Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

An Architecture to Support Control Theoretical-based Verification of Goal-Oriented Adaptation Engines

Ricardo Diniz Caldas

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Profa. Dra. Genaína Nunes Rodrigues

Brasília
2019

Ficha Catalográfica de Teses e Dissertações

Está página existe apenas para indicar onde a ficha catalográfica gerada para dissertações de mestrado e teses de doutorado defendidas na UnB. A Biblioteca Central é responsável pela ficha, mais informações nos sítios:

http://www.bce.unb.br
http://www.bce.unb.br/elaboracao-de-fichas-catalograficas-de-teses-e-dissertacoes

Esta página não deve ser inclusa na versão final do texto.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

An Architecture to Support Control Theoretical-based Verification of Goal-Oriented Adaptation Engines

Ricardo Diniz Caldas

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Profa. Dra. Genaína Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Vander Alves
Universidade de Brasília

Profa. Dra. Cecilia Mary Fischer Rubira
Universidade Estadual de Campinas

Profa. Dra. Genaína Nunes Rodrigues
Coordenador do Programa de Pós-graduação em Informática

Brasília, 23 de Outubro de 2019

Agradecimentos

Em especial, agradeço aos meus pais, Lea e Ricardo, e meus irmãos, Gabriela e Leonardo, com os quais aprendi honestidade, humildade e resiliência, que foram fundamentais na concepção deste trabalho. Agradeço também à orientadora e professora Genaína Rodrigues por proporcionar uma experiência acadêmica única e de excelência, que me levou ao encanto pela pesquisa. Por fim, agradeço aos colegas do laboratório (LES) e do departamento (CIC) pelas discussões e colaborações que encontram-se diluídas nas reflexões do presente trabalho, Arthur Rodrigues, Gabriela Solano, Gabriel Rodrigues, Léo Moraes, Gabriel Levi, Eric Gil, Samuel Couto, Jorge Mendes e tantos outros.

Resumo

Sistemas de software de longa vida devem evoluir e ser mantidos para lidar com as necessidades flexíveis das partes interessadas, mudanças no ambiente e o comportamento incerto dos componentes internos. Diversas abordagens na Engenharia de Software propõem aplicações de uso intensivo de software com recursos de autogerenciamento para superar as barreiras ao sucesso de sistemas intrinsecamente dinâmicos e complexos, com nenhuma ou pequena intervenção humana. No entanto, a natureza da adaptação autonômica não é trivial, pois a combinação de todas as condições operacionais possíveis levaria a incalculáveis soluções baseadas em pesquisa para atingir o objetivo do sistema. O processo de projeto de software orientado a objetivos defende que colocar os objetivos do sistema como prioridade restringe as possibilidades de adaptação e fornece uma estrutura direta que garante o comportamento confiável do sistema, orientando atividades de desenvolvimento, manutenção e evolução propensas a erros. O presente trabalho propõe uma contribuição para o processo de projeto orientado a objetivos de para sistemas auto-adaptativos, por meio do fornecimento de uma arquitetura para verificação de sistemas auto-adaptativos, que mapeia modelos de objetivos para o código executável do *Robot Operating System* (ROS) executável sob a influência das incertezas. A etapa de verificação é baseada na coleta de dados em tempo de execução e na análise de séries temporais, seguindo métricas da Teoria de Controle. Assim, os engenheiros de sistemas de software auto-adaptativos podem contar com evidências quantitativas para avaliar os mecanismos de adaptação com garantias de confiabilidade. A abordagem foi avaliada pela aplicação do processo de verificação em um mecanismo de adaptação orientado a objetivos, que adapta o comportamento de um sistema médico para melhorar a confiabilidade do sistema. Como resultado, a verificação forneceu informações sobre como melhorar o mecanismo em relação às suas configurações para combater o ruído sensores, levando a uma solução mais robusta.

Palavras-chave: Sistemas auto-adaptativos, Projeto de software orientado a objetivo, Teoria de Controle

Abstract

Long-lived software systems should evolve and be maintained to cope with flexible stakeholders' necessities, changing environments and internal component's uncertain behavior. A large body-of-knowledge has been proposed for software-intensive applications with self-managing capabilities to overcome the barriers to the success of inherently dynamic and complex systems with none or tiny human intervention. Nonetheless, autonomic adaptation nature is not trivial since the combination of all possible operational conditions would hinder infinite search-based solutions towards reaching the system's goal. The goal-oriented software design process advocates that embracing the system's goals as first-class citizens constrains the adaptation possibilities and provides a straightforward framework that guarantees the system's trustworthy behavior by guiding error-prone development, maintenance and evolution activities. The present work proposes a contribution to goal-oriented design process of self-adaptive systems approaches by means of providing an architecture for verification of self-adaptive systems, which maps contextual goal-models to executable Robot Operating System (ROS) code that runs upon the influence of uncertainties. The verification step is based on runtime data collection and time-series analysis w.r.t control theoretical based properties. Thus, engineers of self-adaptive software systems can rely on quantitative evidences to evaluate adaptation engines with guarantees of trustworthiness. The approach was evaluated by the use of the verification process upon a goal-oriented adaptation engine, which adapts the behavior of a medical system in order to improve the system reliability. As a result, our solution provided insights on how to improve the engine configurations for tackling the noise in sensing source of uncertainty, leading into a more robust engine.

Keywords: Self-adaptive software, Goal-Oriented Design Process, Control Theory

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Long-lived software systems should evolve and be maintained to cope with flexible stakeholders' necessities, changing environments and internal components uncertain behavior [1]. However, complex applications, inaccessible or dangerous environments and high cost of specialized human work places barriers on human-driven software adaptation. Therefore, Kephart J. et al. discussed the need to systematically address software systems capable of autonomously adapting to runtime disturbances with none or tiny human intervention, the self-adaptive systems (SAS) [2]. Since then, a widely spread effort on producing approaches to enable modeling, development and maintenance of SAS has been placed [3, 4].

Nevertheless, tackling such dynamic needs is not trivial mostly due to uncertainties arising from multiple sources. First and foremost, uncertainty is an intrinsic property of unexplored scenarios in the software design phase, given those unpredictable events that arise at runtime, as well as error-prone requirements elicitation techniques, and unexpected situations that demand creative solutions [5]. Thus, reasoning on the impact of uncertainty on the system's runtime behavior is fundamental for building trustworthy adaptation mechanisms in order to provide continuous goals achievement [6]. As a result, the reasoning process needs strict guidelines to assure that the end behavior follows high level rules that prevents the system from reaching any state that violates non-negotiable requirements (e.g. safety-critical applications) [7].

In order to help humans build trust in SAS, it is paramount to have goal-definition and visualization paradigms so that the specified goals do represent what is really desired [2]. For this reason, goals have become a first-class entity in SAS [8]. To recognize and manage uncertainties in the assurance process from early on, GORE (Goal-Oriented Requirements Engineering) offers proved means to decompose technical and non-technical requirements

into well-defined entities (goals) and reason about the alternatives to meet them. Hence, it has been used as a means to model and reason about the systems' ability to adapt to changes in dynamic environments [7, 9, 10, 11, 12].

Thereby, it is noteworthy that the system goals continuous satisfaction is challenging, specially when the system operates under the presence of uncertainty [13]. This has been subject of research on the self-adaptive software engineering community in the last few decades [4, 14, 15, 16]. However, many of the proposed solutions lack on a strong theoretical background for assuring the system dependability, what culminated on a research front that employs Control Theory for guiding the adaptation mechanisms design [8, 13]. After all, Control Theory provides a set of strong mathematically-based techniques which could guide the adaptation mechanisms design. In that sense control theoretical methods have been proposed [17, 18] to pave the way for performing trustworthy adaptation where the system goals are placed as first-class citizens.

1.2 Research Challenges

The assessment of uncertainty during the software design phase is vital for long-lived software systems. With that in mind, approaches rely on goal model verification [9, 10] to shorten the design-time and runtime gap by generating runtime models augmented with uncertainty from previously verified models in the light of model-checking. However, it is not sufficient to verify whether the system can reach the desired goal in the control-based mechanisms design process for SAS. With this in mind, we raise the first research question that we aim to address in this work.

How to ensure that the goal-oriented adaptation engine guarantees hold even when the system operates in the presence of uncertainty?

Goals have been used as a means to model and reason about the systems' ability to adapt to changes in dynamic environments. However difficulties w.r.t eligible software system models, methodologies as well as architectures that pursue controllability as a first-class concern still remain. It is noteworthy that the contributions so far step forward into supporting the development of self-adaptive software, while they shorten the distance between design-time and runtime [6]. However they still lack on concrete architectures for evaluating whether the employed adaptation policies are in compliance with the control-theoretical requirements regarding the system's reaction in face of unforeseen changes, i.e. uncertainty, during runtime. Then, we raise the second research question.

Can we conceive a concrete architecture for SAS that seamlessly integrates goal-oriented adaptation process and control theoretical verification while accounting for uncertainty?

To the best of our knowledge mRubis [19] is the only exemplar in SAS literature that supports the development, runtime evaluation and comparison between model-based adaptation techniques. Even though the approach may hinder scenarios execution for empirical validation of SAS, it would demand extra effort for integrating the proposed architecture to goal model specification language. Also, the metrics provided by the architecture are not compliant with control-theoretical guarantees. In addition, means are necessary to verify the system behavior following implemented and integrated adaptation engines [17]. Finally, we raise the third research question.

How to analyze the guarantees provided by goal-oriented adaptation engines from a control theoretical perspective?

1.3 Research Contributions

In a nutshell, our contributions pervades the research questions in three complementary directions that summed up contribute to an end-to-end process from goal modeling to runtime verification. Thus, first we present a method that contributes with the assurance that the goal-oriented adaptation engine properties hold in the presence of uncertainty by providing means to exercise the system in a runtime environment with monitorable disturbance injection into uncertainty sources. Second, we conceptually propose a layered architecture that support the integration between goal-oriented adaptation process and control theoretical verification. And finally, we present a process for analyzing the guarantees provided by adaptation engines from control theoretical perspective, that stands upon timeseries analysis. As a minor contribution, we provide an exemplar of SAS for empirical validation.

1.4 Evaluation

We evaluate our work by collecting evidence to our claim that the proposed architecture supports the verification of guarantees provided by goal-oriented adaptation engines from a control theoretical perspective. In two sequential experimental sets in which we evaluate whether the goal-oriented model and the analysis algorithm provide enough information for a sound control theoretical analysis of the system behavior and whether an assurance process based on the model may lead the system into ensuring the desired prop-

erties even when operating under uncertainty. All upon an architecture for self-adaptive systems following which a Body Sensor Network prototype is implemented, where we monitor and adapt individual components parameters for ensuring a reliable system. For the first, the results show that the relative error between the goal-oriented model and the monitored behavior w.r.t control theoretical metrics are sufficiently acceptable, but more investigation could be placed on the model's sensitivity in respect to the terms contribution to its global state. In addition, results derived from the second experimental set assert that the approach enables a sound verification of the system behavior under influence of the goal-oriented adaptation engine even in presence of noise in sensing, source of uncertainty, towards guaranteeing continuous fulfillment of the desired system goals. Summed up, our evaluation efforts build evidence that our runtime verification architecture seamlessly contributes to the goal-oriented SAS design process.

1.5 Document Roadmap

The rest of the document is organized as follows. Chapter 2 provides further detail on the theoretical foundations necessary for the proposal explanation. Chapter 3 details the approach that tackles the claimed research questions. Chapter 4 discusses the prototype implemented and the experimental scenario employed in the work along with the results. Conclusions about the work itself and future work are presented in Chapter 6.

Chapter 2

Theoretical Background

2.1 Self-Adaptive Software Systems

Continuous behavior change is a basic necessity for long-lived software systems, since stakeholders' needs can be volatile, operating environments be dynamic and the system's internal structure, hardware and/or software, may degrade in time. However, change might lead to unwanted behavior if not performed with caution, to address that, many software engineering techniques (e.g. BDD, TDD, refactoring, etc) have been proposed for human-driven adaptation. Though, strict needs on the behavioral change may unable human-driven change such as budget availability, unreachable environments, time constraints for performing software adaptations. Self-adaptive software engineering have been studied to address this issue by enabling systems to adapt themselves with tiny or none human intervention. When it comes to the self-adaptive system high-level architecture, it's well-established that the participating entities are organized into managing system, managed system and environment, see Figure 2.1.

A long list of studies [3][4] on how to develop autonomous software systems from a wide variety of perspectives have been published in the last twenty years right after the seminal work of Kephart and Chess [2] which proposed a first version of the MAPE-K adaptation loop architecture. Among all, we have decided to deepen the study on the MORPH reference architecture [20] since a hierarchical multi-layer which fits well with goal-oriented approaches is proposed, which is the core of this work.

