decisions which not only consider the static achievability but also the achievability under the dynamic context and its effect on the fulfilment criteria of a goal. The achievability of a goal and the space of adoptable alternatives to achieve it are essential information to plan adaptation, seen as a selection and an enactment of a suitable alternative to reach a goal under a certain context.

However, the enhanced expressiveness allied with the context-dependency also largely affects the model’s variability. Given the multitude of possibilities and alternatives given by the PGM (OR-Nodes, contexts, task and goal-annotations, QoS-Constraints and provided QoS levels), finding a configuration within the expected behaviour considering both task and composite quality constraints may become humanly infeasible.

Thus along with the definition of a pragmatic goal model we have also developed and implemented the QoS-Constrained Planning Algorithm to engineer an execution plan that abides by the interpretation of the goals within the model, if possible, or to lay out the best effort allowed by the model under the current context.

3.7.1 Goal Achievability Function and QoS-Constrained Planning Algorithm

In a nutshell, the QoS-Constrained Planning Algorithm algorithm initially performs a depth-first filter, aggregating the Task Quality Constraints on each branch of the CGM tree, so that all task-wise constraints respect all the upper level goals' interpretations. Once it finds a leaf-node (task), it uses the collected task quality constraints to decide whether the task at hand is able to fulfil the collected constraints. Then, it returns this task and begins agglutinating, at each goal, the possible plans for its refinements, always choosing the plan that provides the best quality for the composite requirement being considered[1].

We present the QoS-Constrained Planning Algorithm in Algorithm 1. It implements the `Requirement` entity's *isAchievable* method (Figure 3.2) and correlates three context-dependent aspects from the model: (1) the applicable requirements; (2) the goals' interpretations and; (3) the delivered QoS level provided by the tasks.

The QoS-Constrained Planning Algorithm algorithm decides whether the root goal is achievable and, if so, lays out an execution plan, *i.e.* a workflow, which is most likely to achieve the desired QCs.

QoS-Constrained Planning Algorithm considers firstly the type of refinement it is dealing with. Should this refinement be a task (line 5) then the algorithm can decide whether this particular task abides by all of the task-wise and workflow-wise constraints (`canFulfil(interp)`) and returns a plan consisting solely of `this` refinement, already indicating whether it is a usable plan or not (lines 7 and 8). On the more likely case that this is not a leaf-goal and, therefore that this node has refinements, the first step in the algorithm is to invoke itself recursively for each refinement so that the plans for its subgoals are created (lines 11 - 14) and save them.

Once it has knowledge of the outputted plans for its refinements, it can begin on the workflow-wise QoS constrained planning. At this point, the QoS-Constrained Planning Algorithm algorithm interacts with the `goal annotation` to obtain all the allowed behaviours for the achievement of this refinement (line 16), already annotated with time and reliability QoS metrics calculated through the rules presented in Table 3.1. Initially, the algorithm accepts any approach to realize the goal at hand available, disregarding whether it is achievable or not (line 20). Then, for each successive possible approach (line 18), the algorithm compares the `candidateApproach` with the previously `chosenPlan` to choose the best alternative in terms of achievability (line 22-23) or QoS (lines 24-25), always striving to return the approach with the best possible QoS for the Composite QC being considered in the given interpretation.

The QoS-Constrained Planning Algorithm algorithms validates whether the best candidate plan is still within the interpretation's composite QC limits (line 29) and, if it is not, sets the plan as unachievable.

Either way, the best candidate plan - achievable or the goal's best effort - is finally returned in the end. To exemplify the algorithm, we now present the following scenario: a patient with a heart condition ($C10$) needs medical assistance and needs certainty that it will arrive. He currently has good data coverage ($C4$) and mobile data enabled ($C5$). He is also in a dim location ($C7$) which tolerates phone ring ($C6$). However, the area's GSM tower density is low ($C11$).

In this situation, the algorithm would analyse the Pragmatic GM prioritizing the reliability quality constraint. It would then suggest the plan depicted in Figure 3.4. We also present, in Figure

[1]Currently, only one composite requirement can be considered at each QoS constrained planning.

Algorithm 1 Pragmatic Planning Algorithm

```
procedure qosConstrainedPlanning(Context current, Interpretation interp)
    if !isApplicable(current) then
        return NULL
    end if
    if (isTask()) then
        Plan pTask ← new Plan(this);
        pTask.setAchievable(canFulfill(interp));
        return pTask
    end if
    Map<Refinement, Plan> refPlans;
    for all Refinement ref in getApplicableDependencies(current) do
        Plan pRef;
        plan ← ref.qosConstrainedPlanning(current, interp);
        refPlans.put(ref, plan);
    end for
    List <Plan> approaches ← getRuntimeAnnotation()
                                    .getPlans(refPlans);
    Plan chosenPlan ← NULL
    for all Plan candidateApproach in approaches do
        if chosenPlan == NULL then
            chosenPlan ← candidateApproach;
        else
            if (!chosenPlan.getAchievable() AND
               candidateApproach.getAchievable() ) then
                chosenPlan ← candidateApproach;
            else if (chosenPlan.getAchievable() AND
                   candidateApproach.getAchievable()) then
                chosenPlan ← chooseBetterPlan(candidateApproach,
                                              chosenApproach, interp);
            end if
        end if
    end for
    if !interp.withinLimits(chosenPlan) then
        chosenPlan.setAchievable(false);
    end if
    return chosenPlan
end procedure
```

3.5, the original Pragmatic GM with the chosen tasks highlighted to depict the adherence of the chosen tasks to the current context and goal annotations.

To comprehend the reasoning performed in choosing this approach, we must first identify the alternatives available at such time. To facilitate the reader's understanding of this example, all the tasks are considered to have the same reliability level, let's say 99% reliability.

So first, it considers the root goal and the fact that it is an AND-decomposition with linear sequencing of tasks. Then it invokes the QoS-Constrained Planning Algorithm algorithm recursively for goal $G2$, then a new recursive call for $G6$, and yet another for $G11$. At $G11$ the recursive calls made over tasks $T1$ and $T2$ return a plan consisting of task $T2$ and a `null` plan, since $T1$ is inapplicable under the considered context. $G11$ then chooses such plan and returns this to $G6$. $G6$ now invokes the QoS-Constrained Planning Algorithm algorithm at $G12$, which checks and sees that $G16$ is unachievable it is not applicable at the current context. Thus it returns a plan consisting of

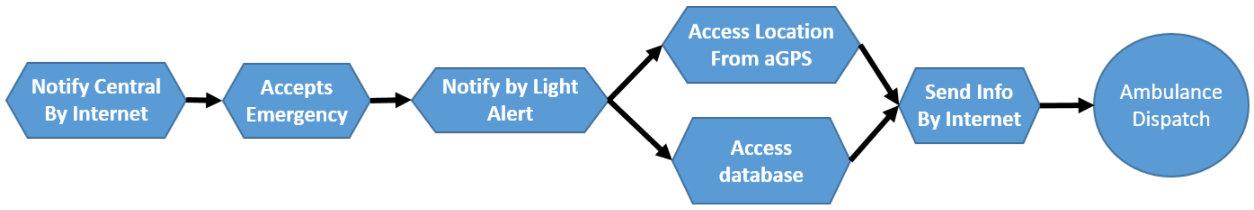


Figure 3.4: *Workflow output for context [c4, c5, c6, c7, c10, c11]*

$T3$.

Now that $G6$ has a complete plan for $G11$ ($T2; T3$), it invokes the QoS-Constrained Planning Algorithm algorithm on $G7$. $G7$ execution returns a plan consisting of ($T7\#(T5\#T6)\#T8$). Considering the annotations in Table 3.1, the observed reliability of these two alternative plans for $G2$ would be 0.9801 for ($T2; T3$) and 0.96059601 for plan ($T7\#(T5\#T6)\#T8$). Since the QoS-Constrained Planning Algorithm algorithm is prioritizing the reliability QoS constraint, the plan ($T2; T3$) is returned as the best alternative for $G2$ and the algorithm proceeds in similar way for goals $G3$, $G4$ and $G5$, providing the plan

$$T2; T3; T11; (T18\#T19); D1 \tag{3.1}$$

as a valid alternative for this situation. This plan is depicted as a workflow in Figure 3.4, whereas Figure 3.5 highlights the chosen tasks in blue to aid the reader to verify their adherence to the model's semantics.

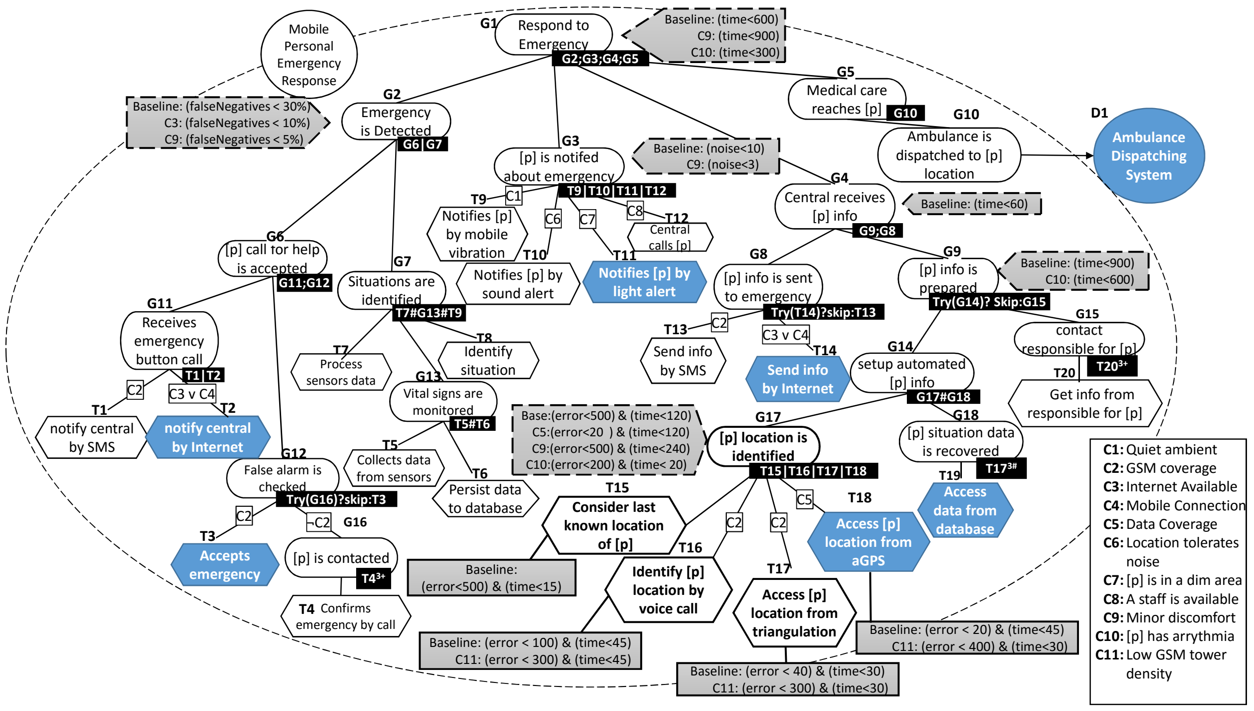

Mobile Personal Emergency Response
G1 Respond to Emergency
G2;G3;G4;G5
Baseline: (time<600) C9: (time<900) C10: (time<300)
G2 Emergency is Detected
G6|G7
Baseline: (falseNegatives < 30%) C3: (falseNegatives < 10%) C9: (falseNegatives < 5%)
G3 [p] is notifed about emergency
T9|T10|T11|T12
Baseline: (noise<10) C9: (noise<3)
G5 Medical care reaches [p]
G10
G10 Ambulance is dispatched to [p] location
D1 Ambulance Dispatching System
G4 Central receives [p] info
G9;G8
Baseline: (time<60)
T9 Notifies [p] by mobile vibration
C1
C6
C7
C8
T12 Central calls [p]
T10 Notifies [p] by sound alert
T11 Notifies [p] by light alert
G6 [p] call for help is accepted
G11;G12
G7 Situations are identified
T7#G13#T9
G8 [p] info is sent to emergency
Try(T14)?skip:T13
G9 [p] info is prepared
Try(G14)? Skip:G15
Baseline: (time<900) C10: (time<600)
G11 Receives emergency button call
T1|T2
C2
C3 v C4
T7 Process sensors data
T8 Identify situation
T13 Send info by SMS
T14 Send info by Internet
G14 setup automated [p] info
G17#G18
G15 contact responsible for [p]
T20 T20³⁺ Get info from responsible for [p]
T1 notify central by SMS
T2 notify central by Internet
G13 Vital signs are monitored
T5#T6
Base:(error<500) & (time<120) C5:(error<20) & (time<120) C9:(error<500) & (time<240) C10:(error<200) & (time< 20)
G17 [p] location is identified
T15|T16|T17|T18
G18 [p] situation data is recovered
T19 T17³⁺ Access data from database
G12 False alarm is checked
Try(G16)?skip:T3
T5 Collects data from sensors
T6 Persist data to database
T15 Consider last known location of [p]
T16 Identify [p] location by voice call
T17 Access [p] location triangulation
T18 Access [p] location from aGPS
C5
T3 Accepts emergency
¬C2 G16 [p] is contacted
T4³⁺
T4 Confirms emergency by call
Baseline: (error<500) & (time<15)
Baseline: (error < 100) & (time<45) C11: (error < 300) & (time<45)
Baseline: (error < 40) & (time<30) C11: (error < 300) & (time<30)
Baseline: (error < 20) & (time<45) C11: (error < 400) & (time<30)
C1: Quiet ambient
C2: GSM coverage
C3: Internet Available
C4: Mobile Connection
C5: Data Coverage
C6: Location tolerates noise
C7: [p] is in a dim area
C8: A staff is available
C9: Minor discomfort
C10: [p] has arrythmia
C11: Low GSM tower density

Figure 3.5: *MPERS model with tasks chosen by the QoS-Constrained Planning Algorithm algorithm highlighted in blue*

From Figure 3.5, we can see that such workflow respects all the annotations' rules in the model: it contains tasks from $G2$, $G3$, $G4$ and $G5$, sequentially linked between them; it contains tasks from $G6$ but not $G7$ as defined in $G2$ alternative annotation; it contains $T3$ because $G16$ is unachievable given that $C2$ is not active effectively deeming $G16$ unachievable. From the four $G3$ dependencies, the only ones applicable for the patient's context were $T10$ and $T11$, where the algorithm opted for

$T11$, which provided better Composite QoS levels. $G4$ annotation required that $G9$ subtree tasks (“Access location from aGPS” and “Access Database”) were performed before sending info over the internet. Also, since $G14$ was achievable, the *else* branch for $G9$ was not executed because G14 was achievable. Finally, the $G10$ delegation was performed and the root goal achieved correctly.

Chapter 4

IMPROV Architecture

Chapters 1 and **??** have presented the problem we are trying to solve; in this Chapter, we introduce our solution: the IMPROV architecture. The term *improv* is coloquially used as an abreviation for improvisation. The Cambridge dictionary defines improvisation as *a performance that has not practised or planned* or *the act of doing something with whatever is available at the time*. This idea sums up the main goals of the IMPROV architecture: allowing for dynamic choreographies which depend on the current state of the world (context), the provided QoS levels for the tasks and on the user's expected level of quality, in a request-centered and individualized manner.

The proposed architecture provides a dynamically adaptive service choreography that respond to changes in the user requirements and in the context. It adapts the service composition so that the performed choreography uses only applicable alternatives under the current context and delivers QoS levels which abide by te user's expectations. It also ensures that the architecture can autonomously withstand and recover from failures in the web services and in the actual choreography actors and that the actors themselves can have sufficient autonomy to define its own priorities and plans while also being able to evolve or change its own behavior without disrupting the dynamic composition as a whole.

To reach such goals, the IMPROV architecture performs the steps depicted in Figure 4.1. Initially, it receives a set of QoS requirements from the user and the contextual information for the system's environment. Then the agents which implement the architecture communicate among themselves in order to dynamically define the choreography and fault-tolerance techniques to be used to achieve such QoS levels. Finally, the actors move on to the distributed and fault-tolerant execution of the devised choreography, via a decentralized peer-to-peer communication scheme, while monitoring the QoS levels provided by the tasks in order to keep track of their performance and further enhance future QoS level predictions on future invocations.

In a nutshell, our proposal presents the IMPROV architecture which can adapt to different QoS requirements, different contexts and different levels of provided QoS via a decentralized planning algorithm which allows several autonomous actors to engineer a service choreography at the time of execution and deliver the expected QoS under various scenarios.

During the next Sections we will depict the IMPROV architecture. Initially, we present how to depict actors' behavior through a PRCGM model; next - in Section 4.1.1 - we go from the abstract and high-level objectives presented in the first chapters to more palpable requirements for the IMPROV architecture. Then we begin to drill down into the architecture beginning by describing the MAPE-K cycle that summarizes the higher-level process in Section **??** and the supporting communication structure in Section 4.3.2. At this point we move on to describing IMPROV's core element: the actors. Section **??** presents their internal modules, their interactions and the part they play on the composition. In Section 4.5 we introduce and categorize the faults considered by our architecture and the fault-tolerance mechanisms in place to mask or recover from such faults. Finally, during Section **??** we present the execution process and actor's interaction for carrying out the complete decided execution, Section **??** presents the lifecycle of a dynamic composition and Section 4.7 present some of our project decisions and the reasons behind them.