2.1.1 MORPH reference architecture

The MORPH reference architecture for self-adaptive systems, proposed by Braberman V. et al., provides a four-layered architecture in which adaptation reasoning is divided in two perspectives, reconfiguration and behavioral. The dynamic reconfiguration stands

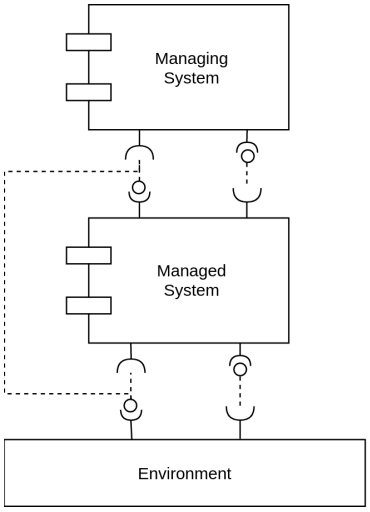


Figure 2.1: Self-adaptive systems generic component architecture

for runtime changes of component structure and operational parameters in order to guarantee non-functional requirements such as the ones comprising dependability [21] (e.g. reliability, availability, security). On the other hand, behavioral update guides the system behavior to ensure that high-level goals are satisfied and is taken as more of an orchestration of the system components. A representation of the proposed architecture is depicted in Figure 2.2.

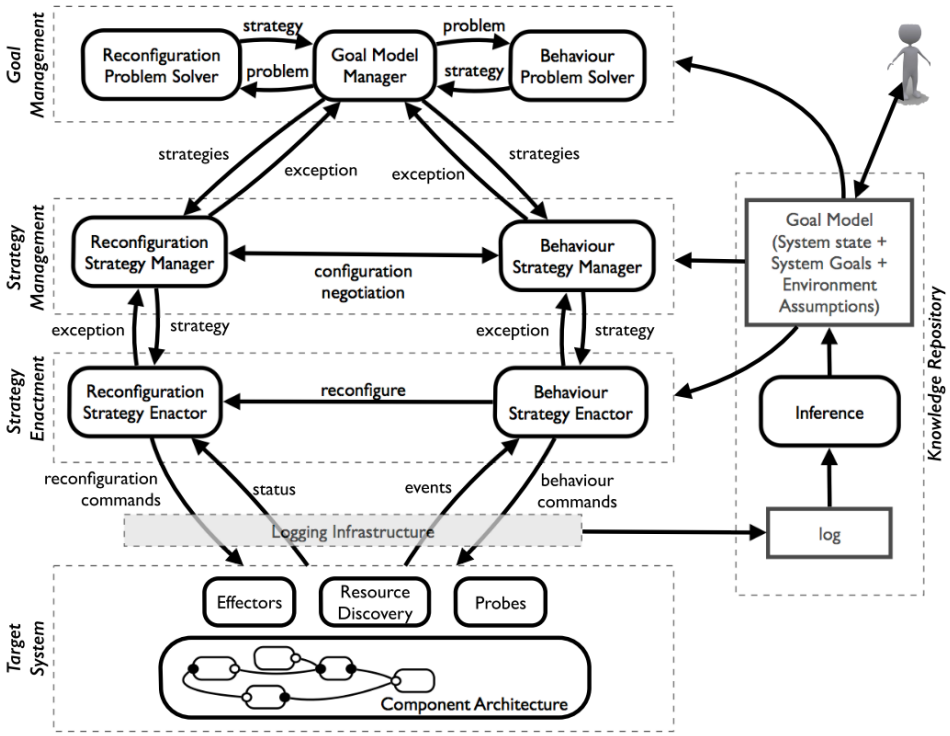


Figure 2.2: The MORPH reference architecture [20]

The *Goal Management* layer encompasses the highest level management reasoning processes of the architecture. In which, a Goal Model Manager triggers reconfiguration or behavior problem solvers for determining solutions in respect to strategies that satisfy the target system goals. This layer interacts with a *Knowledge Repository* from which the goal models, for example, are extracted, as well as with the *Strategy Management* layer by, within a feedback loop fashion, sending new strategies resolutions and receiving exceptions that are seen as constraints towards finding acceptable strategies.

Then, the *Strategy Management* layer is responsible for triggering adaptations to changes that can sufficiently be addressed by pre-processed strategies, the ones synthesized by the *Goal Management* layer. The main concept of the layer is to permit quick adaptation in face of not attending runtime strategies. It receives adaptation requests from the *Strategy Enactor* layer and replies it with a deployable strategy. However, when no strategy suffices the constraints imposed by the lower layer, it must throw an exception to the *Goal Management* layer. There also is internal information exchange in between the behavior and reconfiguration managers to ensure consistency in the strategy selection.

Further into the architecture, the *Strategy Enactor* layer involves the target software system constant monitoring and triggering either behavioral or reconfiguration operations thus guiding the system towards the strategy's goals. It directly interacts with the system's sensors and actuators. In case of unsolvable adaptations regarding the system's state, exceptions are thrown to the upper layer.

A logging mechanism is recommended in between the *Strategy Enactor* layer and the target system, it can collect information regarding the system states during the execution and the adaptations triggered. This information can be further treated and interpreted to refine the models within the *Knowledge Repository.* At last, the *Target System* layer, that contains the components that execute the system to satisfy the stakeholders' needs. It is necessary that the target system provides mechanisms for state monitoring and either reconfiguration and/or behavioral actuation.

2.2 Control Theoretical Analysis

Control systems are subject to inputs not known at design-time, based on that, they are designed to respond within acceptable boundaries to dynamic environmental interactions. The analysis on whether the designed system is operating accordingly demands metrics that can be used for comparison in respect to the performance of various control systems [22]. Design-time analysis methods mainly consist on stressing a model of the system with known input signals and initial conditions variations, and collecting its response in terms of the properties to be evaluated, then the metrics that correspond to the

response behavior are calculated and can be used for further comparison. This section details the analysis of typical second order responses and presents the metrics employed throughout the work.

When mechanical systems are affected by an input, the amount of energy contained varies and it responds by sparing the received energy towards the equilibrium. The energy dispersion can be realized by movement, emission of sounds or release of heat, for example. Then, suppose a piano key that, when pressed, unleashes a hammer that hits a stretched string, that produces sound to disperse the kinetic energy transferred to it. So, once the piano key is pressed and released the piano responds with a sound with an intensity proportional to the force applied by the pianist to the key. Now, suppose that the pianist desires that the sound intensity be around $64W/m^2$, characterizing the *setpoint*. He or she, then, presses the intended piano key with a correspondent force producing the curve depicted in Figure 2.3.

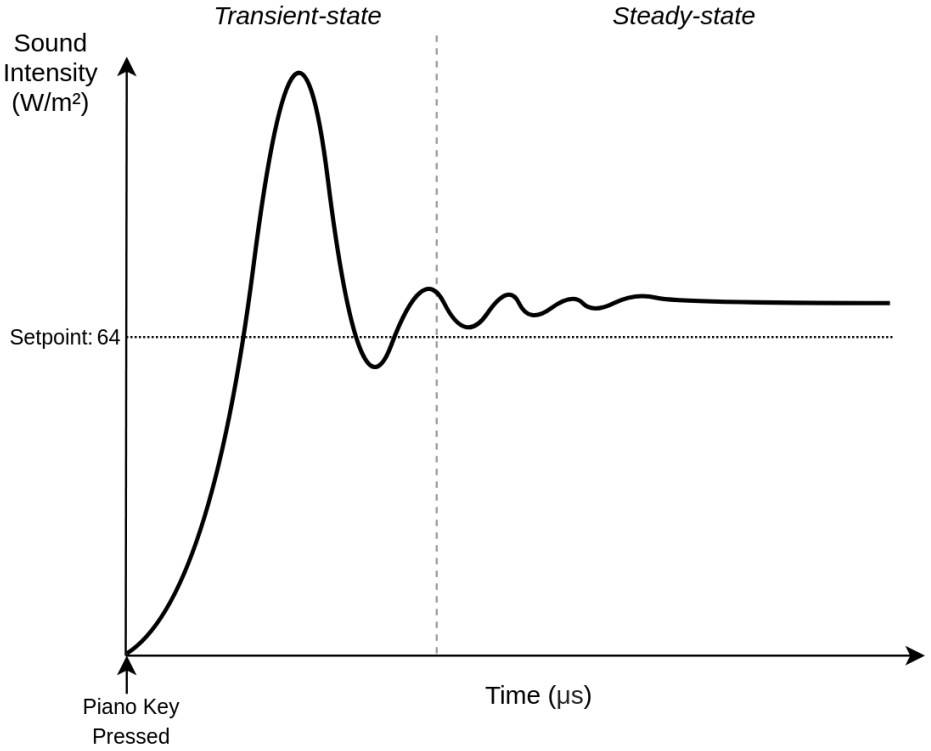


Figure 2.3: Piano's sound intensity response to unitary key press

The transient portion of the curve is defined by the moments before the sound intensity stabilizes nearby the setpoint value, according to the commonly used stability criterion, the system response stabilizes when it converges to a certain value [22]. When the curve is within the stability region, it is said to have achieved the steady-state. The Control Theory community, specifically for transient and steady-state analysis [22][18]

defined metrics to classify and compare responses to inputs be them known (forecasts) or unknown (disturbances). In this work, they are presented in Table 2.1.

ID		Metric	Description
G1	**M1**	Stability	"Ability of the system to achieve its goals"
	M2	Settling time	"The time required to reach the setpoint boundaries after a goal change"
	M3	Overshoot	"Spikes in the system output for different adaptation options"
	M4	Steady-state error	"Oscillations in the response time of the software for different adaptation options"
G2	**M5**	Control effort	"The amount of resources consumed by the adaptation mechanism to achieve goals"
	M6	Robustness	"Deviations in the system output under disturbances"
	M7	Optimality	"The tasks completed and the resources used by the software for different adaptation options"

Table 2.1: Behavioral metrics adapted from [18]

The metrics M1 - M4, grouped by G1, are related to transient and steady-state aspects of the runtime behavior of the system and the M5 - M7, grouped by G2, are related to the controller properties on whether it can optimally adapt to specific situations and is robust enough in face of uncertain scenarios. Where, G1 stands for 'Input Response' and G2 to 'Controller Property'. Formal means of evaluating the metrics in respect to the mathematical models that represent the controller, the system plant and the control loop topology applied to each case are placed. However, white-box approaches are not valid in the scope of the work since it is claimed that not known a priori control algorithms are evaluated through the analysis.

The input response analysis does not comprise the whole collected execution life-cycle, since more than one inputs can be taken during an analysis experimental set. Therefore it is defined that the response itself begins at the instant that the input is observed until a few moments after the system stabilization or another input is taken. See Figure 2.4. On the other hand, the controller property analysis takes into consideration both, the complete period of simulation execution. In contrast to the input response, the controller properties analysis is related to the amount of computational resources demanded, its capacity of dealing with uncertain scenarios and whether the adaptation decisions leads the system to an optimal behavior.

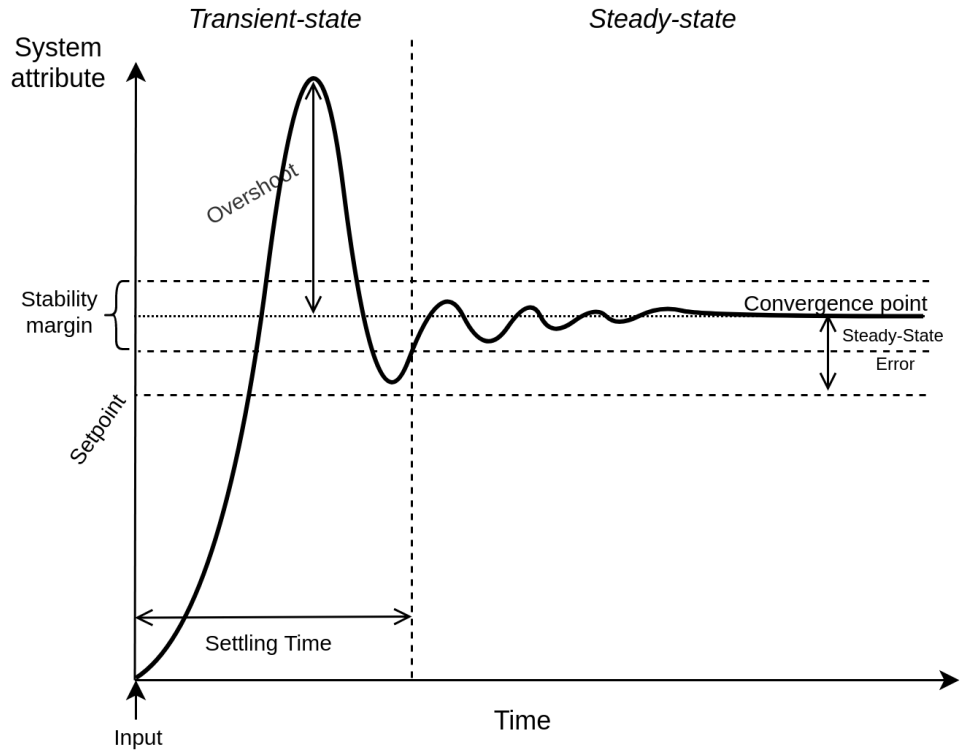


Figure 2.4: Input response with metrics

2.3 Goal-Oriented Software Design Process

The goal-oriented requirements engineering (GORE) has been an active field of study for the last two decades, in which goals are taken as first-class citizens [23]. Typically, GORE approaches advocates the use of goal models for elicitation, conceptualization and analysis of the system requirements, in which, interactions between the system's and environment's entities collaborates positively or negatively for the stakeholders' needs satisfaction. Despite that, approaches that not only employs goal models for creating and reasoning, but, for the entire life cycle (e.g. architecture, process design, coding, testing, adaptation, evolution) have been extensively proposed [24], with the advantage that the system operations continuously meet the goals it was designed or adapted for.