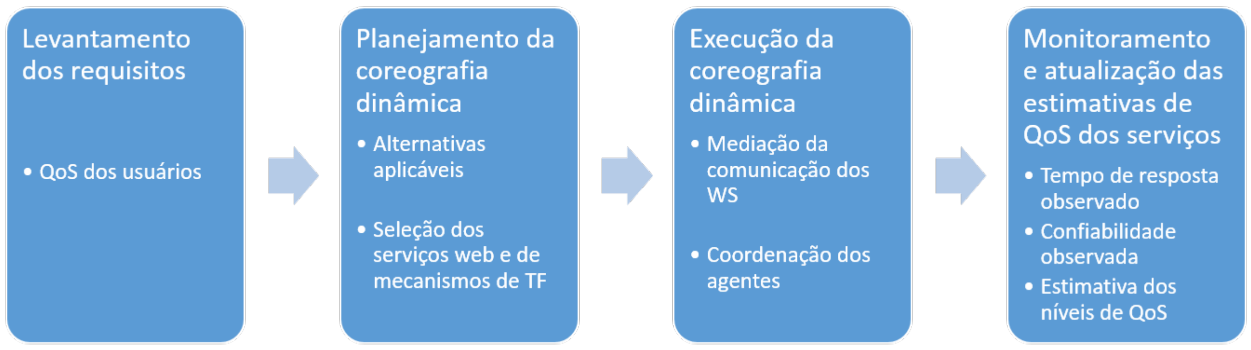

Levantamento dos requisitos
• QoS dos usuários
Planejamento da coreografia dinâmica
• Alternativas aplicáveis
• Seleção dos serviços web e de mecanismos de TF
Execução da coreografia dinâmica
• Mediação da comunicação dos WS
• Coordenação dos agentes
Monitoramento e atualização das estimativas de QoS dos serviços
• Tempo de resposta observado
• Confiabilidade observada
• Estimativa dos níveis de QoS

Figure 4.1: *Visão macro do processo de tratamento de requisições pela arquitetura IMPROV*

4.1 Using a Pragmatic GM to model actor behaviour

In the IMPROV architecture, we use Pragmatic Goal Models to describe the actors' possible behaviours and context-dependent strategies. Through these models the actors are able to understand their role in the choreography. They may also evaluate the applicable alternatives to provide a suitable QoS level and collaboratively interact with the other actors to dynamically engineer a service choreography that would meet the user's pragmatic requirements.

Each actor's task is mapped to a service method invocation, which will actually perform the task. The IMPROV architecture and its modelling is mainly concerned with the integration and dynamic nature of the composition. This means that the model will allow a choreography to have a different topology for each enactment. This also means that the architecture is able to remain agnostic to the underlying implementation details by restricting itself to the SOA interface level.

With regard to the PGM design process, there are two possible ways of defining the actors' PGMs: either top-down or bottom-up. The top-down approach is more natural when using the PGM as a RE tool. Starting from the root goal definition and then decomposing it into sub-goals and/or tasks. The sub-goals are then refined and the process is repeated up until all the leaf nodes in the model are simple enough to be considered as tasks and mapped to a service invocation.

On the other end, a bottom-up design process starts from the available services and their methods, which are represented in the PGM as tasks. Then the choreography designer pieces them together and generates higher-level goals. This process is also repeated by creating even higher abstraction level goals up until the point in which the root goal can be defined.

Regardless, the task-nodes are mapped to a web service invocation which are the building-blocks for any service composition [WB10].

The next step in the choreography design and actor behaviour description is the definition of the task/goal sequencing and the definition of the applicable fault-tolerance techniques which may be applicable for each task and their contextual restrictions.

Using a PGM with goal sequence and task fault-tolerance annotations allows us to:

1. harness the power provided by the Goal-Model as a requirement engineering tool and as an easily user-understandable model;

2. depict the complexity of the underlying variability and adaptivity alternatives in an easily understandable model and;

3. formally state the possible execution alternatives and fault-tolerance strategies.

This formal definition of the possible execution alternatives is crucial for the actors' autonomy. Since we have a formal statement and strict applicability rules (goal decompositions, context and sequencing) the actors are able to understand the role they are supposed to play. This, in turn, allows them to decide on an acceptable strategy built to suit the user's QoS requirements. Such strategy is composed of a valid sequence of tasks and delegations.

4.1.1 Requirements for Composition of Unmodified Web Services

As previously discussed during Chapters 1 e **??**, one of the main goals for IMPROV is to allow the distributed composition of unmodified web services. Such services follow the SOC main characteristics of *loose coupling*, *technology neutrality* and *location transparency* [Pap03]. Thus, the requirements for the composition of generic web services below derive both from such characteristics as well as from reliability issues.

In [Pap03], some of the main characteristics of SOA are presented. Its services must be invocable through standardized technologies (*technology neutral*), must not require knowledge about the implementation (*loosely coupled*) and have its definition and location publicly available (*location transparency*). These characteristics do not allow for intrusive interventions. Moreover, services in-the-wild are usually beyond a user's control scope anyhow. Thus, any interaction must be done according to the standard protocols and with little or no knowledge of the host's platform.

One of IMPROV's main goals is to enable a distributed, fault-tolerant and context-aware web service composition. But given these characteristics, it becomes clear the need to deal with unmodified web services instead of developing specialized services, at the risk of losing the SOA's benefits.

Therefore we will, during this Section, break this goal apart into several concrete requirements. These requirements will ensure the reuse of existing models, concepts, tools and systems. They will guarantee that existing systems will be able to be integrated and interoperate in novell ways, dynamically adapting their behaviour to changes in the system's context and in the user's QoS requirements without the need for deep modifications or code refactoring.

These requirements will be central in the remainder of our work, serving as the guiding principle for most of the taken decisions and providing a clear objective to be achieved by the IMPROV architecture. To assist the reader and materialize the concepts, we will use the *MPERS* use case presented in Section **??** as a leading example and present how each of these requirements falls inline with the use case scenario.

Unmodified Web services A core requirement to IMPROV is the ability to integrate non-specialized, previously existing web services in a novell way. However, the *loose coupling* characteristic of web services implies that a service composition must not make any assumption regarding the provided service's implementation (no mandatory software installation, service specialization or any other modification to the services themselves). Such requirement also implies that the IMPROV architecture may not pose any requirement whatsoever onto the underlying web services, except for those required by SOA itself (see Section **??**). The architecture must work with both *in-house* and *in-the-wild* services, and therefore beyond the users' administrative domain. In the `MPERS` use case, we have a clear example for this requirement. In `MPERS` we have several different administrative domains (...). In such an heteregeneous environment, demanding that all involved services abide by a pre-defined set of ruled, provide a set of data or present a certain behaviour would severely cripple the architecture's applicability and its capability to interoperate with, for instance, legacy systems that do not have such requirements.

Requirement 1: The service composition mechanism must respect the loose coupling characteristic of SOC, thus not demanding modifications to the service, code or infrastructure-wise.

Context- and user's requirement-aware adaptivity As previously discussed in Section 2.4, the environment may have deep repercussions on the system's behaviour, either by forcing or preventing some alternatives, altering the underlying services' provided QoS levels or by requiring a certain level of quality. The user may also impact the possible behaviours by demanding a given level of quality. Moreover, such variability in the planning's input variable makes it impossible to apply predetermined strategies. On the other hand, using a runtime planning approach may mitigate the variability impact by considering only applicable alternatives and current environmental and contextual data at the cost of runtime execution delays. Being so, the IMPROV architecture must decide on the choreography topology, the tasks sequencing, service selection and fault-tolerant alternatives at runtime. Another key-issue brought about by this requirement is the need for runtime responsivity to contextual changes. If, during the performance of a choreography a context change occur the architecture must halt, decide on the impact of such a change and if the plan may be kept and then proceed with the prestablished execution or fallback to the planning phase if the context change renders the current plan inapplicable or inadequate.

Requirement 2: The service composition mechanism must respond to user's requirements and to environmental change presenting a suitable behaviour that respects the limits imposed by both.

Tolerate services and architectural failures The choreography's reliability as a whole depends directly on the reliability of its underlying services. To be able to deliver QoS levels greater than the mere direct composition of services - which may not be sufficient for achieving the expected QoS levels - it is necessary for the achitecture to implent fault-tolerance techniques. IMPROV must, then, provide fault-tolerance techniques that build-up on the multiplicity of available WS to provide higher reliability levels. It must also ensure the smooth functioning of its own components and provide manners to deal with architectural faults and recover the architecture so that the compositions stays operational. It must also allow for the decentralization of its components so that no resource becomes a single point of failure for the composition as a whole. In our leading example, this need is rather clear given that a failure in the process may lead to the ambulance not reaching the patient in time.

Requirement 3: WS' and agents' faults must be masked or recoverable up to the required reliability level.

Requirement 4: Fault-tolerance techniques and their mechanisms may not pose assumptions on the quality or quantity of available services and must adapt to a changing environment to provide reliability

Requirement 5: Web services *in the wild* cannot be blindly trusted regarding content and time domain failures.

Involved agents' are autonomous Requirement 1 states that previously existing services may be distributed among different administrative domains. The same also applies to choreography actors. The choreography must not be - by design - in a single administrative domain. Instead, it must allow for different actors in different administrative domains to interact and collaborate. This means that the actors must be provided with complete autonomy over their own decisions. Each actor must be sovereign to decide on its own actions and no given set of options or decisions may be imposed on it by any other actor. The distinct choregraphy actors, similarly to what happens in the actual artistic performance, must collaborate in a constructive way to engineer and perform the choreography but without interfering on internal affairs of one another. In the leading example, this requirement states that the `MPERS` actor may report on the user's QoS levels and "ask" the `Ambulance` actor's assistance in achieving the root goal and receive an execution plan from the `Ambulance` actor but it may not impose any requirement on the provided plan, except for the goal's interpretation.

Requirement 6: Actors in the choreography are autonomous and may not be imposed any restrictions except for those explicitly provided by the architecture or by themselves.

Allow the evolution of the actors' behaviour over time Demanding that all the subtleties of the context-dependent composition be understood before implementing the choreography is not acceptable. In a scenario with such variability is only natural that the requirements and the understanding of the problem may evolve over time. Even further, new alternatives may arise and older ones may be rendered obsolete. The IMPROV architecture must allow for agents to revisit and change their behaviours over time, either by adding or removing elements from their PGMs. This requirement relates to requirement 6 in the sense that the actors' autonomy also implies that an actor may change its behaviour without previous notice. Thus, the architecture must ensure that such modifications are performed transparently to the composition as a whole with little impact on the other actors and on the current and future executions of the composition. This is paramount to the viability and maintanability of the system as a whole. As for our leading example, this requirement means that a change in the `Ambulance` actor's behaviour will not require any modification on the `MPERS` actor. The reciprocal also applies, changes in the `MPERS` actor's behaviour does not imply or depend on any modification of the `Ambulance` actor's behaviour.

Requirement 7: Changes in an actor's behaviour must not impact other actor's behaviour definitions

4.2 MAPE cycle

IMPROV's core elements are its Actors. The choreography actors are the elements responsible for planning and delivering the dynamic service compositions. It is in the actors that all of IMPROV's data structures and features lie.

Each actor brings along the knowledge of how to perform its role in the composition by consuming a set of web services and interacting with a set of adjacent actors. To do so, each actor performs a MAPE-K cycle where it continuously monitor its web services and analyse their historical data in order to enhance the actor's knowledge about their behavior and enhance its capability to estimate the QoS for the subsequent WS invocations. It is then able to plan a web service composition under QoS and Context constraints and finally execute it in a fault-tolerant way.

To present how the IMPROV architecture uses the MAPE-K cycle we will during this Section present the macro view of the process, leaving the underlying details for the following sections of the thesis.

The cycle starts with the actors monitoring the contexts defined in their PGM and the services they utilize to perform the PGM's tasks.

The second step in the MAPE-K cycle - Analysis - takes the collected monitoring data and transforms it into usable information. In our case, in this phase the actor uses the context information to exclude behavioural alternatives rendered inapplicable by the current context and the WS quality levels to enhance its estimate on the expected QoS-level for the next executions.

During this step, the actor drills down the PGM to exclude behavioural alternatives, rendered inapplicable by the current context. At this stage, it also communicates with other actors in order to figure out the possible alternatives and QoS levels that they can provide.

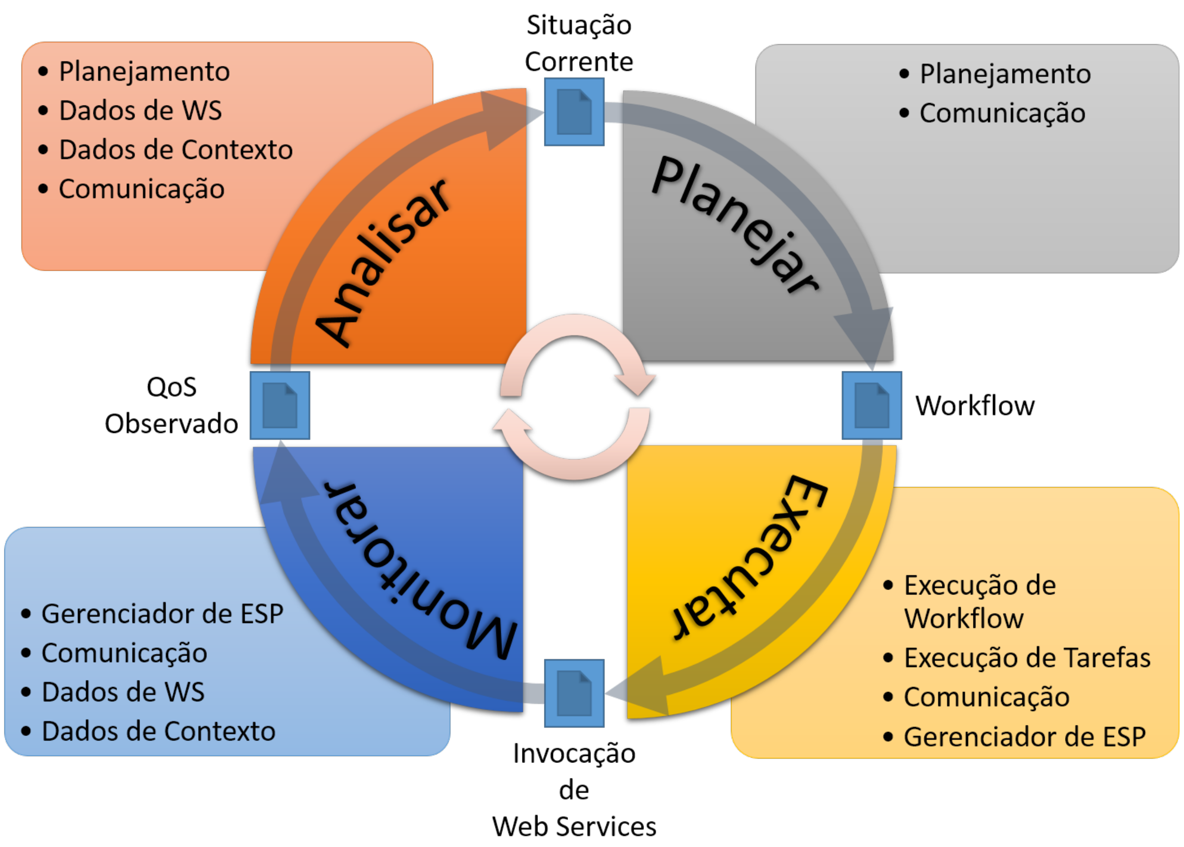


Figure 4.2: *MAPE cycle implemented by the agents and involved*

At the analysis step of the MAPE-K cycle, the actor drills down the PGM to exclude inapplicable behavioural alternatives under the current context. At this stage, it also communicates with other actors in order to figure out the possible alternatives and QoS levels that they can provide.

The third step - Planning - is then triggered by an user's request or an actor's delegation. In this stage the actor interacts with other actors and compares the applicable alternatives to engineer a plan which can achieve the expected QoS levels under the current context.

Finally, in the Execution step, each actor performs the service invocations that perform the laid out tasks and communicates with adjacent actors to delegate the goal executions.

4.2.1 Monitoring

The MAPE-K cycle starts with the monitoring of the system's environment. In IMPROV, the monitoring aspect has two main focuses: contexts and WS QoS-levels.

Context monitoring is a crucial part of the IMPROV architecture in the sense that the actor's behaviour is itself limited by the current state of the environment. As described in its PGM, some circumstances may force or prevent some alternatives to be used. Therefore, every actor involved in the choreography must be aware of its environmental context and respond adapting its behaviour accordingly.

Context monitoring is not, however, performed actively by the actor itself. Instead, external monitors and sensors evaluate the current state of affairs and report them to the actors. This allows for more comprehensive and application-specific solutions and does not limit the concept or the definition of "context" but leaves it open for the choreography designer to define what a context is and how to monitor it.

The service monitoring on the other hand is performed by the actor. This is by itself a challenge for the IMPROV architecture because of the non-intrusiveness requirement described in Section 4.1.1. Since the actor holds no knowledge of the WS implementation and can only interact with it via service invocations, the service QoS-level monitoring is performed on externally observable attributes. As per exemplification, we list the measured time between the WS invocation or the response receipt and the amount of times it has timed-out. Although other externally observable metrics are viable and may be used within the IMPROV architecture with little modification to the actors, for the remainder of this paper we will focus on time and reliability metrics.

In IMPROV, the QoS and reliability metrics monitoring is performed by logging the WS invocations and evaluating the related metric. For the performance monitoring, before every WS invocation the actor starts a clock and then stops it upon the response receipt. Thus, the whole communication process can be monitored.

The reliability monitoring is a little more complicated in the sense that the actor may capture both time and content domain faults. Time faults are simpler in the sense that a response is considered a success if it has been received and a failure if a timeout has been detected regardless of the fault-tolerance technique used. On the other hand, content domain faults can only be detected when using a voting FT mechanism. In this scenario, if a service has produced an answer distinct from those produced by the other instances, this is considered a content domain fault and duly logged as such.

4.2.2 Analysis

The analysis step of the MAPE-K cycle consists of turning the monitors' gathered data and transform it into usable information. Provided IMPROV strives to present itself as an architecture fully compatible with pre-existing web services and that such web services may have different QoS levels over different time spans, the QoS-levels estimates cannot be trivially considered. Using a simple average calculation would result in slowly responding and adapting to changing loads and behaviours. Timely response to changes and variations in the WS QoS levels require more emphasis in recent values when compared to older ones.

To do so, we use the historical QoS-levels sequence and apply a technique similar to what has been used for years in Memory management: the Aging algorithm [1].

The aging algorithm is a descendant of the not frequently used (NFU) algorithm, with modifications to make it aware of the time span of use. Instead of just incrementing the counters of pages referenced, putting equal emphasis on page references regardless of the time, the reference counter on a page is first divided by 2, before adding the referenced bit to the left of that binary number. For instance, if a page has referenced bits 1,0,0,1,1,0 in the past 6 clock ticks, its referenced counter will look like this: 10000000, 01000000, 00100000, 10010000, 11001000, 01100100. Page references closer to the present time have more impact than page references long ago. This ensures that pages referenced more recently, though less frequently referenced, will have higher priority over pages more frequently referenced in the past. Thus, when a page needs to be swapped out, the page with the lowest counter will be chosen.

In our case, for a given historic dataset $x_0, x_1 x_2, x_3 \dots x_i$ (with x_n being the most recent value), we apply a weighted mean T_i which represents the estimate for the WS QoS level x_{i+1}. The degree at which the importance of the metrics decay can be defined by the designer for each WS and is expressed in the formula as δ. The weight attributed for metric x_i is then calculated by δ^{n-i}. At last, the formula as a whole can de defined as:

$$T_0 = x_0 \tag{4.1}$$

$$T_n = \sum_{i=0}^{n} \delta^{n-i} * \frac{x_i}{n} \tag{4.2}$$

This equation can be further enhanced by placing the most recent element x_0 element outside the sum:

[1] http://ieeexplore.ieee.org/document/5461964/

$$T_n = (\frac{x_n}{n} * \delta) + (\sum_{i=0}^{n-1} \delta^{(n-i)} * \frac{x_i}{n}) \tag{4.3}$$

$$T_n = (\frac{x_n}{n} * \delta) + (\sum_{i=0}^{n-1} \delta * \delta^{(n-i-1)} * \frac{x_i}{n}) \tag{4.4}$$

$$T_n = (\frac{x_n}{n} * \delta) + (\sum_{i=0}^{n-1} \delta^{(n-i-1)} * \frac{x_i}{n}) * \delta \tag{4.5}$$

$$T_n = (\frac{x_n}{n} * \delta) + (\sum_{i=0}^{(n-1)} \delta^{((n-1)-i)} * \frac{x_i}{n}) * \delta \tag{4.6}$$

$$\tag{4.7}$$

Since $\sum_{i=0}^{(n-1)} \delta^{((n-1)-i)} * \frac{x_i}{n}$ is, by definition, T_{n-1}, we produce the simpler equation:

$$T_n = (\frac{x_n}{n} * \delta) + (T_{n-1}) * \delta \tag{4.8}$$

This is an interesting approach because of workload-dependent QoS-level fluctuations for instance. The choreography designer may define an appropriate δ value for each service and allows for more rapid response to abrupt changes in behaviour when compared to the simple average approach. By default, IMPROV sets the δ value at 0.5. Also, since performance is a key matter in the IMPROV architecture, through equation 4.8 we can perform the estimate calculation in constant time, regardless of the amount of previous evaluations.

Another characteristic in the analysis of the WS QoS levels is that a time threshold may be established to determine a point in time when the estimate looses its value. This threshold allows for any estimate that has been made with old and out-of-date measurements to be discarded in favour of an expected QoS-level predetermined in the PGM model.

4.2.3 Planning

The planning stage of the MAPE-K cycle is triggered by a request at runtime, either a user's request or a fellow actor's delegation request. To perform the choreography planning, the actors build on the PGM's innate capability of providing an executable plan in linear time over the actor's behavioural model size to provide feasible execution alternatives within a considerably small time span.

Initially, an actor - which holds the root goal - receives an user request to improvise a choreography, possibly under a certain goal interpretation. This actor will then trigger the `Pragmatic Planning Algorithm` on its PGM, to engineer an execution plan. If the PPA algorithm reaches up a task-node, the latest analysed data will provide, alongside with the user's interpretation, the information the actor needs to decide on whether to choose that particular task and - if convenient - which fault-tolerance technique to employ. If, on the other hand, the PPA algorithm reaches a delegation-node, it recursively invokes the planning algorithm of its adjacent actor so that it can collaborate and participate on the improvised choreography. The adjacent actor will then, in turn, execute the PPA algorithm on its own PGM, reaching more tasks and more actors. Ultimately, the adjacent actor will report the QoS levels it can provide to the root actor and the execution plan that does so. Finally, with these information the root actor's PPA can, at last, decide on a complete execution plan to achieve the root goal under the user's QoS constraints.

4.2.4 Execution

After deciding on an execution plan the root actor may or may not start the execution phase of the MAPE-K cycle depending on the engineered plan ability to deliver the expected QoS level or, at least, an acceptably lower level. To start the execution phase, the root actor starts playing its role and, whenever needed, delegating goals to other actors. To delegate a goal during the execution phase, one must invoke the other actor's communication interface and supply him with the decided workflow (plan) it must follow as well as any required input data. The delegatee, similarly to what happened in the planning phase, will then start playing its own part in the choreography or, if input from other actor is also needed, wait for those. After playing its part, the actor will either pass the required data on to other actors or report back to the root actor depending on the workflow's layout. This process repeats itself until all the plan's tasks have been executed and the last actor reports back to the root actor the successful execution of the workflow.

4.3 IMPROV choreography agents

IMPROV architecture is implemented through what we have been referring to as Choreography Actors. These actors concretize the PGM actor entity in the sense that they are the ones who understand that they are taking part in a larger choreography and they are the ones who withhold the knowledge on how to perform their goals through task executions or fellow actors' collaboration. They are at the core of IMPROV architecture's structure thus providing the means through which IMPROV is able to deliver the user requirements described in Section 4.1.1.

Each actor is an autonomous entity that contains the knowledge on how to reach a given root goal under varying contexts, via collaboration or web service invocations. In this Section we will depict the internal workings of the choreography actors alongside with their internal structure, its communication and collaboration with other actors, its planning, WS handling, input/output management, fault-tolerance and WS monitoring mechanisms.

4.3.1 Communication Architecture and WS Handling

The Communication Architecture and WS Handling structure in the actors was designed to favour WS fault-tolerance and delivered QoS levels. At the same time, its design takes into consideration `Requirement 1`, in the sense that the actors must deal with unmodified web services, therefore they are not supposed to make demands or impose premises on the WS themselves other than those imposed by the SOA architecture itself.

The more traditional approach to dealing with unmodified and unspecialized web serviced is service orchestration. However, using an orchestrated structure would harm `Requirement 3` and `Requirement 6`. With regard to `Requirement 3`, we must consider that in an orchestrated structure the orchestrator holds all the execution information and all the knowledge about the composition thus being itself a single point of failure. Also, the delivered QoS levels would suffer due to the orchestrator's resource bottleneck in terms of communication [GKB+12]. `Requirement 6` would also be affected by the decision of working with an orchestrated approach in the sense that it presumes that the orchestrator holds the of the entire process, in order to plan the execution workflow. However, due to `Requirement 6`, no one entity may have the knowledge and the authority over the entire process. Each actor is autonomous to decide by itself on the proper way to dealing with requests.

Because of such considerations, IMPROV architecture utilizes agent-based software engineering (see Section **??**) and web services choreographies. Each IMPROV actor is then free to handle requests as desired and contain the necessary knowledge to do so. It works similarly to a transducer: receiving messages from other agents and mediating the communication between the actors and the underlying web services.

To play out their role in the composition, agents interact and communicate among themselves, via an exposed WSDL interface. This interface, as it will be further described ahead, is composed of web methods that permit the actor to be informed of external conditions of the environment or contexts, respond to planning requests, receive input needed as input for its tasks or to receive notice of another actor's task completion.

The interaction with the underlying web services on the other hand is entirely handled by an specialized module known as Equivalent Service Pool or ESP. This module handles all individual invocations to the web services and their monitoring and will be described in the next Section.

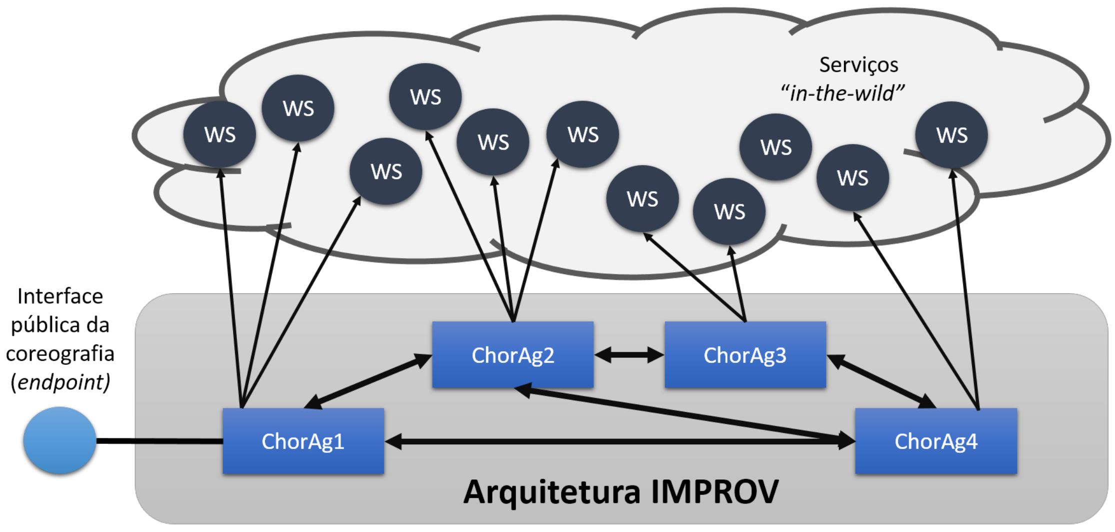


Figure 4.3: *Esquema de cooperação entre os agentes da coreografia.*

To summarize the communication architecture, we present Figure 4.4. In this image we can see four PRCGM actors (`ChorAg` 1 to 4) and several web services. Each agent deals with a subset of the available services through its ESP and also with adjacent actors through their control interface. `ChorAg 1` also exposes another WSDL interface which is referred to as the choreography's end-point. This is the endpoint through which the final user will be able to request the execution of the composition and inform its QoS requirements, if there is any.

4.3.2 Communication Architecture and WS Handling

The Communication Architecture and WS Handling structure in the actors was designed to favour WS fault-tolerance and delivered QoS levels. At the same time, its design takes into consideration `Requirement 1`, in the sense that the actors must deal with unmodified web services, therefore they are not supposed to make demands or impose premises on the WS themselves other than those imposed by the SOA architecture itself.

The more traditional approach to dealing with unmodified and unspecialized web serviced is service orchestration. However, using an orchestrated structure would harm `Requirement 3` and `Requirement 6`. With regard to `Requirement 3`, we must consider that in an orchestrated structure the orchestrator holds all the execution information and all the knowledge about the composition thus being itself a single point of failure. Also, the delivered QoS levels would suffer due to the orchestrator's resource bottleneck in terms of communication [GKB+12]. `Requirement 6` would also be affected by the decision of working with an orchestrated approach in the sense that it presumes that the orchestrator holds the of the entire process, in order to plan the execution workflow. However, due to `Requirement 6`, no one entity may have the knowledge and the authority over the entire process. Each actor is autonomous to decide by itself on the proper way to dealing with requests.

Because of such considerations, IMPROV architecture utilizes agent-based software engineering and web services choreographies. Each IMPROV actor is then free to handle requests as desired and contain the necessary knowledge to do so. It works similarly to a transducer: receiving messages from other agents and mediating the communication between the actors and the underlying web services.

To play out their role in the composition, agents interact and communicate among themselves, via an exposed WSDL interface. This interface, as it will be further described ahead, is composed of web methods that permit the actor to be informed of external conditions of the environment or contexts, respond to planning requests, receive input needed as input for its tasks or to receive notice of another actor's task completion.

The interaction with the underlying web services on the other hand is entirely handled by an specialized module known as Equivalent Service Pool or ESP. This module handles all individual invocations to the web services and their monitoring and will be described in the next Section.

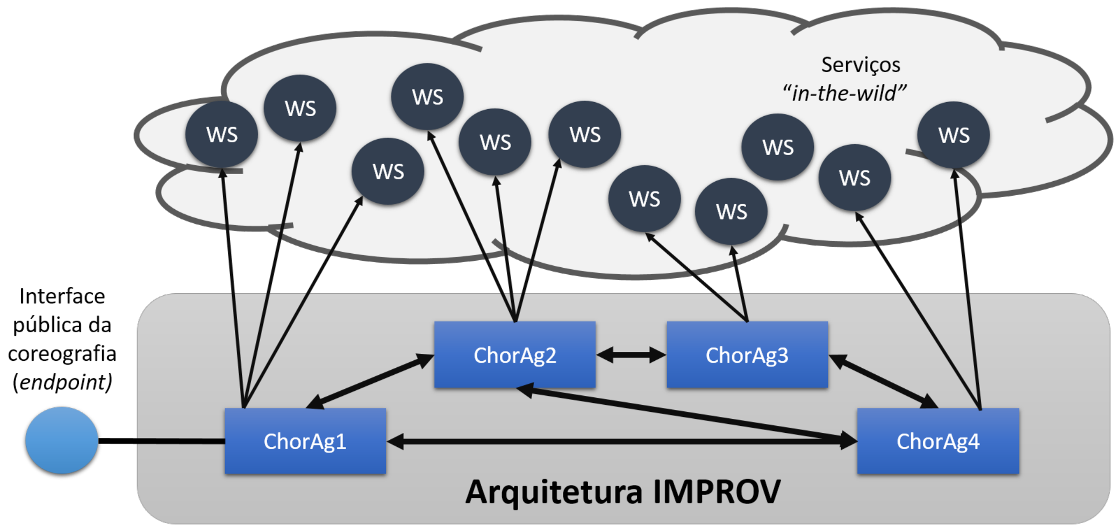


Figure 4.4: *Esquema de cooperação entre os agentes da coreografia.*

To summarize the communication architecture, we present Figure 4.4. In this image we can see four PRCGM actors (`ChorAg` 1 to 4) and several web services. Each agent deals with a subset of the available services through its ESP and also with adjacent actors through their control interface. `ChorAg 1` also exposes another WSDL interface which is referred to as the choreography's endpoint. This is the endpoint through which the final user will be able to request the execution of the composition and inform its QoS requirements, if there is any.

4.3.3 Agent Interactions

The interaction among several composition actors is made through SOAP requests. These requests are performed on the actors' control interface and, since they also rely on the WS* stack for the communication used for the interaction with the underlying services, they do not impose any further restrictions on the architecture or on the computational environment other than the ones already demanded by the web services themselves.

The option of using WSDL interfaces and SOAP requests to perform the communication among actors also facilitated the architecture's fault-tolerance and recovery mechanisms due to SOA's location transparency, loose coupling and late binding capabilitities. Such mechanisms allows the architecture to replace a failed actor with another replica of it instantiated at another location/resource and simply update the adjacent actor's endpoint for the control interface.

Each actor exposes its control interface which has a predetermined set of methods publicly available to receive messages from the other actors. On the other hand, sending a message to another actor is as simple as invoking a web method on its interface. It is then via this interface that all actor interactions occur. Planning and Analysis interactions are performed by the invocation of a "`plan`" method in the actor; execution coordination is performed by invoking the "`playRole`" method; context updates are informed using the "context" method and so on.

4.3.4 Internal Agent Architecture

The communication architecture laid out in Section 4.3.2 as well as the requirements presented in Section 4.1.1 set the basis for the engineering of the actors themselves, and their internal structure are a direct reference to such aspects.

In the communication architecture we have stated that actors work similarly to a transducer mediating the communication between the remaining actors, their web services and its own web service pool in order to enable the interaction and composition of those services into a higher level functionality. This capability is now translated into three main modules: the `Communication`, `Workflow Execution` and `Equivalent Service Pool (ESP) manager` modules. The `communication` module is the one responsible for exposing the WSDL interface and dealing/routing the requests for the remaining modules. On the other end of the architecture, the ESP manager deals with the invocation and monitoring of the web services it handles. Between these two ends, we have the `Workflow Manager` that evaluates and handles the sequencing of the tasks.