Using goal models in the entire life cycle of the application demands trustful transformations, mappings or integration from source, goal model, to the target, which corresponds to the artifact to be used in one or more phases of the software design or runtime. Horkoff J., et. al [24] performed a meta-study on the literature regarding transformations from/to goal models that pervades the entire life cycle of software systems. In which a classification of the transformations is placed. This work employs Horkoff's classification, see Table 2.2, in order to both delimit the scope of its contributions to the goal-oriented design process approaches and be able to further compare them to existent approaches in

Transformation	A process that takes one or more source models as input and produces one or more target models as output by following a set of transformation rules.
Mapping	A set of rules that describes how one or more constructs in the source modeling language can be connected to one or more constructs in the target modeling language.
Integration	The creation of a new modeling language which is made up of constructs and relations from the source and target modeling languages.
Exogenous Transformation	A transformation between models expressed in different languages.
Endogenous Transformation	A transformation between models expressed in the same language.
Vertical Transformation	A transformation where the source and target models reside at different abstraction levels.
Horizontal Transformation	A transformation where the source and target models reside at the same abstraction level.

Table 2.2: Transformation definitions [24]

the literature.

Also, we leverage the importance of having one goal model structured in several languages, since distinct reasoning processes with specific constraints might read or write in it. To maintain consistency, exogenous transformations are to be held whenever is necessary. A contextual goal model is used as the main model of this work, due all its advantages such as the proximity with the natural language and the liability for transformations. Then, a transformation from CGM to parametric formulae is advised, since at runtime, the goal tree does not scale for fast decision making. Also, a transformation from goal model to system architecture is proposed in this work. In the next topics, the languages and respective transformations used throughout the work are detailed and conceptual remarks are placed for the goal model to architecture transformation.

Furthermore, we follow the design-time process from Solano et al. [10] from contextual goal modeling to the parametric formula generate and contribute with an automatic runtime policy synthesis at runtime, see Figure 2.5. In this work, the contextual goal modeling extends GODA [9] with capabilities of taking uncertainties into account. In addition, the CGM to parametric symbolic formula process generates formal models, i.e. algebraic formulae, embedded with the representation of uncertainties for reliability and cost. The processes are further detailed in the next topics.

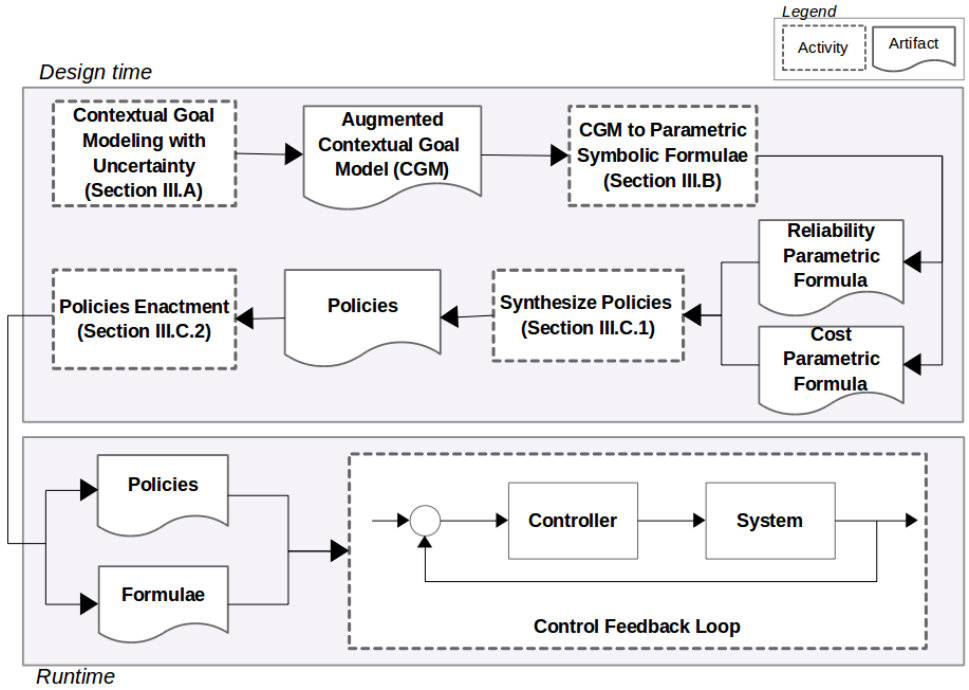


Figure 2.5: Goal-oriented design process for SAS

2.3.1 Contextual Goal Modeling

According to [11], a contextual goal model (CGM) is composed of: (i) actors such as humans or software that have goals and can decide autonomously on how to achieve these goals; (ii) goals as a useful abstraction to represent stakeholders' needs and expectations, offering an intuitive way to elicit and analyze requirements; (iii) tasks as atomic parts that are responsible for the operationalization of a system goal, that is, an operational means to satisfy stakeholders' needs; and (iv) contexts as partial states of the world that are relevant to a goal. A context is strongly related to goals since context changes may affect the goals of a stakeholder and the possible ways to satisfy the goals. Goals and tasks of a CGM can be refined through AND-decomposition or OR-decomposition, that is, a link that decomposes a goal/task into sub-goals/tasks, meaning that all or at least one, respectively, of the decomposed goals/tasks must be fulfilled/executed to satisfy its parent entity. The link between a goal and a task is called means-end, and indicates a means to fulfill a goal through the execution of a task.

In the example of Figure 2.6, the Body Sensor Network, actor, and its main objective is elicited as the root goal "G1: Emergency is detected". To decompose the goal tree until the tasks that operationalize the root goal, one should ask herself what it needs to be done to satisfy the root goal and how. Then, G1 is decomposed into a soft goal (how) and a hard goal (what), as for the soft goal, it is stated that G1 must be achieved with at least 90% reliability, and for the hard goal that the patient should be monitored, "G2:

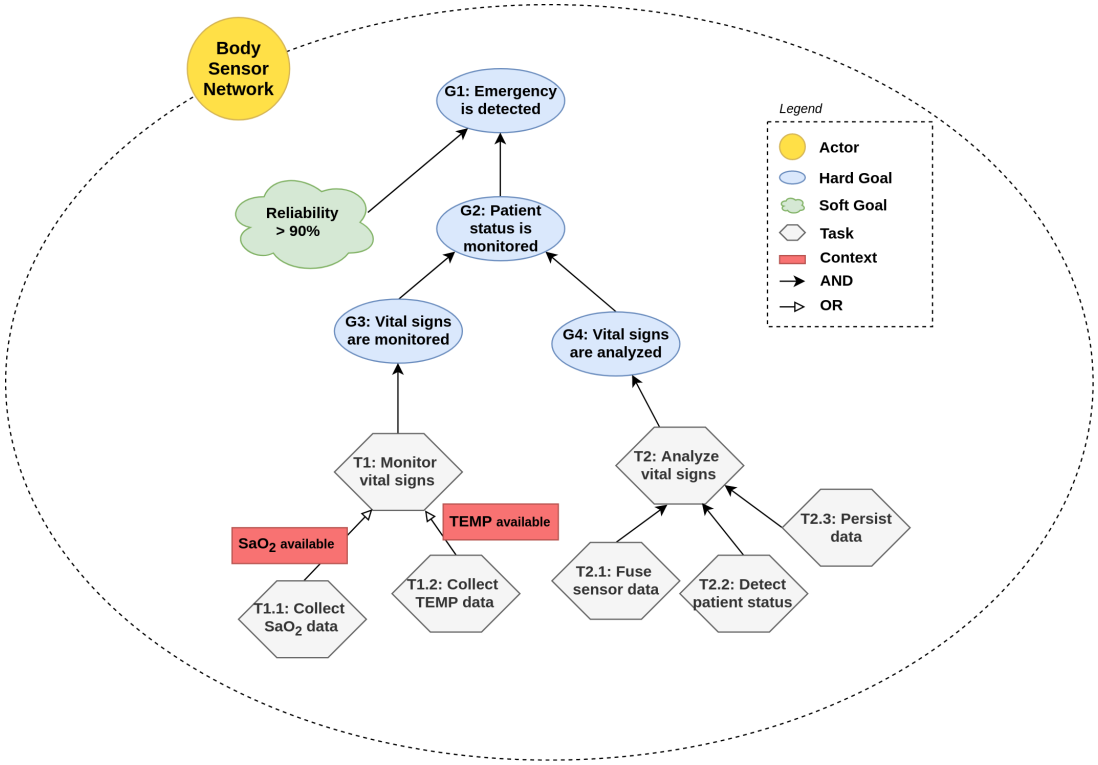


Figure 2.6: Contextual goal model example

Patient status is monitored". Next, G2 is decomposed into "G3: vital signs are monitored" and "G4: Vital signs are analyzed", through an AND decomposition since both must be realized. Both goals are decomposed into tasks, "T1: Monitor vital signs" and "T2: Analyze vital signs", respectively. Such tasks are decomposed, within the boundary of the Body Sensor Network actor, to finally reach executable tasks. T1 is decomposed into T1.1 and T1.2, supported by an OR decomposition, stating that to fulfill the vital signs monitoring, the system could collect SaO_2 data or temperature data. T2 is decomposed into "T2.1: Fuse sensor data", "T2.2: Detect patient status" and "T2.3: Persist data" within an AND decomposition. The operation of the Body Sensor Network is subject to two context conditions, both regarding the availability of the sensors needed to collect vital signs data.

2.3.2 CGM to Parametric Formulae Transformation

Parametric formulae are key enabler towards runtime analysis [9], since its computation is constant. The produced formulae composes each tasks' QoS attributes, regarding its operation, the contextual information associated and the means to satisfy the root goal. Therefore, the formulae can be used in the application behavior analysis since the parameters are constantly collected and updated during runtime.

The transformation technique is automatic stands for traversing the contextual goal tree and composing the nodes through arithmetic operations regarding the nodes and edges relations, in terms of decomposition, runtime behavior, weights, contexts, thus comprising an exogenous vertical transformation. In piStarGODA-MDP for example, the formulae is created over a process of building parametric MDPs, in respect to the execution behavior in Figure 2.7, in PRISM language for each leaf task node in the CGM and assembling them to represent the root goal fulfillment.

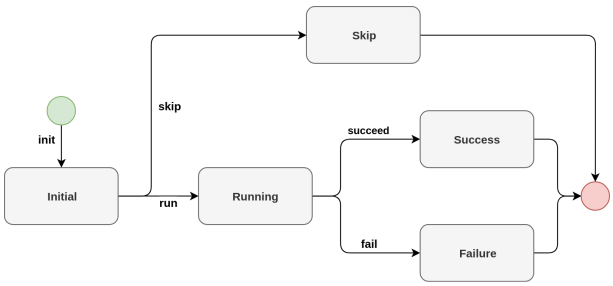


Figure 2.7: Software module execution behavior [9]

Furthermore, the tool relies on the root goal specification upon the probabilistic existence property, which evaluates the probability of the system eventually reaching a state that satisfies a goal of interest. Finally, it uses the parametric model checking PARAM [25] to generate the parametric formulae in a bottom-up fashion (i.e. from leaf-tasks to root goal), see Figure 2.8. Since G3 has no specific feature, its reliability will be the same as its subtree "T1: monitor vital signs". Subtrees $T_{1.1}$ and $T_{1.2}$ have both an AND-decomposition, thus the reliabilities of each subtrees are multiplied to obtain the reliabilities of each, $T_{1.1}$ and $T_{1.2}$. Finally, the leaf nodes have their reliability retrieved by PARAM, in which rT_i and fT_i represent the reliability and execution frequency of leaf node i. Similarly, cost formulae are generated by the algorithm.

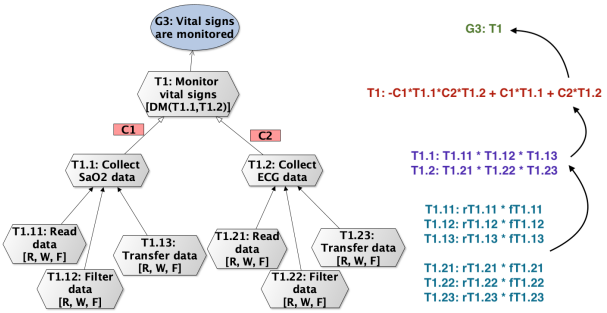


Figure 2.8: Bottom-up formulae generation process [10]

Chapter 3

Our Approach

3.1 Introduction

The present work provides an end-to-end conceptualization of an architecture for supporting adaptation engines verification and analysis. Moreover, it contributes seamlessly to the goal-oriented design process of trustworthy self-adaptive software systems. First, we introduce our view on the design process where we contribute to the goal-oriented SAS process from Solano et. al [10]. Then we present a vision of the concrete architecture that integrates the elements for a goal-oriented adaptation process in which the system is exercised with the injection of disturbance that lead into a control theoretical verification.

The design process is represented by three parallel lines that merge into one that in a big picture ensembles a feedback loop, see Figure 3.1. The first line, withdraws the Contextual Goal Model (CGM) and transforms feasible tasks into components that compose the target system. The second, derives from the parametric formula which is used on the adaptation engine design. Both lines, merge into a single one which culminates into a self-adaptive system (SAS) through an integration process, which finally is configured by the system engineers and it is ready-to-go to runtime. The third line consists of setting uncertainty scenarios which will exercise the SAS in a runtime simulation in parallel to the desired properties specification for verification purposes. During runtime, information regarding the system behavior is collected and a control theoretical analysis is performed. This provides the system engineers with quantitative evidence on how the SAS behaves in the presence of uncertainty and may lead to refinements into the adaptation engine design, closing the feedback loop and contributing to trustworthy adaption engines design.

Our architecture provides means for performing each of the aforementioned processes that contribute on the provision of concrete means for the adaptation engine verification process. It is composed of four major layers, two horizontal and two vertical. Wherein the *Knowledge Repository* centralizes the information necessary for reasoning on the adap-

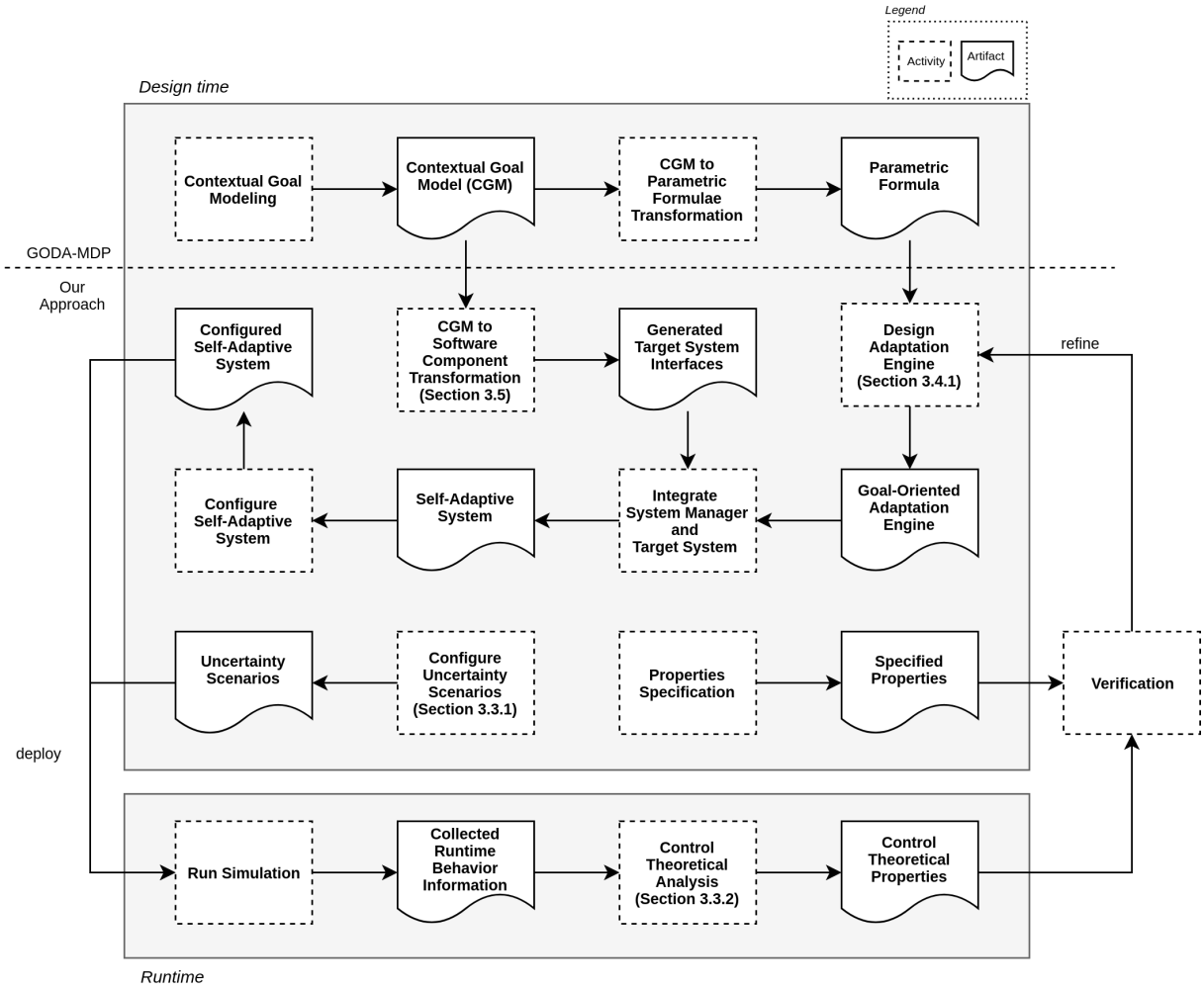


Figure 3.1: The process view

tations to be taken, the requirements in goal-model fashion and the runtime behavior information. The *Analysis* layer contains the components for exercising scenarios into during runtime and the verification of the system behavior. The *System Manager* layer is represented by the adaptation engine and strategy enactor components which guides the system to fulfill its adaptation goals. Finally, the *Target System* layer is composed by the system to be adapted and the components that monitors and effects on it. Figure 3.2 depicts the architecture where white round-shaped squares represent procedures and white sharp-shaped squares artifacts, arrows represent the direction of the data flow, and, as a result, dependency between components.

The next few sections furthers down into each layer of the architecture presenting conceptual aspects of the containing components and how they pervade the design process.

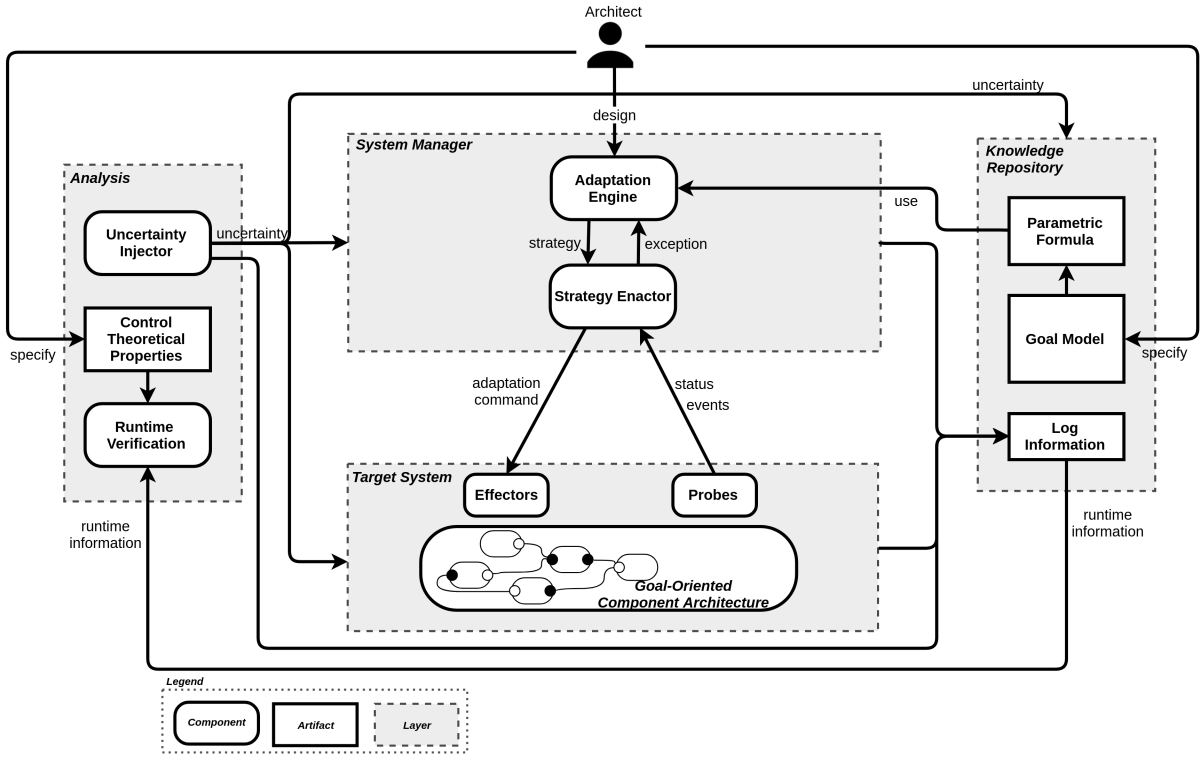


Figure 3.2: The architecture

3.2 Knowledge Repository

The knowledge repository is orthogonal to the other layers since it contains the models necessary for the reasoning procedures at all levels. As in this approach we advocate the importance of a goal-oriented design-process, the models suggested are goal models, following from goal-oriented requirements engineering literature [12]. The knowledge repository content can be fed either at design-time through stakeholders (e.g. requirement engineers, product owners, analysts) or at runtime through mechanisms that collect system information for requirements and models improvement.

The goal model is subject to input/output operations at any time, since the stakeholders and automatic inference mechanisms might guide the system towards new or updated goals. It is desired that the model updates are performed without any system outage because the components might rely on the goal model at runtime regarding the decision-making process. Then, the goal model must be ready for use in persistent storage or in-memory. Persistent storage approaches tend to be more resilient to faults while in-memory provides better performance, a hybrid approach might be eligible when trade-offs between the qualities are required.

Either way, the employed approach should be transparent to the goal model consumers and producers. Therefore, an interface must provide the high-level operations on the

model with guarantees that no inconsistent state is present and that all layers have access to the same model. Each layer requires distinct perspectives of the model (e.g. goal tree, symbolical formula), since the decision-making reasoning is subject to performance issues and lower layers usually tend to require faster response in comparison with upper layers. As a result, models automatic transformation is advised in order to guarantee consistency between the models used in different layers.

Following Solano et. al [10] we advocate for the employment of Contextual Goal Models (CGMs) and its respective derived formula for decision-making in our approach. Contextual information may hinder different system configurations which provides the means for adapting the system. Also, in their work, a set of uncertainties may be represented on the parametric formula. This provides flexibility on the adaptation engine reasoning process with not much loss on computational effort. In addition, the transformation from CGM to parametric formula is fully automatic and complies with the necessary mechanisms for assuring consistency between models in the layer. For more information on the transformation process from contextual goal modeling to parametric formula please refer to the Section 2.3.2.

3.3 Analysis Layer

The analysis layer is orthogonal to all other layers. Even though it is not part of the deployed software artifact, it plays an important role by stressing an instance of the designed system for the provision of qualitative and quantitative arguments regarding the adaptation engine quality. It provides the system engineers with a mechanism for evaluating how well the adaptation engine behaves in face of isolated or composed uncertainties before deploy. The current work, inspired by mRUBiS [19], presents two architectural components that are at the core of the analysis layer, the uncertainty injector and the runtime verification procedure. Finally, it suggests how the user may explore uncertainty injection in all levels of the architecture to gather evidences on the adaptation engine behavior correctness.

In the light of the control theoretical analysis method, the process can be divided in three distinct steps: configuration, execution and verification.

Configuration. Consists of setting up an execution scenario within the model or application to be exercised. It includes the configuration of each architectural component, duration and the means by which the model or system will be exercised, i.e. the input signals. The first strictly depends on the architectural components domain specific task to be realized, the second can be either mathematically predicted or empirically defined

by observing the time necessary for convergence or divergence and the third depends on which uncertainty source should be exercised and the signal the best defines it.

Execution. The system or the model is stressed through cycles of uncertainty injection and the persistence of the variables of interest evolution in time. During this step, the configured injector mechanism acts upon the application by enforcing noise by altering structural system elements, i.e. parameters, flooding buffers, swapping functions or resources.

Verification. Collecting the data gathered during runtime, processing it and synthesizing information w.r.t the property(ies) and plotting the timeseries that define the system evolution in time. The timeseries goes under thorough information extraction and comparison in terms of control theoretical based metrics. Finally, the extracted metrics can be compared to the desired properties.

3.3.1 Uncertainty Injector

Deviation on the delivered service is usually perceived at runtime, when an error turns active mostly due to emergent uncertainties. Despite of that, the uncertainty source might have been placed at any time. The injector as a runtime architectural component is responsible for simulating the emergence of uncertainties during the execution, which is not strict only to runtime related uncertainties. Therefore, this subsection discusses uncertainty injection in a broader scope. The uncertainty sources compliant with the current work are based on Weyns et al. [13] compiled from Hezavehi et al. [5] into six different classes of uncertainty comprising twenty-three sources either isolated or composed into more complex types.

Within the System Itself class of uncertainty, the model uncertainty sources (i.e. **simplifying assumptions**, **model drift**, **incompleteness**) are inherent to the accuracy of the model used in the adaptation reasoning process. Therefore, model uncertainty injection requires that simpler, incoherent or incomplete system models override the models used in the adaptation reasoning process. However, models are not only domain specific but may pursue unique specification rules, since modeling techniques are constantly improved. As a result, the injector should provide means for periodically injecting preset models into the *System Manager* layer, overriding the complete and concrete models loaded from the knowledge repository. In addition to the System Itself class, the **adaptation functions** source of uncertainty might as well be affected by uncertainty injection by the simulation of misbehaving probes and effectors. This could be either through noise in the propagated information or failures that prevent the information from reaching its destination. Uncertainty injection in the form of noise into the *Logging Infrastructure* can also hinder uncertainty for **automatic learning**.

The Goals class of uncertainty is subject to sources of uncertainty upcoming from the high level system goals, which are local to the *Knowledge Repository*. Therefore, the uncertainty injection is performed by read and write operations in the goal models stored in the *Knowledge Repository*. However, the focus of the uncertainties to be injected are taken prior to the simulation, for design-time sources of uncertainty (i.e. **requirements elicitation**, **specification of goals**), and during the execution, for runtime sources of uncertainty (**future goal changes**). The design-time sources are related to the accuracy with which the system was modeled and requires that pre-set faulty goal models be provided in the configuration phase. The runtime sources encompasses changes in the goals which can be realized by noise injection in the goals parameters at runtime.