However, the actor's role in IMPROV is not limited to working as transducers to route the messages from users/actors to the proper web service. They not only handle the distributed communication and coordination execution of the composition but also engineer the execution plan according to the current environment situation, contexts and user-required Qos levels as well as incorporate fault-tolerance techniques to the composed service execution. These characteristics are incorporated in the actor through two other modules: `Planning` and `Task Execution` managers. The planning stage is performed in the `Planning module`, which harbors the algorithm presented in Section 3.7.1. The fault-tolerance mechanism implementation and management is performed by the `Task Execution Manager`

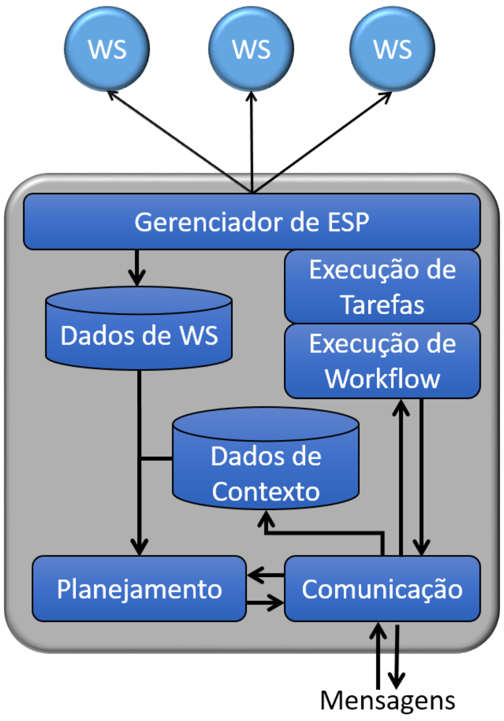


Figure 4.5: *Arquitetura Interna dos Agentes*

The final internal actors' architecture is illustrated in Figure 4.5. In it we can see the five introduced modules, the data flow between them and also two data sets which represent the data that the actor holds about its surrounding environment. Each of these elements are described below.

Communication The `Communication` module is the module responsible for handling all incoming and outgoing communications. It is the `Communication` module which exposes the control and, if necessary, the endpoint WSDL interfaces to receive communications from other actors, components or users. It also the `Communications` module that converts the actor's requests into SOA method invocations and ensures the receipt of the messages by the adjacent actors.

Planning In order to achieve Requirement 2 (Section 4.1.1), it is paramount that the composition's execution plan is decided collaboratively and at runtime. This is the main focus of this module: to plan the dynamic aspect of the choreography considering the current context, the possibilities described in the actor's PRCGM and the available service pool and their individual current QoS levels.

The `Planning` module is triggered by the `Communication` module upon receipt of a plan message from another actor on the control interface or by an user request on the endpoint. At this point, the planning module executes the algorithm described in Section 3.7.1 to cross-reference the current QoS level of the underlying web-services (registered in the WS Dataset), the context (context Dataset) and its own PRCGM. While executing the QoS-Constrained planning any possible delegation will also trigger an interaction with the delegate actor via outgoing messages posted by the `Communication` module. Finally, once all delegate actors have reported their plan, the algorithm is completed and the execution plan is devised. The produced task sequencing is then informed back to the `Communication` module, in the case of an actor-sent planning request, or to the `Workflow Execution` module in the case of an user request.

Workflow Execution After the proper course of action has been decided on, the root actor starts its execution of the plan. The execution is coordinated by the `Workflow Execution` module. This is the module responsible for the correct sequencing of the tasks and the overall workflow execution. Once it receives an execution plan, either from the `Planning` or from the `Communication` modules, it analyses it to decide the tasks apt for execution and sends them to the `Task Execution` module. Also, after the `Task Execution` module reports back and informs of the successful completion of a task, it is the `Workflow` manager's job to define whether this completion should be reported to adjacent actors so that they can carry out their part in the overall execution planning. It is this module responsibility to handle the input/output communication between workflow tasks.

Task Execution The `Task Execution` module plays an important role in the fault-tolerant aspect of the task execution. For an actor, a task execution is not merely mapped to a service invocation. Instead, it deals with task execution in such a way that failures in the underlying services would not necessarily impact or impede the execution of the workflow as a whole. Such insurance of successfuly executing the tasks in the composition is the primary focus of the `Task Execution` module. This module receives individual tasks from the workflow manager and dispatches a set of service invocations to the `ESP` module. It thens monitor the success of the individual invocations and takes action accordingly to the fault-tolerance technique chosen at the planning stage.

Equivalent Service Pool The `ESP` modules encapsulates the web service invocation process. This module handles the interactions between the actors and the task-providing web services it handles. At the same time, the `ESP` modules makes use of the service invocations to monitor the WS's behavior and update the observable QoS variables (time and reliability) estimates. These observations feed the WS Dataset with new and updated information, which will retroactively impact on future executions planning.

4.3.5 Internal Elements Interactions

Figure 4.5 presented not only the modules themselves but also the interactions between them. These interactions are the means by which the actors perform the MAPE-K cycle we have introduced

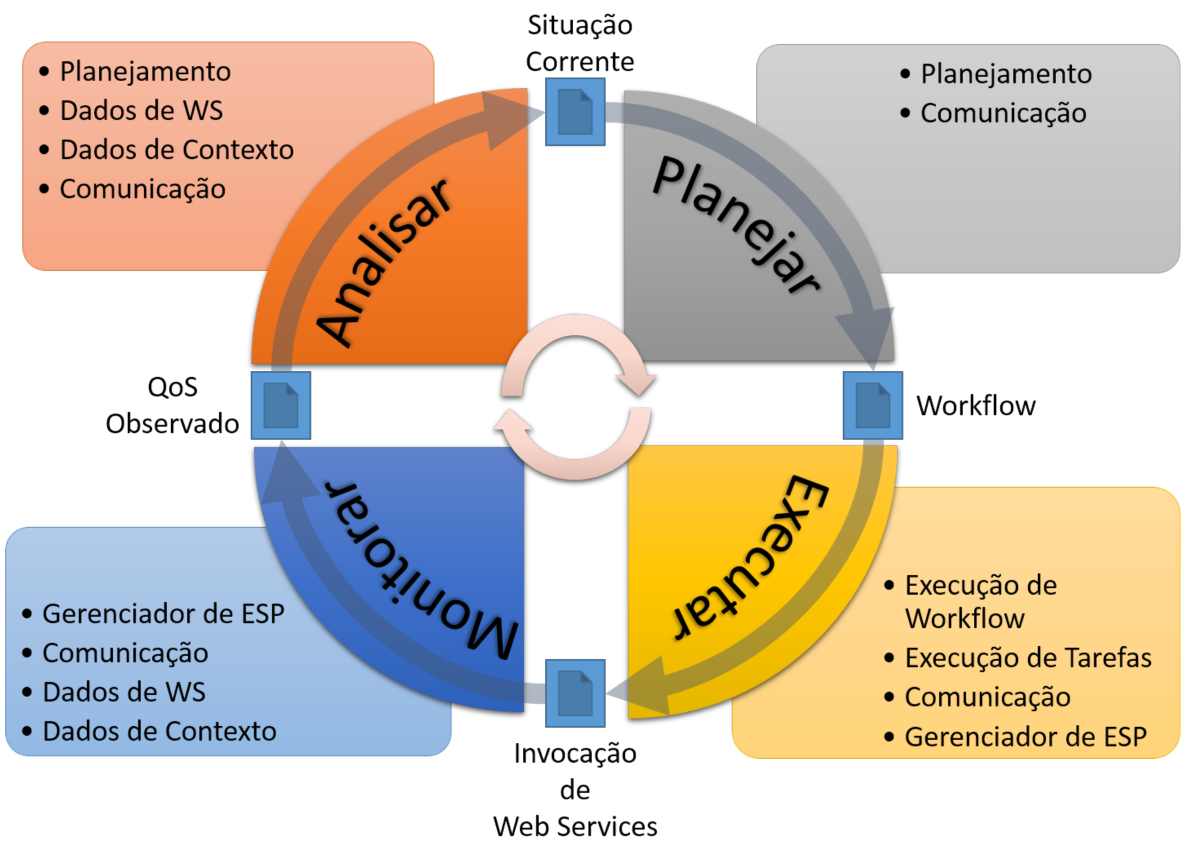


Figure 4.6: *MAPE cycle implemented by the agents and involved*

in Figure 4.2. It is now necessary to dig deeper into such interactions, specially considering their non-linear aspect, to envision the MAPE-K cycle it performs.

Figure 4.6 extends Figure 4.2 and places each of the actor modules into the stages it takes part on. It also introduces the macro output artifacts of each MAPE stage.

We will now present the MAPE-K cycle execution alongside the handling of a user request. Under such pretext, let's consider we already have up-to-date WS and context datasets and eventually received an user request for performing the composition. In this scenario, the first stage of our cycle would be the analysis stage. At this point, the user request was received by the `Communications` module and then forwarded to the `Planning` module. The `Planning` uses the PRCGM and `Contexts Dataset` to trim out any inapplicable alternatives given the current context.

In the next stage (planning), the `Planning` module uses the trimmed out PRCGM tree to engineer the actual execution plan using the QoS-Constrained planning algorithm and aided by the `Communication` module which communicates with delegate actors to gather their execution plans for the possibly delegated goals. At the end of this stage, the `Planning` module produces an execution plan that may involve several delegations and several actors.

Thus, during the planning stage the `Planning` module evaluates the distinct alternatives and paths to reach the root goal and the QoS levels they would provide, according to the gathered information available at the WS Dataset. It then decides on an alternative that is likely to satisfy the required QoS levels. It does so by agglutinating its own plans with the plans provided by adjacent actors in order to generate a global task workflow. This task workflow is the final output of this stage and represents the conclusion of the planning stage and the beginning of the execution stage.

In the third stage the execution of the devised plan is performed. This stage begins with the receipt of the execution plan by the worflow manager. During this stage the `Workflow` module oversees the tasks execution and handles their senquencing, both internally via `Task Execution` manager and externally sending messages to other actors through the `Communications` module. It parses the plan and dispatches tasks (with the appropriate fault-tolerance mechanism indication) to the `Task Execution` module and delegations to the communication mechanism along with any necessary input.

If, in one hand, the `Workflow Manager` is responsible for the overall task sequencing and coordination, on the other the `Task Execution` module oversees their individual execution process. The `Task Execution` manager receives the tasks and their fault-tolerance mechanism and converts it into the appropriate service invocations. Each of these service invocations is then passed on to the `ESP` module which then places the actual invocations onto the web services and relays the corresponding responses back to the `Task Execution` module which decides on the next step.

After successfully executing a task, the Task Execution manager reports the completion back to the `Workflow` Manager so that the workflow may be carried on.

The monitoring stage in IMPROV is actually performed at the same time as the execution itself. However, to make it easier on the reader to understand the flow, we placed this step as a final step. In IMPROV, to diminish the impact of the monitoring in the execution and due to the restrictions imposed by the requirements, we do not monitor the service themselves but actually monitor the interactions the actor has with it. One can think of the monitoring in IMPROV as the “personal opinion” the actor has on each service. During each service invocation performed by the `ESP` module it also controls the amount of time needed to receive a response. This amount of time is then added to the WS Dataset. For the context dataset, we have opted to update it using external features or sensors that notify the actor of a change through the `Communications` module.

4.4 Dynamic Choreography Execution

During the planning stage of the MAPE cycle, a workflow was devised. The creation of this document and its receipt by the `Workflow Manager` are the starting events of the dynamic choreography execution. During this process, the tasks in the workflow are evaluated and converted into a set of service invocations to be made by the `ESP manager` to the task implementing web services.

During the execution, each actor becomes a service monitor and an execution coordinator for the undergoing composition. Starting with the endpoint actor, each actor submits its available tasks to their web services and then they await the response from the WS.

Each workflow task already contains the expected fault-tolerance technique to be employed, the web services involved and the web method to be invoked, leaving for the workflow manager simply to point those out to the task manager.

The `Workflow` manager starts the process seen in Figure 4.7 by defining the tasks and delegations apt for execution.

The `Task Execution` manager in converts the tasks into an appropriate set of service invocations according to the chosen fault-tolerance technique and relays the responsibility for the individual invocations to the `ESP module`. The `ESP module` invokes the web service while also taking note on the elapsed time between the invocation and receipt of the reply. Once the response is received, the `ESP` relays the produced output to the `workflow manager` (5) who checks the tasks that will consume such output as well as the actor responsible for them. Finally, each of these actors is notified of the task completion and sent its output. Upon receiving a task completion notification and/or an input data the workflow manager re-evaluates the ongoing workflow to assess the existence of any newly available tasks. If the dependencies set of any task has been satisfied, that task is considered ready and promptly sent to execution.

Similarly, an available delegation is converted into an adjacent actor invocation which will trigger the same process at the fellow actor’s side. After the actor successfully executed its tasks in the choreography it will notify the current actor of the conclusion (5) and restart the process with the workflow manager re-evaluating the dependencies to check if any new tasks or delegations are ready.

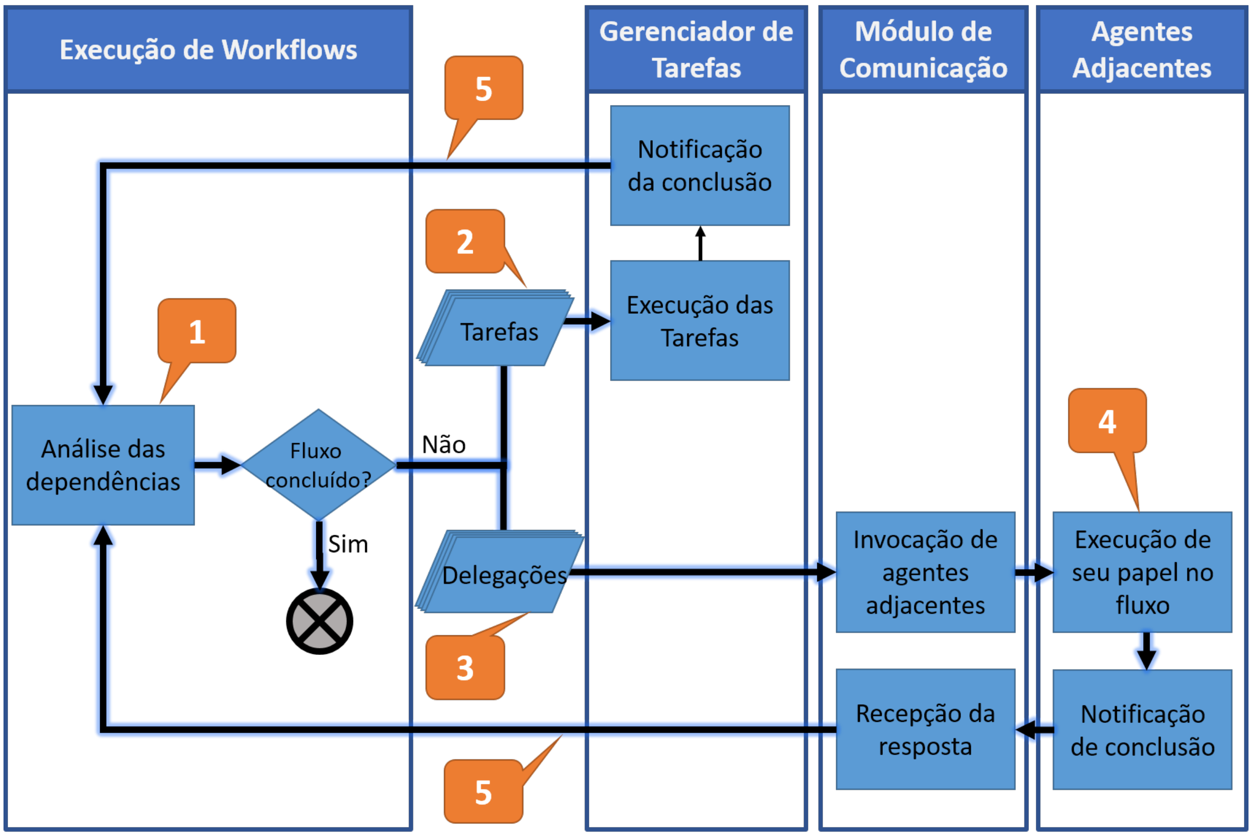


Figure 4.7: *Laço de execução do workflow*

4.4.1 Tasks Execution Process

After receiveing a task from the `Workflow Execution` module, the `Task Manager` module steps up to convert the tasks into service invocations and coordinate their sequencing. By doing so, it tries to make sure of the successful execution of the task in spite of eventual web service failures.

It has two basic elements: **(1) Fault-Tolerance Managers:** A implementation for each of the available FT techniques. **(2) Thread Factory**: A component to allow the parallel coordination of all invocations.

The `Task Execution` process is illustrated in Figure 4.8. As it illustrates, for each task received the `Task Manager Module`:

1. Instantiates a Fault-Tolerance Manager that will oversee and coordinate the service invocations in order to implement the desired fault-tolerance technique. The `Fault-tolerance manager` will then:

   (a) Create execution threads for each parallel invocation and each thread will:

      i. Interact with the *ESP Manager* to invoke the appropriate web service;

      ii. Await for the response receipt

      iii. Gather the outputted data;

      iv. Notify the FT Manager of the successful conclusion or of the timeout

   (b) After being notified of a Thread conclusion, the FT manager decides, depending on the FT mechanism it implements, whether to carry on with the execution, fail or wait for another thread's completion.