Additionally, the Context class of uncertainty is strict to runtime. Uncertainty injection for this class are realized within the target system. Such system interacts with the environment and might have distinct modes of operation depending on the context it is emerged in. Then, noise can be injected in the variables representing contextual data inside each component in the target system for the **execution context** source. Also, applications with sensors for perceiving the context are subject to **noise in sensing**, due to accuracy, interference with other elements, and so on. The injection is thus straightforward.

It is notable that Human Interaction class of uncertainty is out of the scope of this work. It is domain specific and the variety of options for human interactions in the goal-oriented design-process is wide enough and prone to innumerous possibilities.

3.3.2 Runtime Verification

The runtime verification is responsible for performing the tasks from raw data collection, processing and transformation into meaningful information to finally extracting relevant properties. In this work, the verification step is performed on system properties that evolve in time, a timeseries. First, the data collection is realized on the information persisted by the *Logging Infrastructure* into logs. Second, the procedure performs a processing and transformation of actuation signals, input signals and the system status into a timeseries response. Third, the properties extraction are placed in respect to the aforementioned control-theoretical metrics 2.1.

In this work, several components contribute to each other for the fulfillment of a main application goal represented by the concrete goal model tasks and decompositions. Then, the processing and transformation step should compose the information persisted for each of the components into a single overall behavior of the system, as in many cases the properties to be guaranteed are in respect to the general system behavior. The parametric formula derived from the CGM and stored in the *Knowledge Respository* can be employed

for verification purposes. As it provides enough information for being used for control theoretical verification even in face of uncertainties. All in all, it supports contextual information modeling, system properties representation and is prone to uncertainty.

In Figure 3.3, a typical response to a perturbation in the system is illustrated.

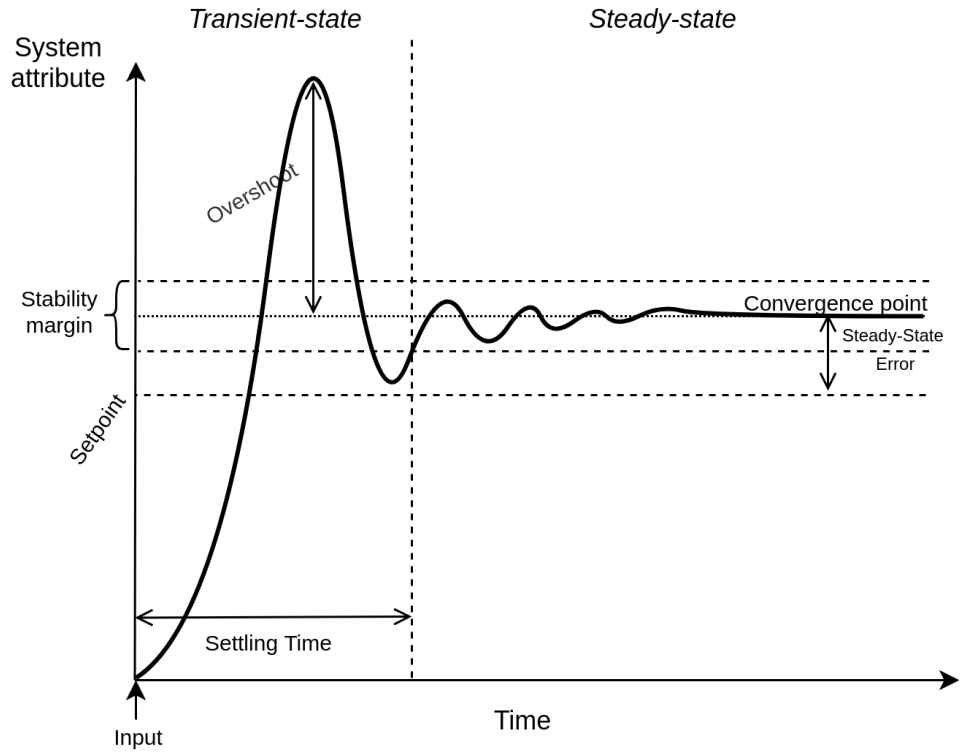


Figure 3.3: System behavior and metrics

The figure represents the outcome of the processing and transformation phase, which paves the way to the properties extraction step, further discussed in the next subtopics. Given that the outcome of the processing phase is an array of size N containing the values that represent the system response at each instant (TS).

Stability

A system is said to be stable when it reaches its state of steadiness. In Control Theory, the steady-state is defined by the interval in which the response curve stays inside the acceptable stability margin from the convergence point (usually 2% or 5%). In this work, the convergence point is calculated by the simple average of the last quarter of the timeseries values, Equation 3.1.

$$CP = \frac{\sum_{n=\frac{3N}{4}}^{N} TS[n]}{\frac{N}{4}} \tag{3.1}$$

The evaluation that defines whether the response reached the steady state or not consists of querying if the last value ($TS[N]$) is within the interval determined by the stability margin ($-SM < TS[N] < +SM$).

Settling Time

The settling time is time required for the response curve to reach stability. Marked in Figure 3.3 by the crossing from the vertical dashed line with the lowermost dashed stability margin line. From the settling time on, it is guaranteed that the response curve stays inside the stability margin.

Algorithm 1 Calculate Settling Time

```
1: procedure CalculateST
2:     ST ← 0
3:     for each sample in TS do
4:         if −SM < sample < +SM then
5:             if ST is 0 then
6:                 ST ← sample instant
7:         else
8:             ST ← 0
       return ST
```

The Algorithm 1 returns the first sample instant that is inside the interval ($-SM < sample < +SM$) until a sample is out of the interval or the array is over. When the response curve is not stable, the algorithm returns 0.

Overshoot

The overshoot, or maximum (percent) overshoot, indicates the maximum peak value of the response curve measured from the convergence point. The Equation 3.2 illustrates the mathematical model used for calculating the overshoot.

$$OS = \frac{TS[t_p] - CP}{CP} \cdot 100\% \tag{3.2}$$

Where the t_p is the sample related to the highest peak on the response which can be extracted with a maximum function iterating over the TS array.

Steady-State Error

The steady-state error is a measurement of how far from the precise setpoint value has the system response reached. It is calculated by the relative distance between the

convergence point and the setpoint, defined by the system stakeholders, see Equation 3.3. Be *setpoint* the value determined as the desired final state for the response curve,

$$SSE = \frac{|\, CP - setpoint \,|}{setpoint} \cdot 100\% \tag{3.3}$$

3.4 System Manager Layer

The *System Manager* layer contains the architectural components for adapting the target system. It constantly monitors the system status and triggered events. Then, based on a model and the system requirements, an analysis is performed towards deciding whether it is needed to adapt the system. Furthermore, a planning routine decides a strategy that might lead the system towards reaching its goals. At the end of a cycle, the strategy is executed. This structure might be repeated on as many sublayers inside the system manager layer as the engineers find useful. Distinct layers can be applied by means of performing hierarchical adaptation in which lower layers propagate exceptions upwards when they cannot find a strategy that fulfills the current goal, due to contextual constraints. Thus, upper layers which perform higher level reasoning processes may find a sufficient or the best strategy for the situation, propagating it downwards.

We present a minimal two-layered architecture containing an adaptation engine for higher level reasoning and a enactor for lower level strategy enforcement. However it is not our intention to limit the number of layers, since our goal is of verifying different proposals of adaptation engines. The two following subsections dive into major theoretical aspects of both architectural components.

3.4.1 Adaptation Engine

Responsible for higher level reasoning, the adaptation engine architectural component must be capable of performing computationally expensive decision-making algorithms. Search in adaptation space is particularly difficult to solve as the size of the adaptation space grows in exponential order since it is prone to the combination of possible solutions. Specially when it accounts to not only the execution or not or each target system component, but the interaction of components to satisfy the goal. Which increases even more when more than one parameters for each components are taken also considered.

Despite the difficulty, approaches propose goal-oriented requirements formalization by enabling uncertainty to be modeled in goal-oriented notation [7, 10, 26, 27]. Reasoning upon these models is still not free from state explosion, but constraints can be elicited to build heuristics which lead to feasible solutions. The parametric formula employed in this work is used for both analyzing the current status of the overall system property

of interest and for calculating through a search-based solution, the combination of local (goal, condition, actions) that would lead the system into reaching its objective.

The Figure 3.4 details the feedback loop corresponding to high-level processes employed the strategy synthesis. Where a P_{ref} is the system overall property to be constantly monitored and guaranteed during runtime, it is provided by an external agent. The *error* is composed by the difference of where the system currently is (P_{curr}) and where it wants to get (P_{ref}). The *need adaptation?* block, queries if the error is bigger than the expected. And the *evaluate actions* loop through the parametric formula, searching for the combination of terms that would lead the system into reaching P_{ref}. When the combination is found it is propagated to the strategy enactor in the form of strategy.

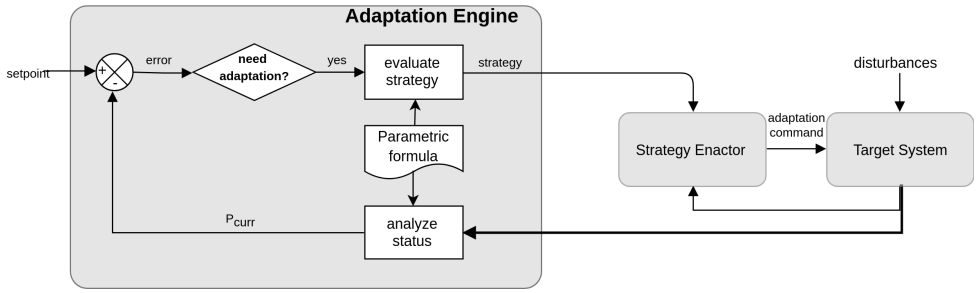


Figure 3.4: Detailed vision of the adaptation engine

The reasoning process follows the Algorithm 2 written in pseudo-code. Where P_s is the set of properties monitored from the system, labeled as "system information" upon the thick line in Figure 3.4. Line 4 calculated the error based on the desired P_{ref} and the current P_{curr}. Line 5 corresponds to the analysis on whether the adaptation needs to be performed, querying if the error is within the acceptable stability margins. And from line 6 - 21, the *evaluate action* block is performed by a search with heuristics within the possible adaptation space. Finally, line 22 corresponds to the strategy selection and propagation to the *Strategy Enactor*.

The *evaluate action* block from line 6 to 21 uses incremental gains on the error for reaching the desired reference value, other strategies could be employed for reaching the desired value. The algorithm is run periodically in a preset *actuation frequency* and collects a configurable *quantity of status messages* from the target system in order to calculate the current system property. It performs a search in the solution space ($\mathbb{R}$), that starts in the value determined by a percentage of the current property value, determined by the *offset*. For example, if the current system property value is 0.80, the offset is 20% and the error is bigger than 0, then the search starts at 0.54 for each and every component. In the case of the error being smaller than 0, the offset is applied as an increment to the current property value. Also, the Kp determines the *granularity* of each search step since. It is noteworthy that smaller Kps might lead the into more precise

Algorithm 2 Adaptation Engine Pseudo-Code

```
1: procedure AdaptationEngine
2:     P_s ← monitor system
3:     p_curr ← apply P_s to parametric formula
4:     error ← p_ref − p_curr
5:     if −stabilityMargin < error < stabilityMargin then return true
6:     for each p_i in P_s do
7:         P_s ← ResetToStartingPoint(offset)
8:         p_new ← apply P_s to parametric formula
9:         if error > 0 then
10:            p_i ← p_i + Kp · error until p_new > p_ref
11:            where p_new ← apply P_s to parametric formula
12:            for each R_j in P_s − p_i do
13:                r_j ← r_j + Kp · error until p_new > p_ref
14:                where p_new ← apply P_s to parametric formula
15:        else
16:            p_i ← p_i − Kp · error until p_new < p_ref
17:            where p_new ← apply P_s to parametric formula
18:            for each R_j in P_s − p_i do
19:                r_j ← r_j − Kp · error until p_new < p_ref
20:                where p_new ← apply P_s to parametric formula
21:        Strategies ← Strategies + P_s
22:    Send a strategy from Strategies that satisfy the stabilityMargin
```

solutions and might even be necessary for reaching a solution. However, the smaller it is, the bigger is the search space, which can invalidate the algorithm in terms of scalability. Finally, the p_{ref} determines the setpoint to be achieved by the search. Thus, the algorithm can be configured in terms of these parameters.

3.4.2 Strategy Enactor

The enactment level architectural components are responsible for enforcing one or more active strategies at the target system. The strategies are synthesized in the adaptation engine and must be read, interpreted and enforced by the strategy enactor. It is the strategy enactor's responsibility to evaluate whether the active strategy truly leads the system into the desired behavior, thus monitoring the system events and status must be constantly performed. If the active strategy does not lead the system into fulfilling the system goals or no strategy among the present ones could lead the system towards its goals, exceptions might be thrown to the upper layer with statements or constraints that supports the adaptation engine strategy synthesis.