   (c) Once enough threads have successfully completed, the FT Manager decides on the outputted value for the task (for instance on a voting mechanism) and;

   (d) Notifies the Task Execution module of the produced response;

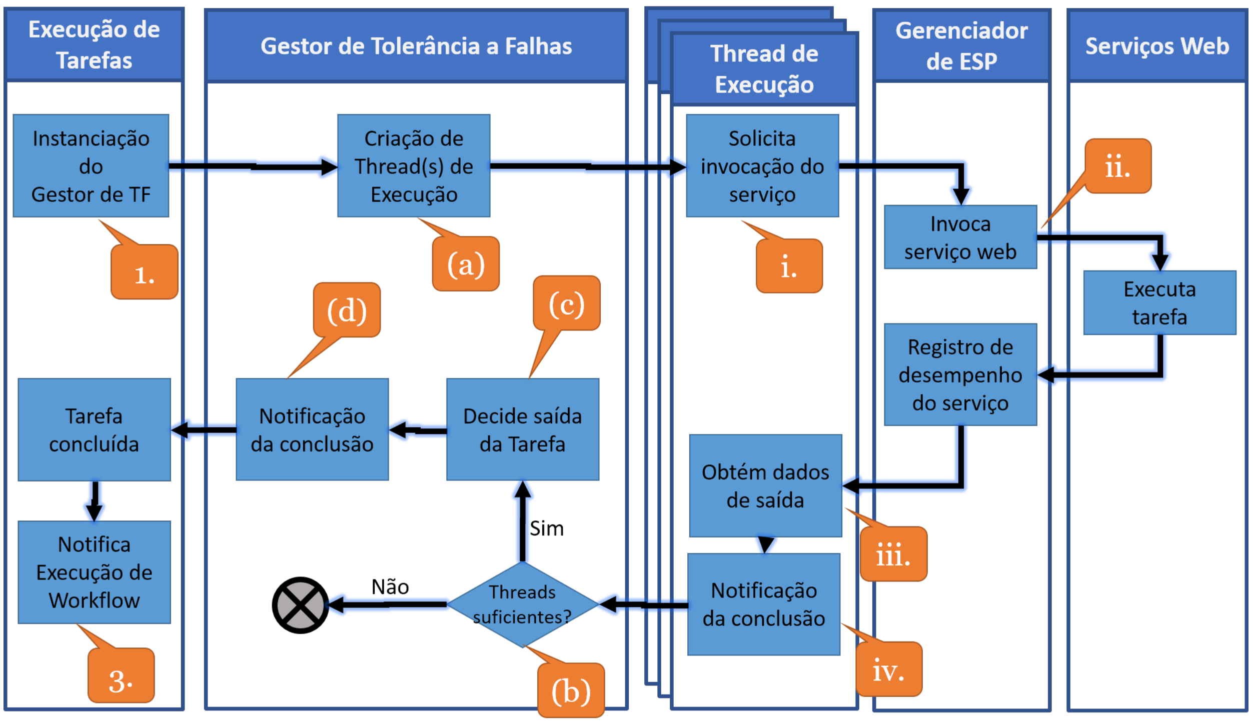


Figure 4.8: *Fluxo para realização tolerante a falhas de tarefas*

2. Finally, the `Task Execution` module notifies the `Workflow` module of the task's completion so that it can proceed with the remainder of the composition.

4.4.2 Task Sequencing and Inter-Services Communication

In the workflow, each task has a set of dependencies (tasks that need to be completed prior to execution), a set of inputs and an output. Each input and output is identified by an unique ID within the PRCGM.

It is through this informations that the actors manage and mediate the temporal sequencing and the communications between the tasks. Every input of a task necessarily has been produced by one of the task's dependencies or has been informed by the user as an input to the composition.

All of these inputs and outputs in the composition are stored by the actor during the execution of the workflow. Whenever a task is about to be initiated, the actor gathers its inputs and forwards it to the web service. Upon receipt of the produced output, the actors stores it, marks it with its identifier and triggers the execution of tasks which were dependending on the conclusion of that task to be initiated.

Therefore, it is through this input/output identification and storage that actors are able to mediate the communication between services.

4.4.3 Actors Execution Coordination

The execution coordination between the actors is performed via the delegation process and task completion notification. Eventually, while performing its role, an actor might need to ask another actor to perform some task(s). This solicitation takes place through a delegation.

However, the mere delegation is not enough to ensure the proper execution and task sequencing. Therefore after each task completion, the `Workflow Manager` verifies all the tasks that depend on the recently completed task and notifies their actors of the completion, eventually passing them any necessary input data allowing for more distributed coordination. For instance, a task that depends on two tasks, performed by two different agents cannot be started at the time of the first delegation arrival. It should wait for another delegation request from the other actor indicating that both tasks are completed before being able to gather the necesary input and before all dependent tasks are completed.

As stated in Section 4.3.2, all of this communication is performed using the SOA architecture, thus facilitating the data transfers and the actors independency.

To perform a delegation, the originating actor invokes the method `playRole` from the delegate actor, passing the decided workflow and any necessary input as a parameter. Similarly to what has been done wih the services, during the delegation process the actor also monitors and probes the correct execution of its partners and may, eventually, detect failures in the delegate actor. If such is the case, a recovery process may be triggered. This process of fellow actors monitoring and recovery will be throughly described in Section 4.5.3.

4.5 Fault Tolerance

As illustrated in Figure 4.4 and discussed so far, actors interact with their equivalent service pool and among themselves to perform the dynamic composition. However this is not their only role in the choreography, they are also responsible for implementing the fault-tolerance mechanisms chosen at the planning stage in accordance to the PRCGM tasks fault-tolerance annotations.

Such responsibility is placed in the `Task Execution Manager` module. During this Section we will categorize the faults handled by the IMPROV architecture, then explain how each of these are handled during execution.

4.5.1 Fault Categorization

Since we are dealing with unmodified web services (`Requirement 1`), the failures types that we may consider are severely restricted by the observable events in SOA itself. This means that the failure categorization cannot be grouped by their actual causes (hardware failures, software bugs, network errors, etc) but by the actual observable event from the service invocator's (in our case, the actor) point-of-view. To better understand this peculiarity let's consider a content fault. From the service consumer's perspective a content fault derived from a bug cannot be diferentiated from another fault due to a network error or a bit change in the hardware. Because the service is unmodified and unspecialized the actor has no power to interfere directly on the service's inner workings and therefore on the fault's actual origin. From the actor's perspective all content failures have the same effect and the same handling mechanisms for all of the afore mentioned situations. Regardless of its origin, in this case the failure is considered simply as an `Incorrect Answer`.

The full scope of the considered failures in this Thesis are presented on Table 4.1. According to the failure categorization defined in Avižienis et al. [ALRL04], the failures have been classified by the component on which they occur (`web services` or `actors`), their domain (`time` or `content`), their detectability and degree of impact (`low`, `medium`, `high` or `catastrophic`)

The first failure type is the unavailability of a replicated service. This kind of failure occurs whenever a service becomes unavailable. However, in this situation there are other replicas or resources able to deliver the same features, thus incurring simply in a service switch. At most, there will be some requests that will have to be replanned should the unavailability of this service impact on the achievability of their root goal.

The second kind of failures considered and treated by IMPROV is similar to the first in the sense that occurs due to a service becoming inavailable but in this situation it is far more harmful because the unavailable service has no substitute candidate to replace it. In this situation, all of the requests depending on such service for their executions will have to be replanned and some may even become unachievable.

The third failure category is the only one in the content domain. For some more data-sensitive tasks, the PRCGM designer may opt to use a voting mechanism to diminish the possibility for an incorrect response. Whenever that occurs, content failures may be detected. An incorrect failure, from IMPROV's and the actor's perspective, occurs whenever a service included in an NVP scheme

Failure	**Component**	**Domain**	**Consistency**	**Detectability**	**Impact**
Unavailable Replicated Service	WS	Time	Yes	Yes	Low
Unavailable Unique Service	WS	Time	No	Yes	Medium
Incorrect Response	WS	Content	Yes	Yes[2]	Low
Irresponsive Actor	Actor	Time	No	Yes	Low
Unreachable Actor	Actor	Time	Yes	Yes	High
Endpoint Unavailability	Actor	Time	Yes	Yes	High

Table 4.1: *IMPROV's Considered Faults Categorization*

disagrees from the majority of the others. If such situation occurs, IMPROV interprets the most voted response as the correct one and any services that did not produce that very same response will be attributed a content-failure.

The fourth kind of failure considered does not regard the web services anymore but the actors themselves. In IMPROV each actor serves as a monitoring and recovery mechanism for all adjacent actors. An Irresponsive Actor failure signifies that a communication attempt has failed. The actor was somehow unable to receive a message sent from an adjacent actor, which will trigger a Irresponsive Actor fault on the emissary actor. This failure is on the time domain, is detectable by the emissary actor and has a low impact on the architecture because it will simply mean that the emissary should retry sending the message.

The fifth kind of failure occurs when the emissary actor has already retried several times to reach the adjacent actor without success. In this situation, the emissary actor throws a Unreachable Actor fault. These faults are in the domain of time and have a much deeper and long lasting impact on the architecture. If an actor has been deemed offline, the emissary will start a recovery process which will bring up a replica of the unavailable actor in the same resource as the emissary actor. Next, all of the unavailable actor's adjacent actors will be informed of its unavailability and instructed to re-route their requests to the newly instantiated replica. This process however diminishes the composition's resources, with one resource providing two actors. It also augments the actors' concentration per resource, meaning that a failure in another resource may cripple several actors at the same time. Therefore this is one of the most costful failures in the IMPROV architecture.

The sixth and last kind of failure considered by IMPROV is the unavailability of the endpoint providing actor. In this situation the actor that goes offline is the very same actor which publishes the composition's public interface. This kind of failure is considered catastrophic as there is no way of notifying all clients of a change in the composition's endpoint. If the WSDL interface cannot be found on the previously advertised location, the composition as a whole is rendered offline. In particular, we consider that raising the same WSDL interface on another location would actually create a new composition, even in the case of reusing the previously deployed actors.

4.5.2 Task-level Fault-Tolerance

Considering the presented failures and - again - `Requirement 1`, we have used in this work only fault-tolerance techniques that would raise the overall reliability levels without any intrusive mechanisms, like the ones that would be necessary to implement a task-level checkpoint technique or to ensure the maintenance of an secondary set of services in the same state as the primary ones to implement a `Passive Replication` mechanism.

Therefore, the techniques currently available in IMPROV are `Retry`, `Active Replication`, `Retry on Alternate Resource`, `Voting` and `Voting with Active Replication`. Although IMPROV currently implements only the aforementioned techniques, its internal architecture and the `Task Execution Manager` implementation allows for non-intrusive extensions to be added on a later time, as long as the SOA characteristics are left untouched.

Each of these techniques are implemented by the Task Manager module by conveniently placing service invocations on the `ESP manager` and awaiting their replies. The active replication simply places requests on all available services and returns the first replied response. The Retry technique places a single invocation and, should it fail by timeout, places another one on the same service. The Retry On Alternate Resource works similarly but places the new attempted request on another service, instead of the same. The Voting mechanism places requests on all the services, awaits all the replies and then decides on the most voted answer. Finally Voting with Active Replication places requests on all services but awaits for only half the replies before deciding on the most voted answer.

4.5.3 Actors' Fault-tolerance mechanism

Whenever an actor is initialized, they go through a setup phase in which they may publish to their adjacent actors their own behavior, as described in their PRCGM and also the list of adjacent actors. They also publish in their control WSDL interface a method called `updateNeighbor` through which another actor may upload or update their behavior on the event of a change.

This structure aims at maintaining at all actors all the information needed to create a replica of an adjacent actor. This way, in the case of any actor becoming unavailable the adjacent actor is capable of creating a replica of the original actor and notify its adjacent services of the change. Whenever an actor sends a message to another actor, it awaits for a receipt confirmation. If the confirmation does not arrive, an Irresponsive Fault is thrown, and then two new attempts are made. Should these attempts fail as well then an Unavilable Actor Failure is thrown and the recovery process begins.

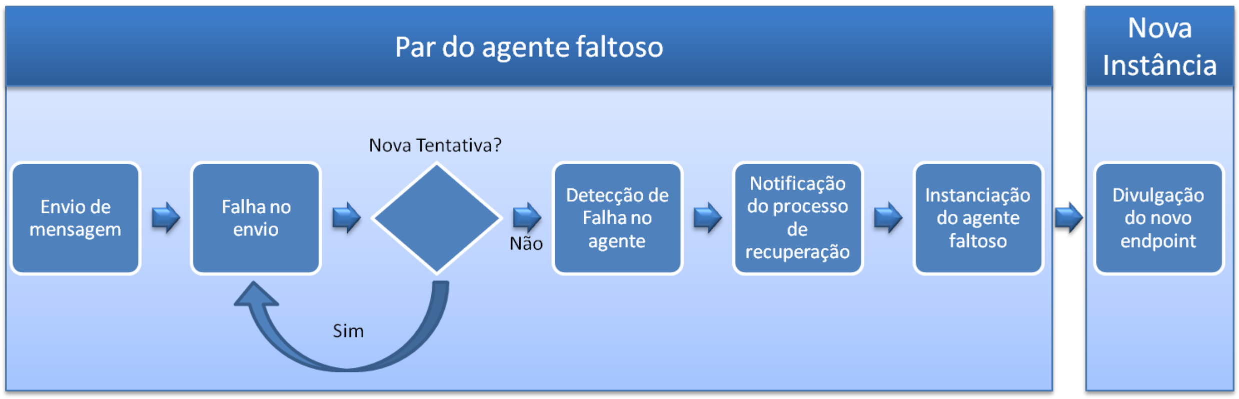

Par do agente faltoso
Nova Instância
Envio de mensagem
Falha no envio
Nova Tentativa?
Sim
Não
Detecção de Falha no agente
Notificação do processo de recuperação
Instanciação do agente faltoso
Divulgação do novo endpoint

Figure 4.9: *Processo para identificação e recuperação de agentes inacessíveis ou indisponíveis*

The recovery process is illustrated in Figure 4.9 and consists of the following steps:

1. **Irresponsive Actor Fault**
   The confirmation of receipt of a message sent to an actor is not received.

2. **New Attempts**
   The emissary actor tries to send the same message again another two times

3. **Unavailable Actor Fault**
   After three failed attempts of communication, the emissary actor declares the receiving actor unavailable.

4. **Recovery Process Notification**
   The emissary actor notifies the failed actor's neighbors that the actor was declared unavailable and that a recovery process will be started.

5. **Faulty Actor Instantiation**
   The emissary actor uses the faulty actor's PRCGM data to create a new instance of it.

6. **Endpoint Publishing**
   After instantiating the faulty actor, the emissary notifies all neighbors of the new endpoint on which the newly instantiated actor may be found.

As an example, let's imagine a faulty actor $A1_{faulty}$ that interacts with actors $P1$, $P2$ e $P3$. During its setup phase, $A1_{faulty}$ has sent its PRCGM data to $P1$, $P2$ and $P3$. However, at any given time, actor $A1_{faulty}$ becomes unavailable or unaccessible by its peers (for instance, because of a hardware failure on its resource). Eventually $P2$ sends a message to $A1_{faulty}$ but $A1_{faulty}$ does not acknowledges the message receipt. $P2$ sends the message two more time, without success. At this point, $P2$ declares actor $A1_{faulty}$ unavailable. $P2$ notifies $P1$ and $P3$ that actor $A1_{faulty}$ is unavailable and that it will be reinstantiated in a new resource. $P2$ creates another instance $A1_{recovered}$) in some available computational resource and then $A1_{recovered}$ notifies $P1$ and $P3$ of its new endpoint effectively restoring the service composition to a working condition.6

4.6 Compositions Lifecycle and Evolution

The lifecycle of an adaptive and distributed service composition using the IMPROV architecture is a long lasting one, with several stages. These stages are represented in Figure **??**.

Every composition starts its lifecycle with the definition of a root goal, derived directly from the goal-based requirements engineering process [ADG10]. This initial process is illustrated in the lifecycle by the means of the first two ativities: Root Goal Specification and Actor Behavior Definition. Initially, to specify the behavioral traits of an actor and, subsequently, of the composition, the compositiondesigner must identify the root goal that is to be pursued by the composition. Next, the actor's behavioral traits must be specified by the means of its PRCGM model and, consequently, by mapping the possible alternative strategies to reaching each specified goal, the contexts that may affect the environment and the necessary actor behaviour and the necessary web services that implement the tasks described in the process.

After defining the actors' behaviours, they must be deployed on the environment. At this point, an instance of each actor is instantiated in an appropriate computational resource and executed. From this point on, that actor is available for answering requests from other actors or from users.

The composition now moves on to a more operational phase of its lifecycle. It is now ready to handle user requests. The next step in the lifecycle and in the actors' history is the handling of these user requests with multiple possible QoS level requirements. For each of these requests, the actor first evaluates the possibilities and alternative strategies to handle the user request, maybe by itself but probably by interacting with other actors to define an execution plan.

Finally, the user request is performed in accordance to the decided plan and the services are monitored, thus improving the actor's knowledge of their behavior and updating the agents behavioral definitions to work with this newly acquired knowledge.

4.7 Project Decisions

During the development of IMPROV we have hit a few speedbumps, specially considering the ambitious goal we have set to achieve. In this Section we present some of these hard choices and concessions that were needed and expose our reasons to make such choices.

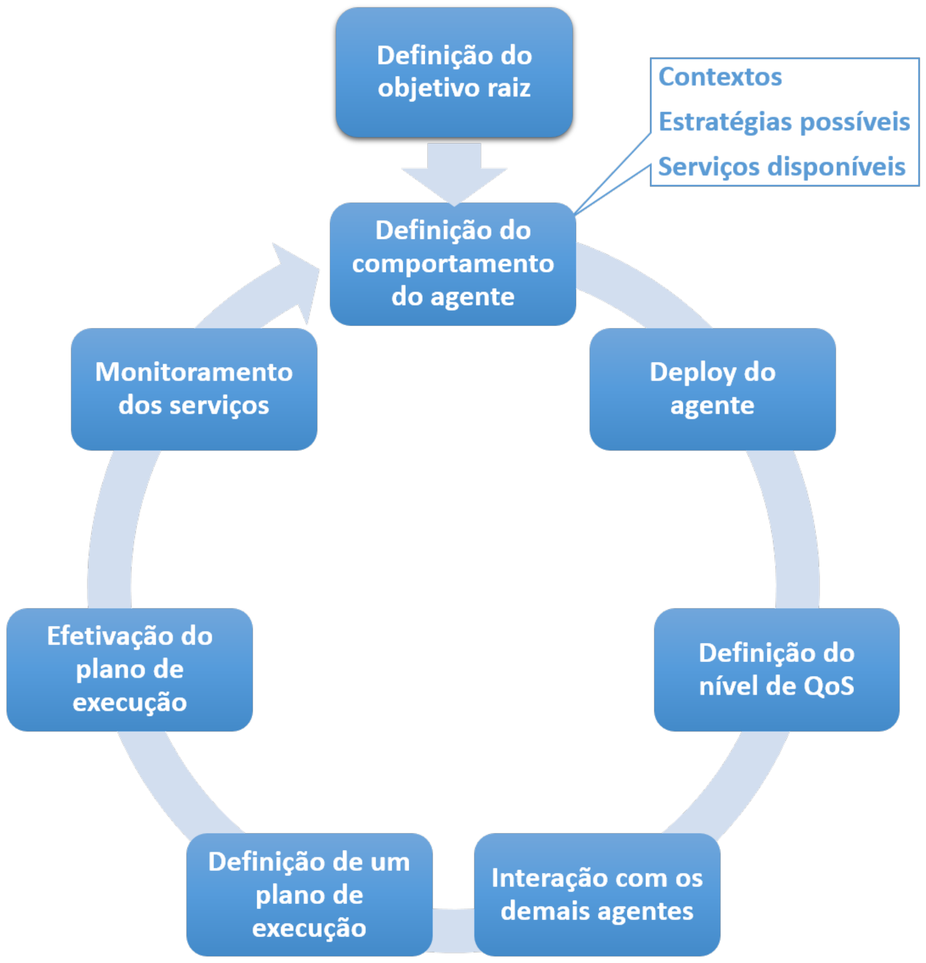

Definição do objetivo raiz
Contextos
Estratégias possíveis
Serviços disponíveis
Definição do comportamento do agente
Deploy do agente
Definição do nível de QoS
Interação com os demais agentes
Definição de um plano de execução
Efetivação do plano de execução
Monitoramento dos serviços

Figure 4.10: *Ciclo de vida de uma composição sensível a contexto*

Only one composed QoS constraint Some of the guiding principles in IMPROV development were: (1) a highly adaptable service composition - adaptive service selection, topology and fault tolerance techniques - that can respond to dynamic changes in the environment; (2) the preservation of the actors' autonomy with regard to their behaviour. From the first principle, we may derive the need for runtime adaptation and, consequently, of a low planning overhead for the algorithm. From the second principle we derive the notion that the actors must be able to decide their own behaviour without a centralized supervenient entity's point-of-view. Therefore, using a multiple selection criteria for composed QoS levels was deemed too heavy-weight a solution to be applicable. Multi-criteria QoS restrictions have much higher complexity than the QoS-Constrained algorithm. If we were to employ such algorithms to decide the actors' behaviours we would not be able to do so in a localized manner - thus infringing the actors' autonomy - nor with a greedy approach - thus providing higher complexity levels and much higher execution overhead. Because of these reasons, we opted to restrict the composed QoS levels to a single one. However, as explained in Section 3.7.1 multiple individually evaluated (filter) restrictions may be employed.

Web Service Discovery Although IMPROV foresees dynamic adaptation due to service discovery, there are already many published papers on this issue and we chose not to focus on that particular aspect, leaving us to give more attention to the adaptivity aspect of the architecture.

Monitored QoS Levels To monitor unmodified web services *in-the-wild* behaviour one must consider the services' loose coupling and location transparency. Therefore, similarly to the fault-tolerance techniques that needed to be non-intrusive, the service monitoring mechanism also has to deal only with non invasive measurements. Thus, a thorough monitoring of their more implementation- and resource-centric metrics is not feasible. The non-instrusive WS monitoring is performed only for the metrics that can be observed from the actor's point-of-view and extracted from its interactions with the service: response time and reliability. These are currently the only two monitored QoS levels. The remaining QoS levels are treated as fixed or externally updated.

Chapter 5

Validation

5.1 Pragmatic Model and Achievability Algorithm Evaluation

In this section, we experimented on the algorithm to find out its effectiveness and to evaluate its usefulness in finding achievable plans and its scalability as to assert its practical application in large scenarios, both at design-time and runtime using the Goal-Question-Metric evaluation methodology [BCR94].

GQM is a goal-oriented approach used throughout software engineering to evaluate products and software processes. It assumes that any data gathering must be based on an explicitly documented logical foundation which may be either a goal or an objective.

5.1.1 GQM Validation Plan

The Goal/Question/Metric (GQM) [BCR94] methodology. GQM is a well-established top-down goal-oriented approach used throughout software engineering to evaluate products and software processes. Its main premise is that any data gathering must be based on an explicitly documented logical foundation which may be either a goal or an objective.

In short, GQM's first step is to define the high-level goals to be reached. From each of these goals derives several quantifiable questions that need to be answered to consider that goal fulfilled. Finally, for each question, the necessary measures for duly assessing the evaluation [BCR94]. These questions identify the necessary information to achieve the goals while the metrics define the operational data to be collected to answer each question. These goals are tailored in such a way that they summarize the macro objectives of our architecture and implemented mechanism. Following these guidelines, our GQM model was designed to verify that all our goals were methodologically tested and assured[1]
.

In such a methodology, we started up by setting the three main goals of our evaluation : (I) verifying the usability of the model for the MPERS case study at runtime; (II) verify that algorithm's performance allows it to also be used also at runtime for larger models and; (3) define whether the model can be used at design-time to pinpoint context sets in which the model may be, per design, unachievable. These three goals were the foundation on which the GQM validation plan was based.

5.1.2 Experiment Setup

The experiment setup consisted in evaluating the Pragmatic model and the algorithm's capability to support design-time and runtime usage.

These parts and their evaluations were engineered to provide the metrics demanded by the GQM plan (Table 5.2).

[1]Although the same GQM model may be used for comparing two equivalent approaches no similar work - fault-tolerant choreographies mechanism - was found. The closest in nature to our proposal was [**?**] which used GQM method to derive the metrics for evaluating the successful adoption of SOA in a project however no GQM plan was presented.

Goal 1: PPA's runtime usage capability in the MPERS case study	
Question	Metric
1.1 How long would it take for the PPA algorithm to come up with a plan for the MPERS case study?	Execution time
1.2 How reliable are the plans provided by PPA for the MPERS model?	% of correct answers

Goal 2: PPA's runtime usage capability for larger models	
Question	Metric
2.1 How does the PPA algorithm scale over the amount of goals in the model?	Execution time
2.2 How does the PPA algorithm scale over the amount of contexts in the model?	Execution time

Goal 3: Algorithm's design-time usage capability	
Question	Metric
3.1 How long would it take to cover all context sets for increasingly large models?	Time elapsed

Table 5.1: *GQM devised plan*

The first evaluation goal is directed towards the MPERS case study and aims at a more comprehensive and detailed evaluation and it is intended as a "proof of concept". On this front we have evaluated the time to produce an execution plan and the reliability of the plans provided evaluation. In this evaluations, we have used the MPERS Pragmatic Goal Model as the input for the evaluation tool and measured the time for producing a plan and its adherence to the model's restrictions.

The second evaluation goal concerns itself with the scalability of the proposed model and its applicability on larger models, both in terms of goals and contexts amount. For this evaluation we have generated models with varying amounts of goals and tasks using each one the possible annotations to evaluate the impact on using such annotations - when compared to one another - and to determine the worst case scenario. This is meant to ascertain the planning overhead on the model. This information also gives designers a baseline on which to base their decision on the applicability of the model to perform runtime decisions.

Finally, the third evaluation front is concerned with the usability of the algorithm as a design-time tool to pinpoint scenarios in the Pragmatic GM which may be unachievable by design, given the goal interpretations, tasks' quality levels provided under specific contexts. For this goal, we have evaluated the time spent to sweep all context combinations in randomized models with 10 contexts and sizes ranging from 500 to 6,000 nodes.

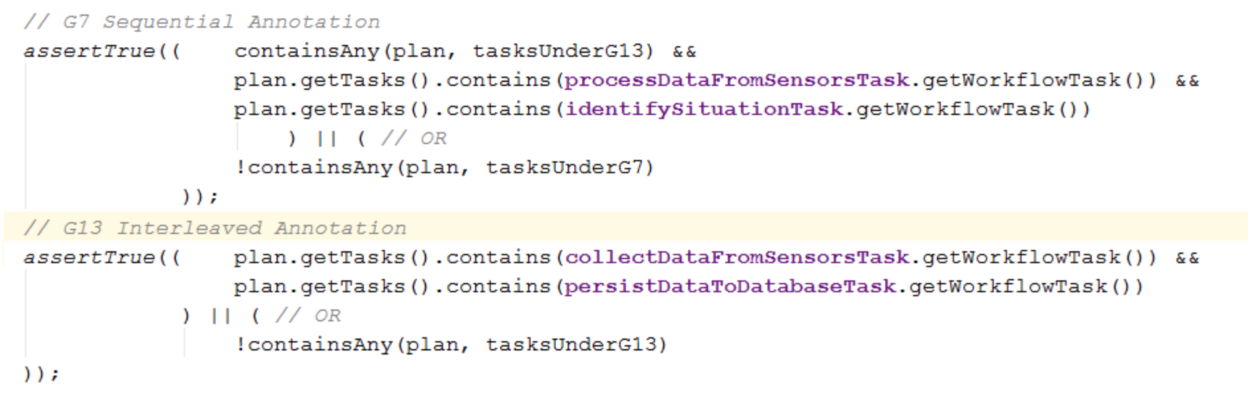
The PPA algorithm from Figure 1 was implemented[2] using Java Oracle JDK 1.8.0_92 and all evaluation tests were performed on a Dell Inspiron 15r SE notebook equipped with a Intel Core i7 processor, 8GB RAM running Ubuntu 16.04, 64 bits and kernel 3.16.0-29-generic.

All experiments to evaluate the correctness and performance of the algorithm were implemented as automated tests under Java's JUnit framework. This guarantees that the evaluation is both effortless and repeatable.

5.1.3 Goal 1: PPA's runtime usage capability in the MPERS case study

For this goal, we evaluated the average time to produce an answer in the presented MPERS scenario (Figure 2.2) and the reliability of the engineered plans.

[2]Source code, evaluation mechanisms, and complete result sets are available at https://github.com/felps/PragmaticGoals/tree/RuntimeGoalModel. Accessed on 2016/07/30



```
// G7 Sequential Annotation
assertTrue((    containsAny(plan, tasksUnderG13) &&
                plan.getTasks().contains(processDataFromSensorsTask.getWorkflowTask()) &&
                plan.getTasks().contains(identifySituationTask.getWorkflowTask())
                    ) || ( // OR
                !containsAny(plan, tasksUnderG7)
            ));
// G13 Interleaved Annotation
assertTrue((    plan.getTasks().contains(collectDataFromSensorsTask.getWorkflowTask()) &&
                plan.getTasks().contains(persistDataToDatabaseTask.getWorkflowTask())
            ) || ( // OR
                !containsAny(plan, tasksUnderG13)
));
```

Figure 5.1: *Assertions to verify the generated plan's adherence to the annotations' semantics*

Question 1.1: How long does it take for the PPA algorithm to come up with an execution plan for the MPERS case study? To evaluate the time for the algorithm execution on the CGM of Figure 2.2, we executed 100 iterations of the algorithm for each possible context set. The results showed that the algorithm took, in average, less than 1 ms to be executed.

This evaluation was performed by executing the algorithm over the MPERS model 100 times for each possible context set, summing up to a total of 409600 repetitions.

The average time for coming up with a plan for the MPERS case study was 0.34 ± 0.09 ms. Out of the 409600 measurements, there were however 62 outliers (0.015%) with execution times of more than 10 ms. These outliers points are still under a second and are likely due to background tasks in the machine, which was not dedicated.

Question 1.2: How reliable are the plans provided by PPA for the MPERS model? To validate the correctness of the plans we have, for each context set for the MPERS, identified all the inapplicable tasks, all restrictions imposed by the goal annotations' and the composite QoS constraints. Then, we formally coded these restrictions through 42 assertion statements, similar to the ones presented in Figure 5.1, throughout the test so as to guarantee that none of the restrictions were disrespected in any of the outputted plans.

Finally this test was performed for every possible context set and none of the coded restrictions were infringed.

Analysis of the results With regard to the MPERS case study, the proposed algorithm have been shown to be efficient. The planning stage for this scenario took around 3 milliseconds to be computed and none of the 4096 executions, effectively sweeping all the possible context sets, triggered any of the several assertion statements introduced in the tests.

It was certainly expected - given the deterministic nature of the algorithm - that all the responses were valid. However the level of efficiency in solving this problem was remarkable.

5.1.4 Goal 2: PPA's runtime usage capability for larger models

The second goal from the GQM validation plan aims at defining if the model and algorithm proposed are suitable for usage in larger models. To uncover some information regarding this objective, we have stated two questions on the scalability of our proposal: how does it scale over the amount of goals in a model and how does it scales over the amount of contexts in a model.

This investigation is important considering the objective of using this model and this algorithm to engineer - at runtime - an execution plan able to achieve the CGM's root goal under the current context and current QoS constraints. To accomplish this goal - and this is a pragmatic goal itself - it is not sufficient to engineer the plan but it needs to done in a reasonable amount of time so that it will not seriously impact the response time.

Question 2.1 and 2.2: How does the PPA algorithm scale over the amount of goals and contexts in the model? To answer questions 2.1 and 2.2, we have performed a scalability analysis on Pragmatic GMs of different sizes, in terms of goals and contexts amounts, for each one of the possible goal annotations.

For each of the available annotations, we engineered and performed a series of evaluations.

A pragmatic model with size ranging from 500 to 10.000 (in steps of 500 nodes) and contexts ranging from 1 to 20 containing goals annotated with the selected annotation would be generated. Then, for each of these models, we executed the `isAchievable` method 50 times and measured the total execution time. Finally, the test outputted the average execution time per method invocation. The resulting average times are presented - one for each annotation - in Figure 5.2.

In these graphs, the x- and z-axes represent the model-size and the amount of contexts used in each execution respectively, *i.e*, the independent variables. The y-axis represent the time for engineering an execution plan, *i.e*, the experiment's dependent variable. In all six experiments, the average time for engineering the plan did not go over 5 seconds.

Also, in the plotted graphs we are able to glimpse into the expected complexity of $O(m * n)$ over the model size m (x-axis) and over context sizes n (z-axis).

Analysis of the results In general, Figure 5.2 shows a good behaviour of the algorithm. The growth ratio for the algorithm's execution time grows linearly over the amount of goals and contexts. None of the annotations had issues in elaborating a plan on larger models, although their performance varied in different degrees. The behaviour of the Sequential annotation was the worst of them all with an execution time of a little under 4.5 seconds for models with 10,000 nodes and 10 contexts. Still this was considered an affordable execution time for the planning stage of such large scenario in terms of goals, tasks and contexts.

5.1.5 Goal 3: Algorithm's pinpointing unachievable context sets capability

Finally, the last goal of the evaluation is to show the applicability of this algorithm to benefit software designers. More specifically, we want to know if it can answer the following question: "Given that the system has been modelled as a Pragmatic GM, is it possible to use this algorithm pinpoint context sets in which such system would inherently be unable to reach its goals?"

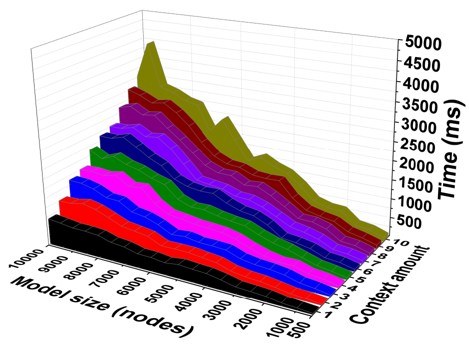
To deal with this we have implemented one last test which would generate 600 random models varying from 100 to 6000 nodes with a fixed set of 10 contexts. Then, the algorithm was executed for every one of the 2^{10} possible context sets and the time spent to do so was measured. The results are now presented in Figure 5.3, with each point representing the average time to sweep all the context sets in 10 randomized models.

Analysis of the results As shown in Figure 5.3, on smaller models - up to 300 goals - the algorithm was able to fully analyze the 32768 context sets within the 10 seconds deadline. On larger models - up to 5000 goals - the algorithm was able to analyse around 40% of the combinations. Even at the limit, with models of 10000 nodes, it was able to cover more than 25% of the possible combinations. This result suggests that even for models with up to 10000 goals and 20 contexts the complete analysis can be performed within a minute.

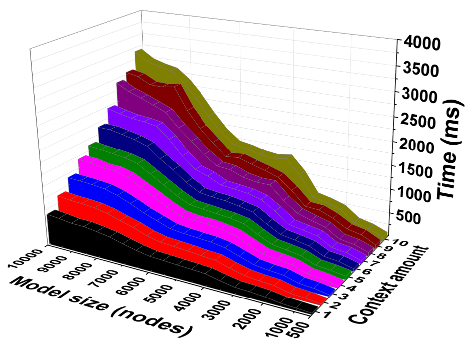
5.1.6 Discussion of the results

As stated in the GQM plan (Table 5.2), the experiments had three main goals: (1) determine if there is the necessity for an algorithmic approach, (2) determine if the algorithm could be used at runtime to find a suitable plan for the current context and; (3) whether it could also be used for pinpointing context combinations in which there is no applicable goals/tasks combination sufficiently good to satisfy the pragmatic goals' requirements.