As illustrated by Figure 3.5 the active strategy is composed by Goal, a Condition and an Action. The goal placeholder determines the property reference to which the component should converge to. The condition is used to determine whether the current error satisfies the requirements. And the action drives the system into reaching the desired goal. The strategy enactor interfaces directly with the target system by sending adaptation commands and receiving system information through status and event messages.

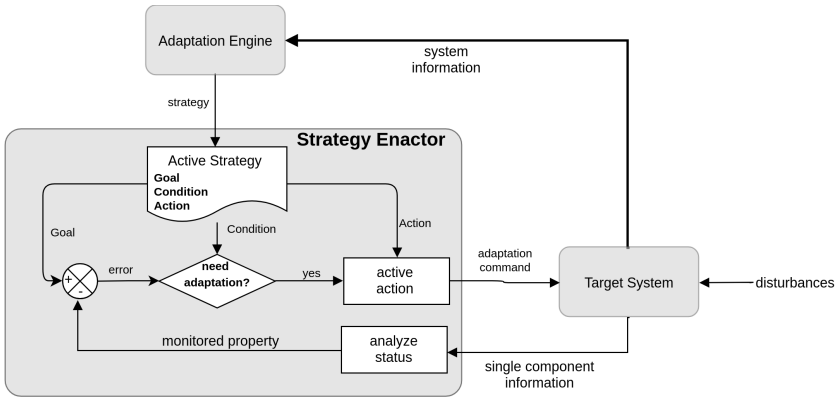


Figure 3.5: Detailed vision of the strategy enactor

3.5 Mapping from Goal-Model to Components

According to Szyperski et al. [28], *"a software component is a unit of composition with contractually specified interface and explicit context dependencies only"*. From such definition, we derive that the most important aspect of components is the separation between implementation and interface that enable seamless integration with other components. In the goal model, the means-end tasks suffice the requirement of being an independent entity that completely fulfills a higher level goal at a concrete level. Then, composed with others, such tasks contribute to the fulfillment of the main goal of the application.

Without loss of generality, in this work, we propose a mapping from means-end tasks to components. In this case, we assume that each component in the architecture encapsulates the tasks that decompose the respective means-end task. Also, this mapping does not limit the software design decisions in terms of what is a component and how the means-end tasks decomposition compose with each other. They might be designed as functions, methods, classes, modules or whatever software building block sound reasonable to the system engineers. Despite of that, it is required that a mathematical model relating the composed building blocks to the property to be adapted is provided to single means-end task. Which can then be plugged into the parametric formula.

It is noteworthy that the mapping suggested in this work is no more than a contract between the requirements engineering process and the implementation of the system. The components correct implementation and binding is left to the system engineers and is a manual process.

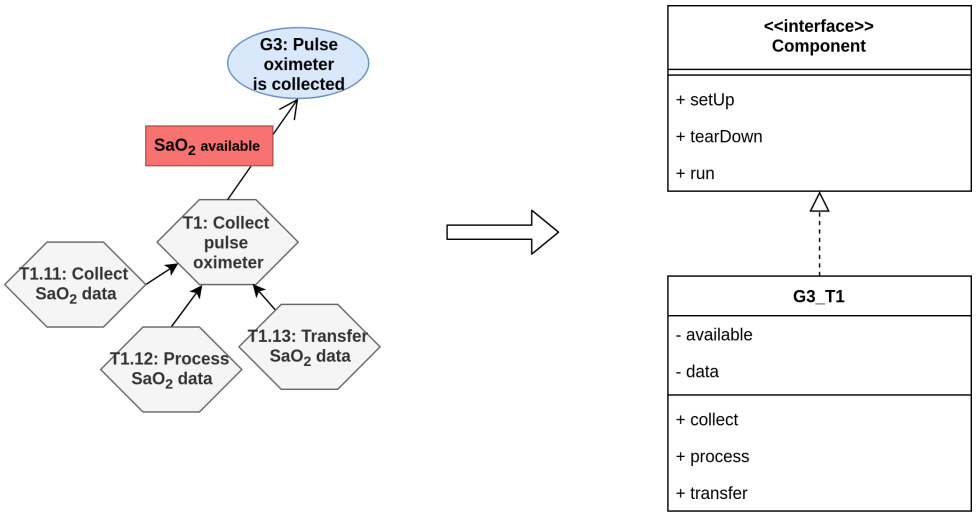


Figure 3.6: Goal to component mapping example

In the illustrated example of Figure 3.6, we demonstrate a mapping between means-end tasks, 'T1: Collect pulse oximeter', into the class $G3\_T1$ that implements the methods defined by the interface class *Component*. The class $G3\_T1$ contains *available* and *data* as attributes, where the *available* is a boolean value that evaluates in correspondence to the contextual information of the task and the *data* contains the information to be exchanged through the leaf-tasks implemented as methods *collect*, *process* and *transfer*. A pseudo-code containing the implementation of the *Component* inherited methods is represented in Algorithm 3.

Algorithm 3 G3_T1

```
1:  procedure SetUp
2:      available ← True
3:  procedure TearDown
4:      available ← False
5:  procedure Run
6:      if available is True then
7:          isCollected ← collect(data)
8:          isProcessed ← process(data)
9:          isTransferred ← transfer(data)
10:     else
11:         skip
        return (isCollected and isProcessed and isTransferred)
```

Whereas the implementation of *collect*, *process* and *transfer* behaviors are in charge of the system engineer. In the case of OR decomposition instead of only ANDs, the return statement would change to suffice the first-order logic derived which state whether a run cycle was successfully performed or resulted in a failure.

3.6 Target System Layer

The target system layer contains the system that is ought to deliver the service that fulfills the functional requirements and mechanisms for monitoring and adaptation. Such layer is subdivided into three architectural components: probes, effectors and system.

The probes are meant for system's data collection and pre-processing data gathered via instrumentation internally to the component or upon the communication infrastructure. Either way, the probes shall continuously evaluate emergent changes into the system state that are relevant to the property to be adapted. The information that evidences the changes should be collected and propagated to decision-making structures, e.g. the *system manager* layer.

The effectors, in turn, are responsible for acting on the target system. The actuation may refer to either behavioral or reconfiguration aspects of the system. Each system component must be instrumented with actuation mechanisms that enable change during runtime. Behavioral changes are domain dependent and subject to the provided services from each component. The reconfiguration can be either domain dependent or independent. While the dependent is prone to the system parameters, the independent deals with components insertion, removal and binding.

Finally, the system to be adapted's architecture should be flexible, though following some basic principles to ensure runtime adaptability is advised. In this work, it is required that each self-sufficient task should be mapped into independent components that upon execution satisfies a well delimited goal. In addition, each component should be mapped into the parametric formula elements. The goals to components mapping provides the necessary structural rules for enacting adaptable components.

3.7 Logging Infrastructure

The logging infrastructure is motivated by the need of monitoring and evaluating the system behavior in face of runtime changes, which is directly related to how well the adaptation engine fits its purposes. Therefore, a transparent logging layer constantly monitors the messages that travel through the communication channels between the *System Manager* and the *Target System* layers.

This work assumes that the manager and target system layers share information through message exchange. Logging can be accomplished in two ways: centralized logging or distributed logging with information aggregation. The centralized solution is simpler than the distributed. However, it demands additional effort for guaranteeing event ordering and for avoiding the bottleneck on systems with many nodes. On the other hand, there is a necessity of time synchronization between the nodes that participate on the logging task in a distributed systems domain. Even though global clock synchronization is a well-established problem in distributed system community, its implementation is harsh and depending on the adopted solution, a communication network overload can be observed. Also, it is required that a log aggregator is deployed to correctly merge the logs, which should not be a problem for globally ordered events.

A fundamental information to be logged for each message that passes by the logging infrastructure is the message's emitting source. Even though the goals to components mapping guarantees that each means-end task are implemented as a singular and self-sufficient component, the logging infrastructure must make sure that this information is associated to the event and status messages propagated upwards by the probe. So, that the *Analyzer* and the *System manager* can correctly map each component property to its respective term in the formula.

Chapter 4

Case Study and Evaluation

4.1 Body Sensor Network a Case Study on ROS

To discuss our proposed methodology, we use the exemplar implementation[1] of a Body Sensor Network (BSN) [29, 30]. The main objective of the BSN is to keep track of a patient's health status, continuously classifying it into low, moderate, or high risk and, in case of any anomaly, to send an emergency signal to authorities on the subject. The structure of the BSN is as follows: a few wireless sensors are connected to a person to monitor her vital signs, namely, a pulse oximeter (SaO_2) for blood oxidation, an electrocardiograph (ECG) for heart rate, a thermometer (TEMP) for temperature and an arterial blood pressure monitor (ABP) for systolic and diastolic blood pressure measurement. Additionally, there may be a central node responsible for analyzing the collected data, fusing it, identifying the patient's health status and finally emitting an emergency message if necessary.

A more precise description of the BSN is detailed by its requirements that are represented by a goal-model, as depicted in Figure 4.1. The goal model goes through a vertical and exogenous transformation for algebraic formula derivation [10] and is mapped into the target system components as suggested in this work. Modeling a SAS requires to take into consideration not only the requirements and means to achieve them, but also the contextual information that may be related to the system's operation. For this purpose, we use a Contextual Goal Model (CGM) since it allows us to specify in a simple structure the stakeholders and high-level requirements, the ways to meet such requirements, and the environmental factors that can affect the quality and behavior of a system. According to Figure 4.1, the Body Sensor Network, actor, and its main objective is elicited as the root goal "G1: Emergency is detected". Then, G1 is decomposed into a soft goal and a hard

[1]The case study is avaliable at `https://github.com/rdinizcal/master_thesis_eval` along with all data generated and processed for the experiments presented in this chapter.

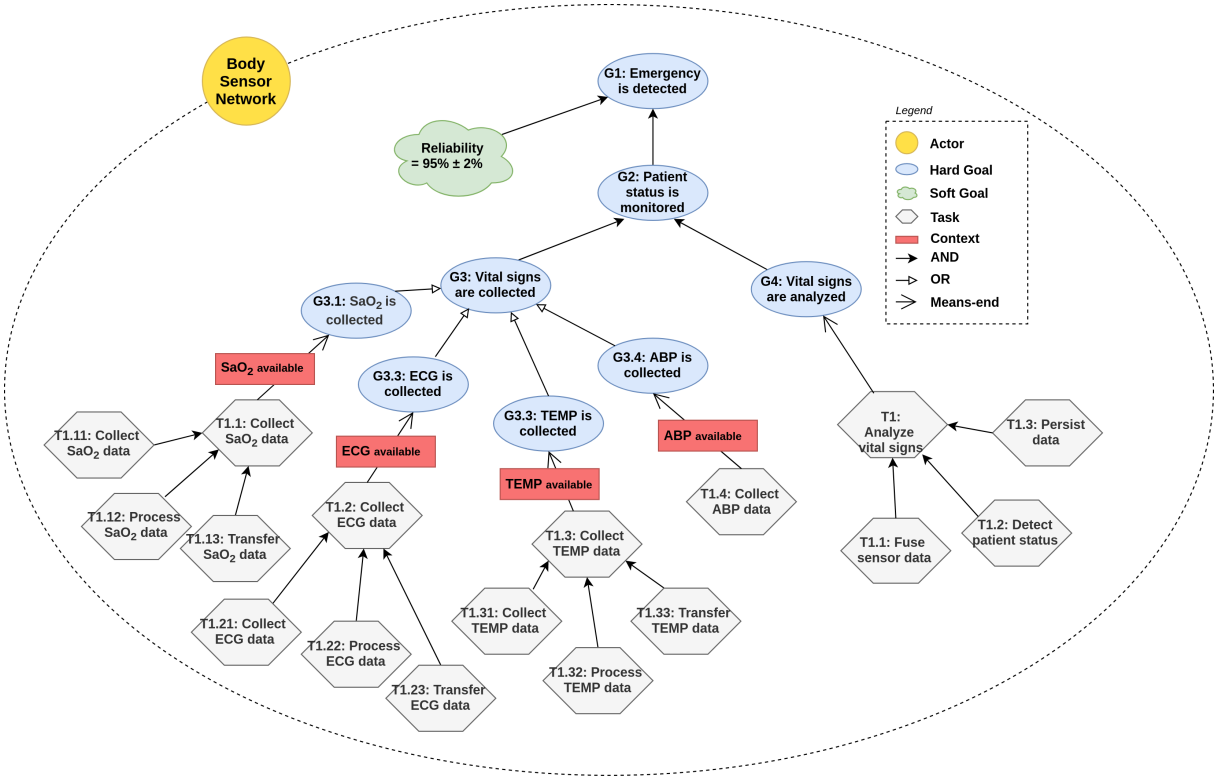


Figure 4.1: Body Sensor Network Goal Model

goal, as for the soft goal, it is stated that G1 must be achieved with 95% ± 2% reliability, and for the hard goal that the patient should be monitored, "G2: Patient status is monitored". Next, G2 is decomposed into "G3: Vital signs are collected" and "G4: Vital signs are analyzed", through an AND decomposition since both must be realized. The former is decomposed on 4 goals with an 'OR' decomposition indicating that at least one should be realized to fulfill its objective, for the collection of specific sensor's data, which are realized by the means-end tasks T1.1, T1.2, T1.3 and T1.4: collect SaO_2 data, ECG data, temperature data or blood pressure data. The latter is decomposed on the means-end task "T1: Analyze vital signs" which states that the BSN should analyze the vital signs to identify and detect an emergency. Finally, each means-end task is still decomposed on the leaf-tasks, that in conjunction operationalize the means-end task. The operation of the Body Sensor Network is subject to four context conditions: SaO_2, ECG, $TEMP$ and ABP. Each of them refers to the availability of their respective sensors needed to collect vital signs data.