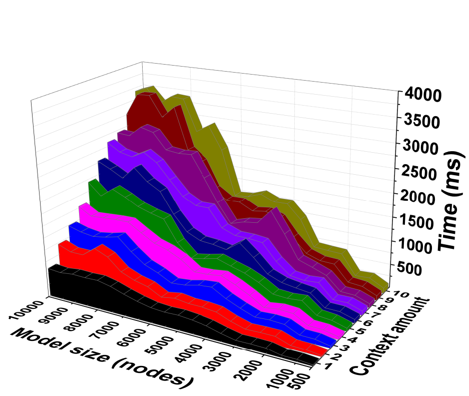
For the first goal, the results have corroborated with the understanding that the Pragmatic CGM complexity is far too great to be dealt simply by the human perspective.



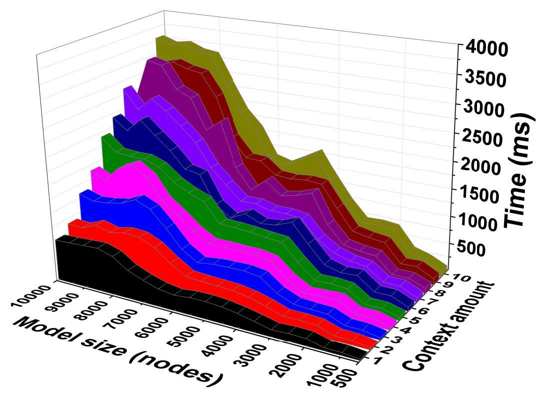
(a) *Sequential*



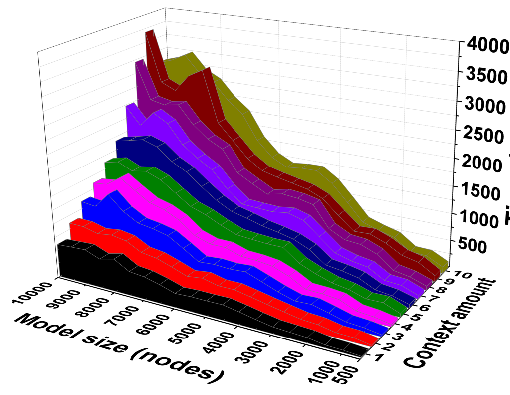
(b) *Parallel*



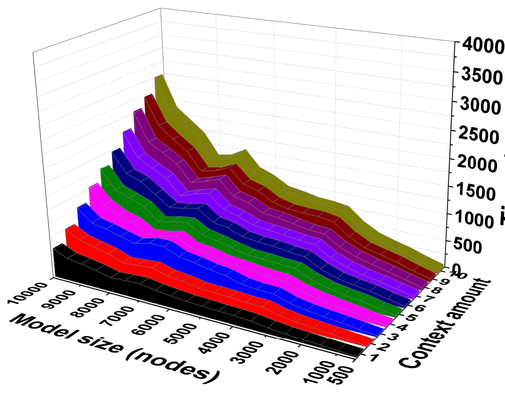
(c) *Sequential Iterative*



(d) *Parallel iterative*



(e) *Alternative*



(f) *Try*

Figure 5.2: *Algorithm's scalability over the model size, in number of nodes, and context amount*

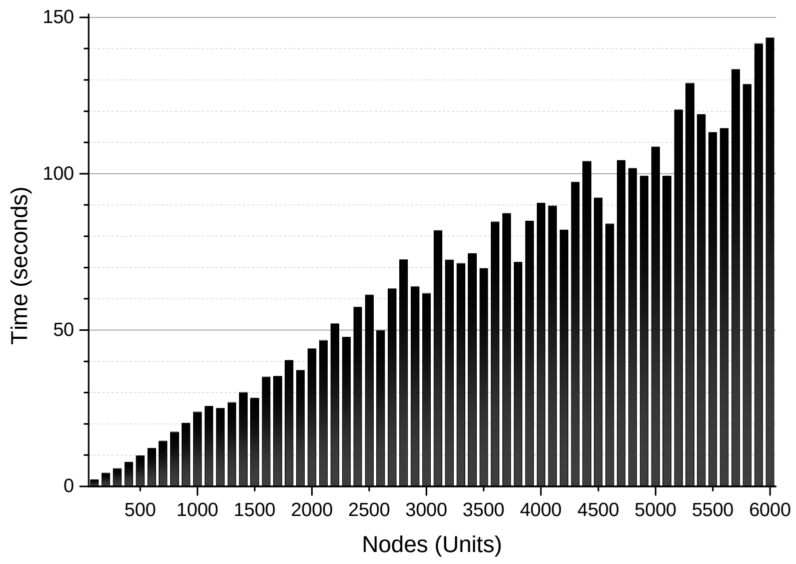


Figure 5.3: *Time spent to run PPA algorithm for all the possible context sets (average execution time over 10 different models)*

On the other hand, the algorithm proved itself as a much faster and deterministic alternative. While the volunteers took up to 17 minutes to provide an answer with 73.19% reliability, the algorithm produced an answer to the same problem in under 1 millisecond. Also, since it is a deterministic algorithm, the produced answers are always valid. Even if considering very large improvements on the human performance, the algorithmic approach would, most likely, still largely surpass human performance.

As it can be perceived from the metrics presented, human judgment takes a lot of time and it is also very unreliable whereas the algorithm provided correct answers in a very timely manner, thus allowing us to state that an algorithmic approach is apparently needed.

Though the comparison between volunteers and an algorithm may seem unfair, the goal of this comparison is to determine the necessity of an algorithmic approach. Being so and as expected, the algorithm has proved itself much better both in terms of efficiency and efficacy as well as subsiding the desired hypothesis that an algorithm is indeed necessary.

In this section, we compare the performance and accuracy of the human and algorithmic approaches.

For the second goal, we aimed to evaluate whether the algorithm was suitable for execution at runtime. This meant verifying two main aspects: that the algorithm would efficiently execute even with large Pragmatic CGM models and within a reasonable time to not affect the response time. We performed this analysis for random models (Figures 5.2a and **??**) which may have incurred in the observed high variability for the average case. However, an upper boundary was also set with the worst case evaluation (Figure **??**).

With regard to the first aspect, Figure **??** shows that the algorithm's execution time grows linearly over the amount of goals as well as over the amount of contexts in the Pragmatic CGM model, both in the average and in the worst case scenarios.

The second aspect can also be derived from Figure **??**. As it shows, even when considering the worst case scenario , the time for evaluating a model with 10000 nodes and 20 contexts was 1081 ms. As a matter of comparison, the default setting of Axis2 (Java's API for web services) for a read timeout is set to be 300 seconds[3]. Therefore, we see this as an acceptable result for the purposes of showing that the algorithm's performance is enough not to severely impact runtime.

Finally, for the third goal we evaluated how long the algorithm took to sweep all context com-

[3]http://www-01.ibm.com/support/knowledgecenter/SSD28V_8.5.5/com.ibm.websphere.nd.doc/ae/rwbs_jaxwstimeouts.html. Accessed on January 14th, 2015

binations. Given the results from Figure 5.3, we can state that for models with up to 10000 goals and 15 contexts, it is actually possible to evaluate all context set combinations within one minute. Thus, the proposed algorithm can indeed be used on a Pragmatic CGM to pinpoint unachievable scenarios with up to 15 contexts, which can then be scrutinized by the CGM designer in order to correct or document it.

5.1.7 Threats to validity

Construct validity concerns establishing correct operational measures for the concepts being studied. This was minimized by the usage of the GQM methodology to lay out the evaluation plan. Firstly, the goals that needed to be achieved were laid out; for each goal we envisioned the questions which needed to be answered and only then the metrics were defined with these questions in mind.

Internal validity concerns establishing a causal relationship, whereby certain conditions are shown to lead to other conditions. The experiment setups guaranteed that all computer experiments and evaluations were performed in the same resource and environment. At each evaluation, up to two controlled variables were used.

The main benefits of the algorithmic approach to tackle with contextual goal-models and pragmatic CGMs is the ability to pinpoint inconsistencies and scenarios under which the CGM's root goal is unachievable, even after considering all possibilities. Doing so manually is a very laborious task due to the exponential explosion of the solution space domain. Take the CGM presented in Figure 2.2 as an example. There are 12 different context annotations ($C1 \dots C12$) and 7 OR-decompositions (5 with 2 refinements and 2 with 4 refinements). This builds up to 2^{12} or 4096 possible context combinations. In addition, the OR-decompositions create 512 CGM variants.

External validity concerns establishing the domain to which the findings can be generalized. We conducted the assessment in the context of the Mobile Personal Emergency Response, which is specific to a given domain. Nevertheless, future work should assess this in different and real-life domains and with higher number of goals, contexts and quality constraints. In particular, this would help to further assess the scalability we observed. Furthermore, other non-functional properties could be evaluated and we consider it as future work.

On the other hand, with the advent of our proposed algorithm the same solution space domain of all context combinations can be analyzed in under a second. This means that the requirements engineers, stakeholders and domain experts may focus on the context combinations under which the root goal is not achievable. Either to modify the CGM and correct inconsistencies, enhance the quality provided by the tasks or to review the quality constraints imposed by the pragmatic goals.

5.2 Fault-Tolerance Mechanism effectiveness

On another evaluation front, we need to evaluate the benefits provided by the fault-tolerance mechanisms implemented by IMPROV.

The GQM plan for such evalation is defined and presented in Table 5.2. The experiments[4] to evaluate each question are described in the following sections. Every presented measurement is the average of 1000 measures.

The Systems' Laboratory at IME-USP provides publicly available services with specific and predefined reliabilities. These services are located in an application server at the University of Sao Paulo. This server has one Intel Core^{TM} i7-2700K processor at 3.5GHz and 8MB cache, 16GB RAM and 1TB of local storage.. For all of the following experiments, these services were used to create the choreographies.

Each instance is invoked sequentially by the choreography agents, located at *Universidade de Brasília*. The two sites are connected via Internet with an average bandwidth of 42 Mbits/sec and 28ms RTT.

[4]The evaluation mechanisms were implemented as automated tests and the complete result set as well as the code for implementing them are available at https://github.com/felps/CloudTolerance

Table 5.2: *GQM devised plan*

Goal: (1) Choreography-Enabling	
Question	Metric
1.1 In a failure-free environment, how many requests are replied properly in a given choreography?	Observed failed requests
1.2 How does the choreography scale over the number of *involved services* regarding *performance*?	Composition's average response time
1.3 How does the choreography scale over the number of *exchanged messages* regarding *performance*?	Composition's average response time

Goal: (2) Enhance Reliability	
Question	Metric
2.1 What is the impact of the services' reliability on the overall system *reliability*, regarding *fail-stop failures*?	Observed failed requests
2.2 What is the impact of the services' reliability on the overall system *performance*, regarding *fail-stop failures*?	Composition's average response time
2.3 What is the impact of the services' reliability on the overall system *reliability*, regarding *incorrect responses*?	Observed failed requests
2.4 What is the impact of the services' reliability on the overall system *performance*, regarding *incorrect responses*?	Composition's average response time
2.5 What is the impact of a fault on a choreography agent itself on the overall system *performance*?	Agent's recovery time

5.2.1 Goal 1: Enabling Service Choreographies

To evaluate the first goal - enabling the composition of unmodified services into a choreography - we dealt mainly with three variables: services involved, messages exchanged and WS reliability. From those variables we evaluated the behavior of the composition with regard to performance and overall composition reliability.

Question 1.1 : In a failure-free environment, how many are replied properly in a given choreography? To evaluate the capability of duly composing several WSs into a choreography, we used a scenario with four services invoked sequentially. We performed 1000 choreography requests, using FTTs as well as with no FTT at all.

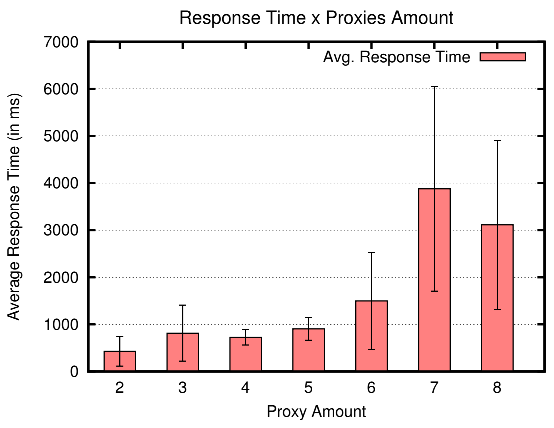
Since we are considering a failure-free environment for this experiment, in this scenario as long as the agents are able to compose the services in an effective manner, the foreseen denouement is that all requests will be replied correctly in spite of the usage or not of FTTs .

As expected, in this experiment all requests were properly replied, thus confirming the efficacy of the choreography agents in performing the WS composition.

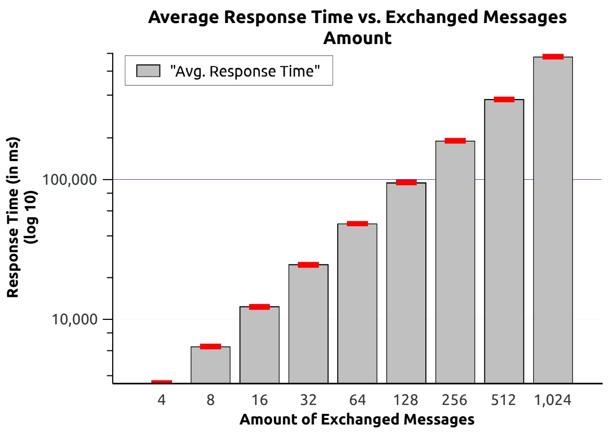
Question 1.2 : How does the performance decay when the number of involved services increase? In order to evaluate the decay of performance over the number of involved services, a series of linear choreographies, ranging from 2 (the smallest composition possible) to 8 services were used. In this choreography each service is invoked once its reply is forwarded as an input to the next agent. The following choreography agent will use it as input for the subsequent service, until the choreography is complete and the final result is returned to the user. All choreography agents resided in dedicated machines.

The observed results for this experiment are shown in Figure 5.4a. The observed results are within expectations, rising as the amount of invoked services increases. There was an unforeseen spike with 7 agents. However, this measurement also had the highest error margin, thus still being within expected values.

Question 1.3 : How does the performance decay when the number of exchanged messages increase? In order to evaluate the decay of performance over the number of exchanged



(a) *Response Time vs Involved choreography agents*



(b) *Average Response Time vs Amount of Exchanged Messages*

Figure 5.4: *Goal 1 evaluation graphs*

messages between the services, a simple choreography, with only 2 services was used. This choreography exchanged messages until a previously established threshold was reached. The threshold for the evaluations varied from 4 to 1,024 messages, doubling the amount at each step.

The observed results for this experiment are shown in Figure 5.4b. As it can be seen in the figure, the observed response time was linear with regard to the amount of exchanged messages between the choreography agents in the choreography.

5.2.2 Goal 2: Increase the composition's reliability with several FTTs

To evaluate the effectiveness of the proposed architecture, a set of services were developed so that their reliability regarding fail-stop and content failures is configurable. When instantiating the web service, the occurrence probabilities for both content and a fail-stop failures is passed as input to it.

Based on these reliabilities, for each processed request by the WS it would take a random number between 0 and 1. Should it be less than the fail-stop probability, the WS would halt. Similarly, should the random number be less than the content failure probability, a randomly generated response would be replied. Such failures were injected in the controlled environment of the services execution.

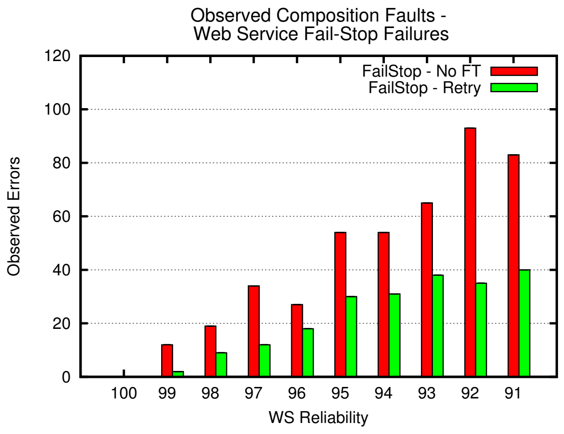
Question 2.1 : What is the impact of the services' reliability on the overall system reliability, regarding fail-stop failures? In this scenario, a set of 3 equivalent web services are available. Each web service is warranted to return only correct answers but will suddenly halt in some of the requests. In this experiment the probability of a fail-stop failure being injected ranged from 0% to 9% with steps of 1%. We performed a total of 1000 requests to each service and compared the observed output of: (1) a client choosing a fixed web service from the pool, *i.e.*, without using ChorTolerance, (2) ChorTolerance's *Retry* FTT, (3) *RB* and (4) *Active* FTTs. The amount of requests that were not properly replied (*e.g.* timed out, thrown an exception) for each WS reliability and FTT are shown in Figures 5.5a, 5.5c and 5.5e. As expected, usage of an appropriate FTT for fail-stop failures steeply lowered the amount of observed failed requests.

Question 2.2 : What is the impact of the services' reliability on the overall system performance, regarding fail-stop failures? For this evaluation the same setup as the one used for Question 2.1 was used. However, instead of measuring the amount of observed failed requests, we measured the average response time, for the correctly replied requests. The observed results are shown in Figures 5.5b, 5.5d and 5.5f. The results show that the overhead for using the *Active* FTT, regarding time is very low. *Retry* has a slightly higher overhead and *RB* has the most significant overhead.

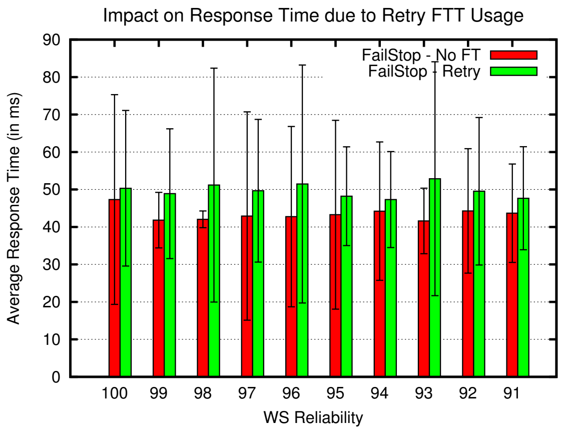
Question 2.3 : What is the impact of the services' reliability on the overall system reliability, regarding incorrect responses? In this scenario, a set of 3 equivalent web services are available, all of which have the same probability of returning an erroneous value. In this experiment the probability of a content failure be injected ranged from 0% to 9% with steps of 1%.

We compared the observed reliability of (1) a client randomly choosing a web service from the pool, *i.e.*, without using ChorTolerance and (2) using ChorTolerance's NVP FTT with three resources.The amount of observed failed requests are shown in Figure 5.6a.

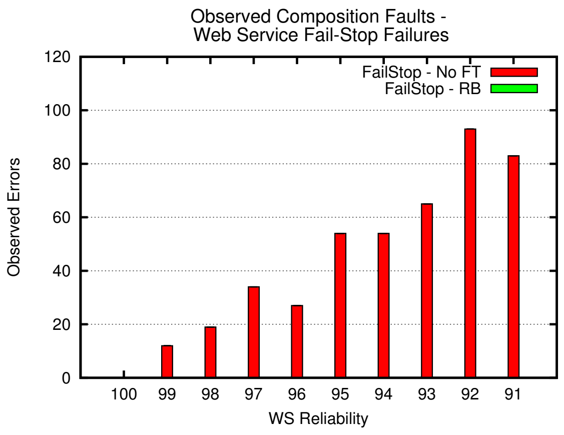
Question 2.4 : What is the impact of the services' reliability on the overall system performance, regarding incorrect responses? Once again, the same set-up as the previous experiment was used, but measuring the average response time for successful requests with and without the *NVP* FTT. The observed results are presented in Figure 5.6b. This FTT had the greatest overhead of all. This was expected since it awaited for all services to reply before deciding on the final value to be returned.



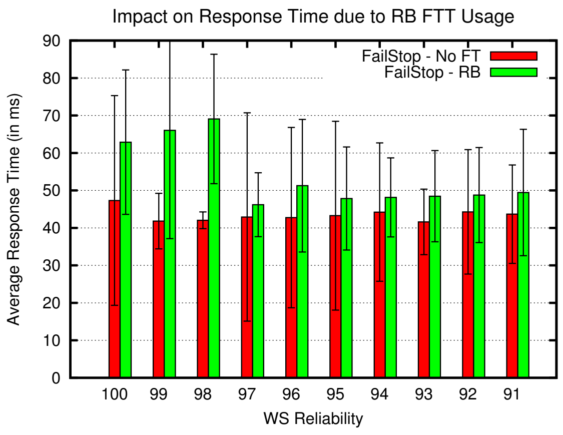
(a) ***Retry***: *Observed faults in 1000 requests*



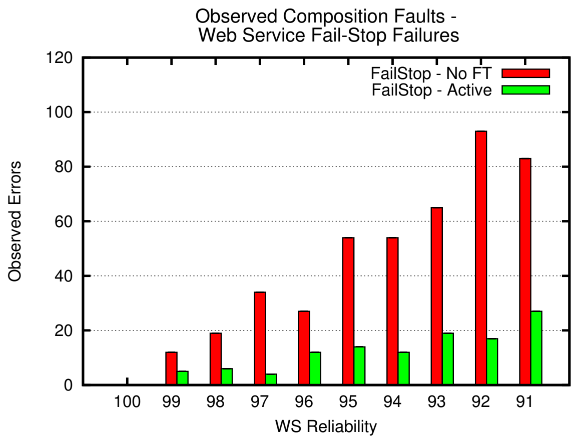
(b) ***Retry***: *Average response time for successful requests*



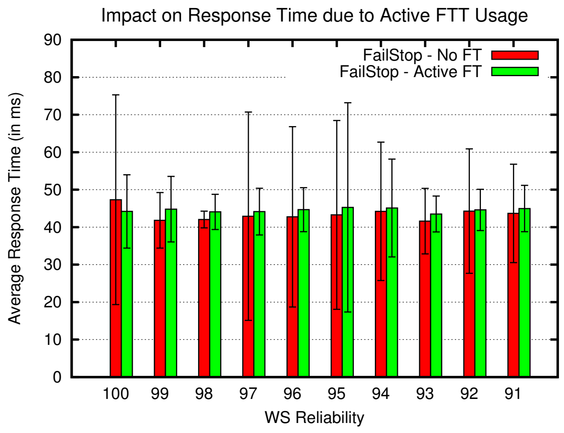
(c) ***RB***: *Observed faults in 1000 requests*



(d) ***RB***: *Average response time for successful requests*

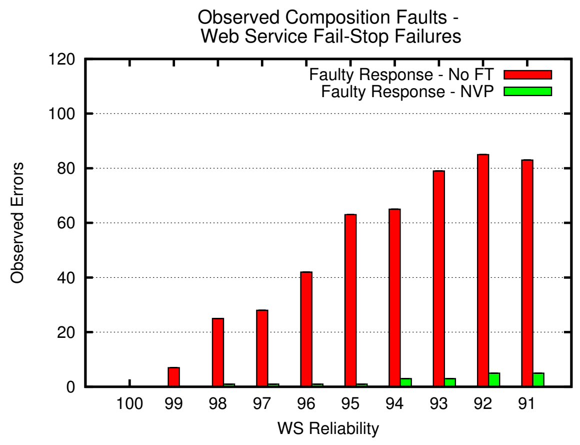


(e) ***Active:*** *Observed faults in 1000 requests*

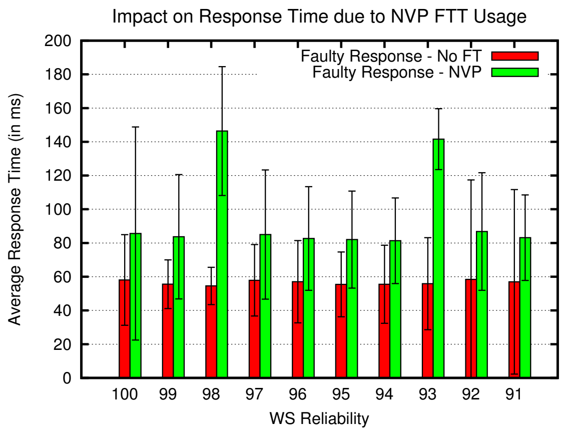


(f) ***Active***: *Average response time for successful requests*

Figure 5.5: *Comparison between observed faults and average response time for fail-stop scenarios with and without each of the built-in FTTs*



(a) ***NVP****: Observed faults in 1000 requests*



(b) ***NVP****: Average response time for successful requests*

Figure 5.6: *Comparison between observed faults and average response time for faulty response scenarios with and without NVP FTT*

Question 2.5 : What is the impact of a fault on a choreography agent itself on the overall system *performance* For this evaluation, we have performed a choreography consisting of two roles, and thus two agents. Firstly, we have performed the choreography in an error-free environment where no recovery was needed to establish the basic execution time. Then we performed the same choreography, but with the second agent missing thus forcing the first agent to identify and correct this situation by performing the second choreography agent's recovery.

The observed results indicated that the average execution time for the error-free choreography was of **(77±13) milliseconds** and for the recovered proxy scenario was of **(122±21) milliseconds**. These results allow us to deduce that the recovery itself takes a toll on the execution time of approximately 45 milliseconds in response to the occurrence of an error on an agent which was considered a good result.

5.2.3 Adaptation to different contexts

In adition, we also executed experiments to evaluate the capability of the proposed mechanism of adapting itself to different contexts, responding to changes observed in the service pool. Much like the other experiments, for this one we focused our attention on the reliability enhancement due to the usage of a single proxy.

We evaluated the capability and the observed behavior of the system under three context variations: Equivalent service pool increase, decrease and single service available. For each context, 1000 requests were issued and their response time measured. The results are depicted in Fig. 5.7. The vertical lines represent a timeout, or a failed request. Context changes over time are identified by the thick, dashed vertical lines.

Three services were used for this experiment: WS_90, WS_95 and WS_98 each one with 90%, 95% and 98% of reliability respectively.

Initially, only services WS_95 and WS_98 were available. Then, after request 1000, WS_90 was added to the pool. After request 2000, WS_98 is removed from the pool thus decreasing the maximum possible overall and individual reliability. Finally, after request 3000 WS_95 is also removed leaving only WS_90 in the ESP. This forces the proxy to dynamically switch to the retry technique.

The observed results are presented in Figure 5.6a (response time) and Table 5.3. They are interesting since in the first context (WS_95 and WS_98) the observed reliability was of 97.7. Once the third WS enters the pool it rises to 99% reliability under the second context and, after the pool falls back to two services it slightly decays back into 98.6% reliability. This was very in line with what would be expected from the mathematical model. The fourth context, with a single

Table 5.3: *Observed errors and reliability under each of the considered contexts*

Context	Observed Errors	Observed Reliability
1	23	97.7
2	10	99.0
3	14	98.6
4	0	100.0

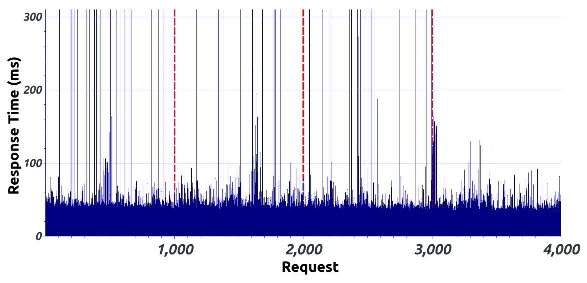


Figure 5.7: *Response time in a scenario with changes observed in the service pool*

service in the pool, actually masked all errors introduced by the web services. We advocate that this behavior was due to the fact that the error distribution was linear and the service provided was a simple mathematical task with very little computational effort and time consumption. However, it can be perceived that the average response time for the last scenario has increased in comparison to the non-faulty requests in the previous scenarios.

Chapter 6

Conclusions and Future Work

In this paper we have proposed the utilization of a Pragmatic CGM in which the goals' context-dependent interpretation is an integral part of the model. We have also shown why hard goals and softgoals are not enough to grasp some of the real-world peculiarities and context-dependent goal interpretations. We have also extended the model with Dalpiaz's goal annotations in order to properly specify the allowable system's behaviours at runtime.

We defined the pragmatic goals' achievability property: whether there is any allowable execution plan that fulfils the goal's interpretation under a given context. We also proposed, and implemented the PPA algorithm. This algorithm is able to decide on the achievability of a goal and, if so, engineer an execution plan in compliance with all the goal annotations to do so.

Finally, we thoroughly evaluated the model and the algorithm. The evaluation was performed in three different fronts, all of which rendered very appealing results. On the first front, we evaluated, for the MPERS case study, the algorithm's performance (execution time of less than 4ms in average) and reliability (no errors detected over more than 4000 executions). On the second front we have performed a scalability analysis on the algorithm running with every possible annotation and shown that in all the situations the PPA algorithm scales linearly over the amount of goals and context amount. For the third front, we have evaluated the capability of using the algorithm to pinpoint at design-time scenarios in which the model's root goal is unachievable by design in order to alert the GM designer of this potential flaw. The results for the third evaluation front showed that for models with 6000 goals and 10 contexts, the full scope of context sets could be swept in about 2'30" (two and a half minutes).

For future work, we plan to:

(1) compare the performance offered by the QoS-constrained algorithm to that achieved by SAT-solvers; (2) study the impact of considering more than one composite QoS constraint and; (3) integrate IMPROV's PRCGM model into a more user friendly tool to allow for easier modelling of the actors' behaviour.

Bibliography

[ADG10] Raian Ali, Fabiano Dalpiaz e Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requirements Engineering*, 15(4):439–458, 2010. 10, 11, 13, 16, 45

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell e Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, Janeiro 2004. 6, 42

[BCR94] Victor R. Basili, Gianluigi Caldiera e H. Dieter Rombach. The goal question metric approach. Em *Encyclopedia of Software Engineering*. Wiley, 1994. 49

[BDB05] A.P. Barros, M. Dumas e P.D. Bruza. The move to web service ecosystems. *BPTrends*, 3(3), 2005. 5

[BDH$^+$12] J. Behl, T. Distler, F. Heisig, R. Kapitza e M. Schunter. Providing fault-tolerant execution of web-service-based workflows within clouds. Em *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, página 7. ACM, 2012. 6, 7, 8, 9

[BPG$^+$04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia e John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004. 10

[BWR09] A. Barker, C.D. Walton e D. Robertson. Choreographing web services. *Services Computing, IEEE Transactions on*, 2(2):152–166, 2009. 6

[CKM02] Jaelson Castro, Manuel Kolp e John Mylopoulos. Towards requirements-driven information systems engineering: the *Tropos* project. *Information systems*, 27(6):365–389, 2002. 10

[DBHM13] Fabiano Dalpiaz, Alexander Borgida, Jennifer Horkoff e John Mylopoulos. Runtime goal models: Keynote. Em *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, páginas 1–11. IEEE, 2013. xv, 10, 11, 13, 17

[Dob06] G. Dobson. Using ws-bpel to implement software fault tolerance for web services. Em *Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on*, páginas 126–133. IEEE, 2006. 6, 7, 9

[FD02] A. Fedoruk e R. Deters. Improving fault-tolerance by replicating agents. Em *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, páginas 737–744. ACM, 2002. 6, 7, 9

[FS01] Anthony Finkelstein e Andrea Savigni. A framework for requirements engineering for context-aware services. *STRAW 01*, 2001. 11

[GCRB13] Felipe Guimaraes, Pedro Celéstin, Genaina Rodrigues e Daniel Batista. A framework for adaptive fault-tolerant execution of workflows in the grid: Empirical and theoretical analysis. *Journal of Grid Computing*, 2013. Em submissão. A terceira revisão foi submetida dia 24 de julho. PDF disponível em http://goo.gl/aIEeWP. 8

[Gen97] Michael R Genesereth. An agent-based framework for interoperability. *Software agents*, páginas 317–345, 1997. xiii, 5, 6

[GKB$^+$12] F. Guimaraes, E. Kuroda, D. Batista et al. Performance evaluation of choreographies and orchestrations with a new simulator for service compositions. Em *CAMAD 2012-International Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks*, 2012. 6, 33, 34

[GRAB17] FP Guimaraes, GN Rodrigues, Raian Ali e DM Batista. Planning runtime adaptation through pragmatic goal model. *Data and Knowledge Engineering*, 2017. 13, 14

[LFCL03] D. Liang, C.L. Fang, C. Chen e F. Lin. Fault tolerant web service. Em *Software Engineering Conference, 2003. Tenth Asia-Pacific*, páginas 310–319. IEEE, 2003. 6, 7, 9

[LR12] H. Luthria e F.A. Rabhi. Service-oriented architectures: Myth or reality? *Software, IEEE*, 29(4):46–52, 2012. 5

[LV07] N. Laranjeiro e M. Vieira. Towards fault tolerance in web services compositions. Em *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, página 2. ACM, 2007. 7, 9

[MAR14] Danilo F. Mendonça, Raian Ali e Genaína N. Rodrigues. Modelling and analysing contextual failures for dependability requirements. Em *Proceedings of the 9th SEAMS*, páginas 55–64, New York, NY, USA, 2014. ACM. xiii, 12

[MBHJ$^+$11] A. Mdhaffar, R. Ben Halima, E. Juhnke, M. Jmaiel e B. Freisleben. Aop4csm: An aspect-oriented programming approach for cloud service monitoring. Em *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, páginas 363–370. IEEE, 2011. 6, 7, 8, 9

[MGMS11] D. Menasce, H. Gomaa, S. Malek e J. Sousa. Sassy: A framework for self-architecting service-oriented systems. *Software, IEEE*, 28(6):78 –85, nov.-dec. 2011. 7, 8, 9

[MRD08] O. Moser, F. Rosenberg e S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. Em *Proceedings of the 17th international conference on World Wide Web*, páginas 815–824. ACM, 2008. 7, 8, 9

[Pap03] M.P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. Em *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, páginas 3–12. IEEE, 2003. 5, 27

[Pel03] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003. 6

[PGNRMBA15] Felipe Pontes Guimaraes, Genaina Nunes Rodrigues, Daniel Macedo Batista e Raian Ali. *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, chapter Pragmatic Requirements for Adaptive Systems: A Goal-Driven Modeling and Analysis Approach, páginas 50–64. Springer International Publishing, Cham, 2015. 13, 14, 19

[PTDL07] M.P. Papazoglou, P. Traverso, S. Dustdar e F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007. 5

[SLRM11] Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson e John Mylopoulos. Awareness requirements for adaptive systems. Em *Proceeding of the 6th SEAMS*, página 60, New York, New York, USA, Maio 2011. ACM Press. 10, 14

[WB10] Y. Wei e M.B. Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *Internet Computing, IEEE*, 14(6):72–75, 2010. 5, 26

[YM98] Eric Yu e John Mylopoulos. Why goal-oriented requirements engineering. Em *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, páginas 15–22, 1998. 10

[Yu11] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011. 10

[ZL10] Zibin Zheng e Michael R Lyu. An adaptive qos-aware fault tolerance strategy for web services. *Empirical Software Engineering*, 15(4):323–345, 2010. 7, 8

Index

DFT, *see* transformada discreta de Fourier
DSP, *see* processamento digital de sinais

Fourier
    transformada, *see* transformada de Fourier

STFT, *see* transformada de Fourier de tempo reduzido

TBP, *see* periodicidade região codificante