The BSN self-adaptive software implementation follows the conceptual architecture presented in the approach and make use of the Robot Operating System (ROS) middleware for message exchange 4.2. Where the ROS Master is the name server for registering, deregistering and lookup ROS nodes and communication topics. The communication topics are used for peer-peer communication with publish/subscribe protocol. The common

bus is provided in default as a TCP/IP connection between the ROS Master and the nodes and within nodes. Each architectural layer was implemented as a ROS package. Each ROS package, provides its own manifest, set of messages, provided services, codes (nodes in this case) and ros launch for command lines configuration. We further detail the nodes behaviors for each package in the next subtopics.

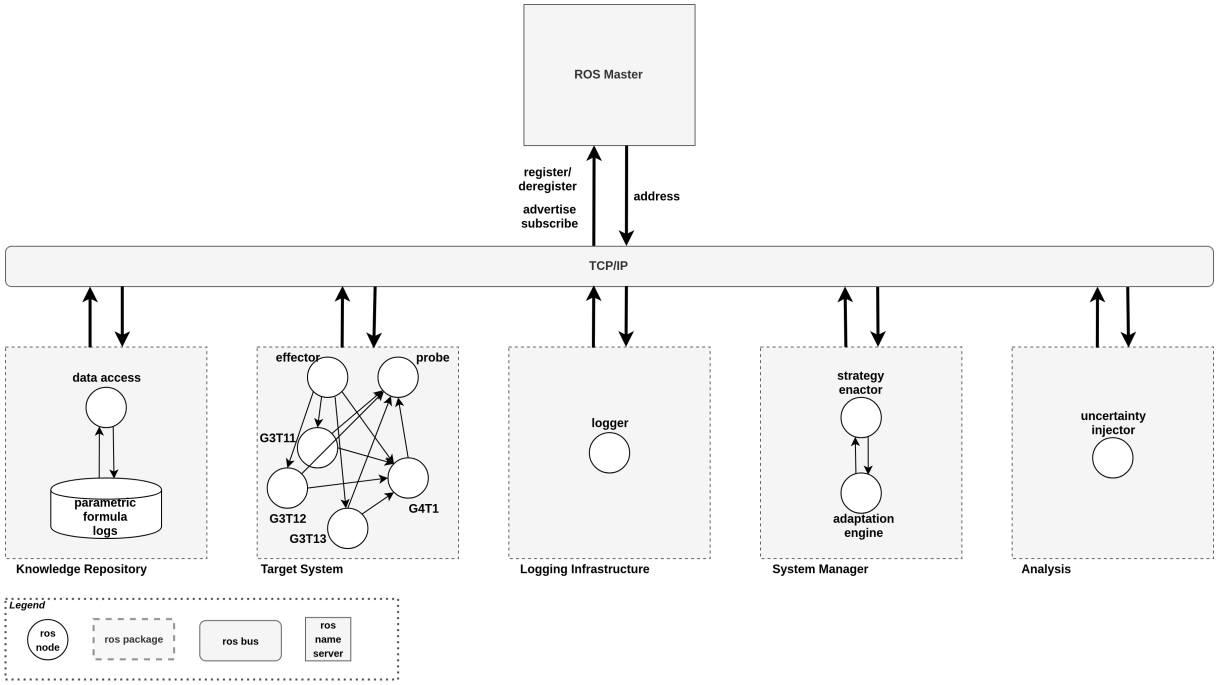


Figure 4.2: Architectural view from the BSN implementation on ROS

4.1.1 Knowledge Repository

The *Knowledge Repository* package contains a data access node, a parametric formula and persisted logs during the execution. The data access node was implemented following a hybrid approach. It stores data in-memory and persists it in bursts once in a while. Then, the most recent data is ready to whoever wants to consume it during runtime and is being persisted for the analysis layer. The data access node persists data coming from the logging infrastructure and mostly responds to data requests coming from the adaptation engine, as illustrated in the activity diagram example from Figure 4.3. It contains data structures for supporting fixed size buffers of adaptation commands, status, events and uncertainty injection messages. Which are occasionally persisted in separate files where information regarding the instant issued, source, target and content are persisted per file line.

It's unique configuration parameter is the execution frequency, set to 10KHz to avoid bottlenecks to due data persistence and mostly to avoid data loss. The parametric formula

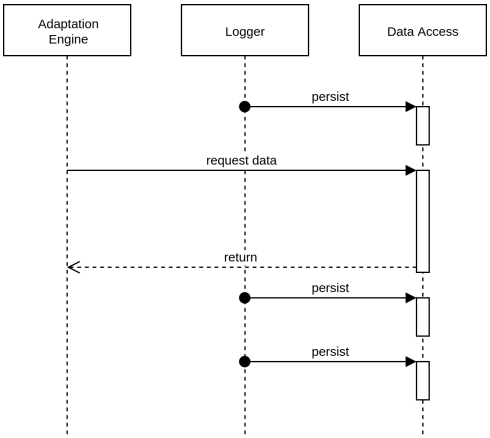


Figure 4.3: Exemplar activity diagram of data access

is stored in the reliability.formula file. The logs are named after the type of message saved incremented to a timestamp.

4.1.2 System Manager

The *System Management* layer contains two ROS nodes: the adaptation engine and the strategy enactor.

The adaptation engine responsibility is to guide the system adaptation into achieving a overall desired level of reliability, specified in the goal model by the soft goal $Reliability = 95\% \pm 2\%$, Figure 4.1. The node configurations are in respect to the adaptation engine Algorithm 2 parameters. And is left to the user to set: the monitoring frequency, the reliability setpoint, the actuation frequency, the amount of messages the system should request to the data access, a offset and a gain (Kp).

The strategy enactor continuously monitors the target system by the processing of *Status* and *Event* messages receipt through pub/sub. In the case of the BSN, the system reliability is being monitored, therefore, status messages that inform whether an invocation of the target system component has succeeded or not is stored in buffers for each active component. At a defined frequency the buffers are read and the reliability of each component computed. The reliability status might trigger active strategies into adapting the system by publishing messages containing the adaptation command.

Figure 4.4 represents a typical execution in which an adaptation is issued to the strategy enactor. The enactor dissembles the strategy upcoming and apply changes to the innermost control loop: goals, conditions and actions. With the defined setpoint (the new goal), the strategy enactor emits messages to the effector with the adaptation command and target component. In defined frequency the probe collects information

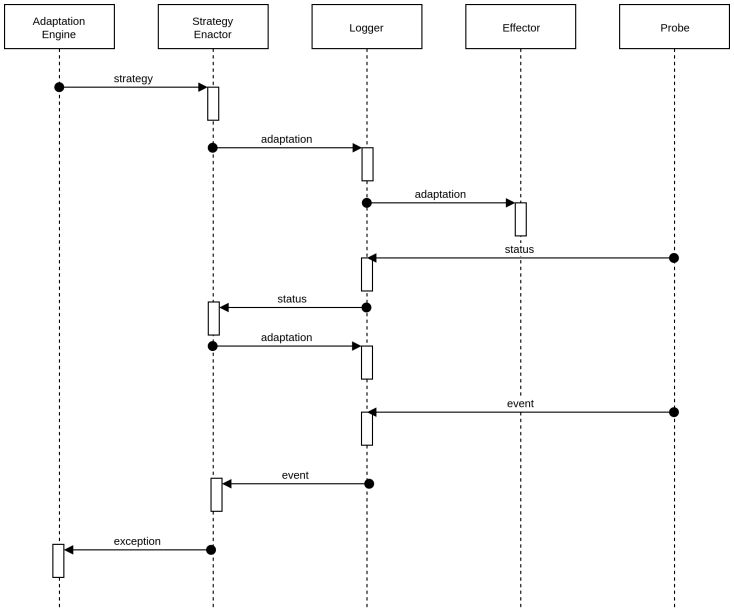


Figure 4.4: Exemplar activity diagram of system manager

from the target system components and propagates it to the adaptation engine, passing through the logger and the strategy enactor.

Two strategies were hardcoded: (i) data collection replication and (ii) sensor frequency adjustment. The first strategy relies on replicating the the sensors' data collection task and performing an average of the collected data, which is more likely to succeed. Then, the system manager constantly monitors each sensor's reliability status and adjusts the number of replicas of the data collection task in order to fulfill the reliability reference requirement. The second is about adjusting the sensor's invocation frequency for reducing package loss due to message buffer overflow in the central hub. The strategy follows the same proportional control approach despite that the controlled variable is not the number of replicas, but the sensor's frequency. Also, the relation between sensor frequency and reliability, due to message loss in the central hub is inversely proportional, which was not the case for number of replicas.

The strategies implemented were coded as increments of proportional gains in a feedback loop style for each component. From which we derive the mathematical model 4.1 that represent the feedback calculation to proportionally achieve the desired x. Where $error = p_{ref} - p_{curr}$, p_{ref} is the desired reliability for the local component, p_{curr} is the current reliability that is calculated using Equation 4.3 and Kp is a gain that is empirically defined based on the desired response. The Kp defines the amplitude of the steps towards reaching the desired p_{ref}.

$$x = Kp \cdot error \tag{4.1}$$

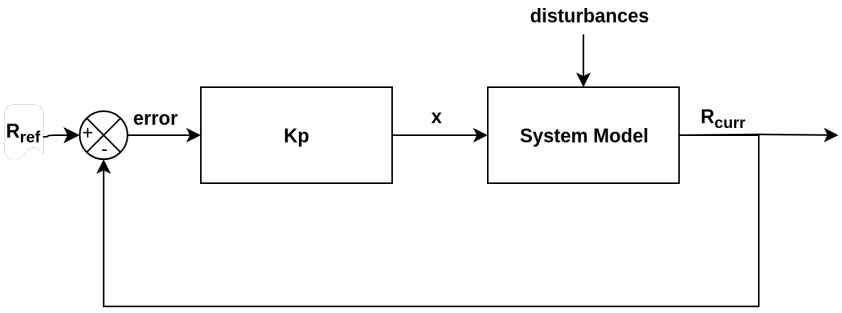


Figure 4.5: Feedback Loop for Proportional Control

4.1.3 Target System

The *Target System* package contains the effector node, the probe node and the nodes which operationalize the functional requirements of the BSN.

The effector's objective is to actuate on the system based on the information upcoming from the strategy enactor. In this case study, the adaptations are realized by the reconfiguration of the components parameters, thus the implemented effector forwards the pair (action, value) to the component through publishing an effect message with the content upcoming from the upper layers. For example, the replicate data collectors adaptation message could contain the following string: "$replicate\_collect = 5$", whereas the action is the "*replicate_collect*" and the value is 5.

Similarly to the effector, the probe is part of the communication interface between the *System Manager* and the BSN. Its responsibility is to reroute the status and event messages upcoming from the BSN components. These messages can indicate success or failure in the component execution or contextual information in respect to its activation status gathered at the component.

The components follow the goal to architecture mapping suggested in the approach. Thus each component, implemented as ROS node, provides the services described by the means-end tasks from the goal model, see Figure 4.1. The BSN components can be divided in two types: the sensors and the central hub. The sensors simulate data collection following a probability distribution generator modeled as first-order markov chains, which simulate data collection from a patient that has probabilities of transitioning from health states. Beyond the data collection, two other sequential tasks were implemented that determine the higher level "Collect Sensor Data" leaf task in the goal model, the data processing task and data transmission. The data processing in this case contains a procedure that employs a moving average filter and the data transmission builds a message and publishes it in a topic that the central hub constantly reads. The central hub is responsible for providing the analyze vital signs behavior, thus, it implements (i) a

procedure for collecting data from the topic and storing in an internal buffer, (ii) a data fusion procedure and (iii) a persistence procedure. The procedures are sequential and their execution must all succeed in order to fulfill the higher level "Analyze vital signs" leaf task.

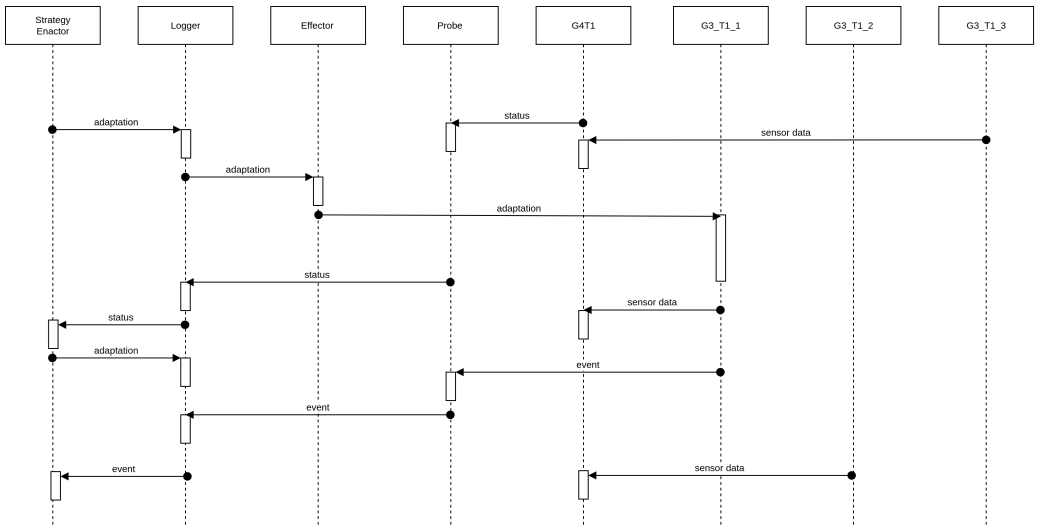


Figure 4.6: Exemplar activity diagram of target system

Figure 4.6 illustrates one of the possible message exchange situations, where the $G4T1$ implements the Central Hub and the others implement the Sensors. Yet, according to the figure the sensors transmit sensor data messages to the central hub. These contain the collect and filtered data for that component. Also, there is intense communication from the system nodes to the effector and the probe.

For both sensor and central hub, the initial configuration requires that the nodes frequency be specified. It is notable that at least three sensors are constantly sending messages to the central hub. If the central hub's processing speed is lower than the sensors transmission frequency, it might lead to message loss in the central hub since the communication buffer has got limited size. Therefore, we suggest that the central hub frequency is set as at least 3x the biggest sensor frequency. Among that, the sensors must be configured with transition matrices of 5x5 representing the sensor markov chain used for data generation. E.g. the thermometer measures at least $25°C$ and at most $50°C$ and we classify the range $40°C$ to $50°C$ as represented by a high risk to the patient. Figure 4.7 represents a visualization of an example of markov chain, in which each state represent the range of possible values. Where the green state is represented by the low risk range, the yellow are the moderate risk range and the red are the high risk range. The probability of transitioning between states depends on the patient profile.

Then, the sensors invoke three tasks in each execution cycle: collect data, process data and transfer data. The data collection consists of randomly getting a data value within

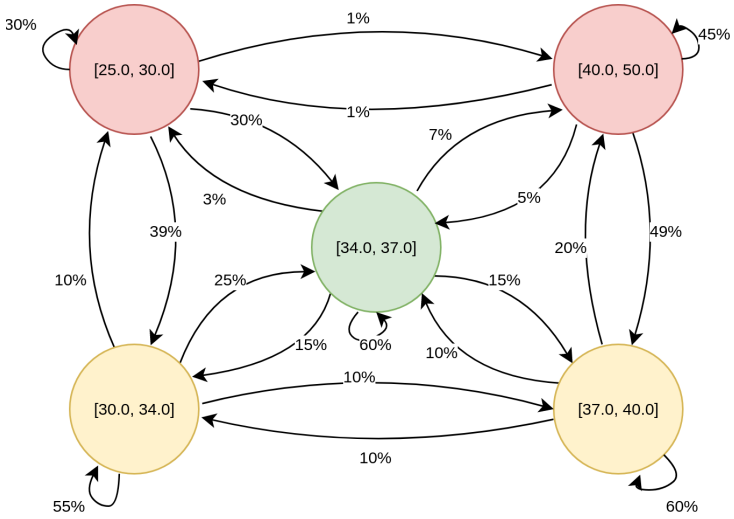


Figure 4.7: Configuration of markov chain for temperature data generation

the range represented by the patient current health status for that sensor. Then, the data passes through a moving average filter which smooths the collected data based on the last Nth data values collected, to avoid outliers. And finally, the data is transferred as sensor data message to the central hub. On the other hand, the central hub procedures consists of first consume the sensor data that arrived in the topic. Then, fuse the sensor data with the last states of each of the sensors data. This procedure works by normalizing every sensor data and combining them with a simple average function. At last, the average value is considered at the total patient health risk status and if it is above or below a threshold, the patient is considered to be in risk and an emergency alert is thrown.

4.1.4 Logging Infrastructure

The *Logging Infrastructure* package contains the logger node.

The implemented logger node objective is to collect messages traveling between the *System Manager*, *Target System* and *Analysis* layers. It is implemented as node in the middle that receives every message that travels in between layers. Every message received is converted into a neutral persist type of message in which every field is copied from the original, the persist message is sent to the data access node and the original is re-routed to the previous target. Every message received comes with the source name, that represents the node that emitted the message, including the ones in the target system. Therefore, every message traveling from node to node has a mapping from the source node produced which is fundamental for the goal to architecture mapping. Also, the problem of distributed synchronization is solved in this implementation by collecting solely the logger timestamp, which is the one sent for persistence. Ensuring an order in the events in a satisfactory but simple solution.

4.1.5 Analysis

The *Analysis* package contains the injector node and python scripts for performing control theoretical analyses from the logged information.

In this case study we explored the noise in sensing uncertainty by employing a parameter modification approach with the uncertainty injector. First, the sensor nodes were equipped with subscribers listening to messages coming from the injector node. When the message arrives, its content is read and the value is attributed to a previously created parameter, *noise*. The parameter is used for generating noise in the collected data following the Equation 4.2. Where the *error* is a randomized value in the interval $[-data \cdot noise, data \cdot noise]$.

$$data = data \pm error \tag{4.2}$$

As represented in the activity diagram in Figure 4.8, the injector constantly updates either the logger and each of the sensors noise state.

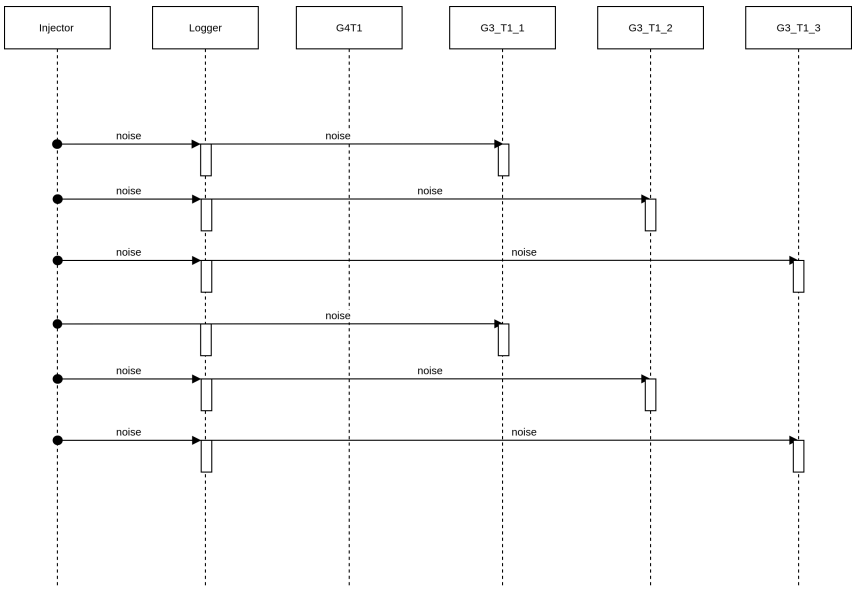


Figure 4.8: Exemplar activity diagram of injector

The injector, just like all the other nodes, enables its execution frequency configuration. Also, it requests that the user enters a list of the nodes that are participating on the uncertainty injection. Their names must be entered on the configuration file. Besides that, the parameters that determine the signal used for uncertainty injection may be configured. The parameters are: signal type, offset, amplitude, frequency, duration and beginning instant. The signal type can be step, ramp or random. The amplitude varies in accordance to the uncertainty to be injected, the frequency in Hz, the injection duration in seconds and the instant that the injection should begin.

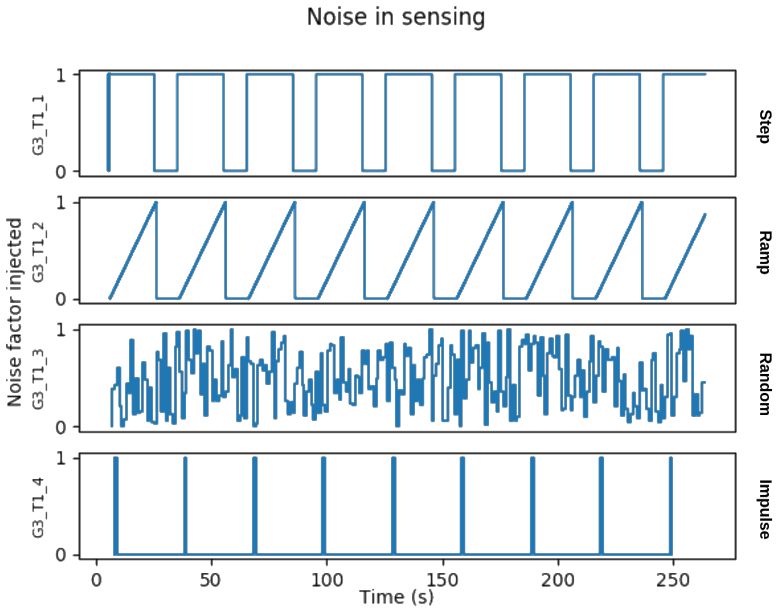


Figure 4.9: Uncertainty injection signals

The analyzer is implemented besides the scope of ROS and is executed only once at the end of the simulation. The logs are fed as the input of the analyzer, which returns a timeseries corresponding to the variation of a system attribute in time, in the case study, the reliability. In this work, we advocate for the use of the parametric formula as well for computing the overall system timeseries based on the system components reliability. The analyzer, first loads the logs collected during the execution and stored under the *Knowledge Repository*, parse and treat the information that is used on the analysis of the system stability, convergence point, settling time, overshoot and steady-state error.

4.2 Evaluation

We claim that the presented architecture enables runtime analysis of goal-oriented adaptation engines w.r.t. control theoretical guarantees for systems operating in scenarios prone to uncertainty. Because, (i) it provides means for the collecting and processing enough information to perform a sound control theoretical analysis and (ii) it permits the injection of monitorable uncertainties during the execution. Therefore, to guide the evaluation of our approach, we follow the Goal Question Metric approach (GQM) [31].

Our experimental evaluation consists of collecting evidence that our approach permits (1) performing control theoretical analysis and verification of whether goal-oriented adaptation engines comply with the specified properties (**Goal 1**) and (2) guaranteeing that the system specifications hold even in face of uncertainty (**Goal 2**). Therefore, we rely

G1: Perform control theoretical verification	
Question	Metrics
G1Q1: Is our control theoretical verification approach sound?	relative error
G2: Provide guarantees in presence of uncertainty	
Question	Metrics
G2Q1: Does our verification approach support runtime analysis of the system operating in face of uncertainty?	control theory metrics

Table 4.1: Goal-Question-Metric definition

on the Body Sensor Network application implemented under the proposed architecture to exercise scenarios that illustrate the questions elicited in Table 4.1.

4.2.1 Performing Control Theoretical Verification

The control theoretical analysis is subject to (i) a mathematical model of the system, (ii) known inputs to exercise the model and (iii) the response to the inputs in time. Since we are concerned with the application's reliability, the evaluation consists of whether the employed reliability formula accurately represents the system runtime behavior or not, in respect to control theoretical metrics (i) convergence point (stability), (ii) settling time, (iii) overshoot and (iv) steady-state error. Furthermore, the correlation between the system's overall reliability calculated using the parametric formula and the monitored system behavior is performed.

Experimental Setup

The experiments were ran in the BSN, implemented in ROS upon the architecture proposed by this work. In this experiment we compare the behavior of a set of executions of the BSN that takes 200s. The behavior is computed through (i) a parametric formula synthesized from a goal model representing the BSN and (ii) the estimated system behavior. The component's local reliability, in this work, is calculated by the Equation 4.3, in which *Success* is the number of succeeded executions and *Fails* is the number of failed executions, in a time frame.

$$R(t) = \frac{Success}{Success + Fails} \tag{4.3}$$

Moreover, the estimated behavior, Equation 4.4, composes the success or not of each means-end task invocation ($G4\_T1$: Central Hub, $G3\_T1\_1$: Pulse Oximeter, $G3\_T1\_2$: ECG, $G3\_T1\_3$: Thermometer).

$$Success = G4\_T1 \text{ and } (G3\_T1\_1 \text{ or } G3\_T1\_2 \text{ or } G3\_T1\_3) \tag{4.4}$$

The model described by Equation 4.4 was derived from the system requirements as follows: when the central hub fails we can state that the system failed on delivering correct service since it is an unique point of failure. Despite that, any sensor that correctly collects the data it is intended to, it is sufficient for processing patient's information on the system. As a result, the 'and' operation between central hub and the sensors in addition to the 'or' operations relating all sensors provide the estimated behavior.

A bottom-up approach is employed to calculate the reliability of the system in time using the parametric formula. When a component is invoked, its local reliability is computed using the Equation 4.3 for every instant within the time window given by the resolution (2.5 seconds). Then, the global reliability is computed by composing the recently calculated local reliability and the last state of each other component's local reliability.

On the other hand, a top-down approach is employed in the system's reliability calculus using the monitored behavior. For every component invocation the Equation 4.4 computes whether the system as a whole failed or not. Similarly to the bottom-up approach, the top-down composes the recent success or failure with each component's last state. The calculated system status are stacked into a buffer along with the instants of invocation. Finally, the buffer is traversed from instant to instant computing the system overall reliability with the Equation 4.3.

Finally, to evaluate whether using the parametric formula in our approach would lead to a sound control theoretical verification, we use the estimated behavior as a reference for performing a statistical correlation and asserting whether it sufficiently represents the system behavior with respect to the control theoretical metrics. Therefore, we calculate the relative error, Equation 4.5 for each and every metric.

$$error = \frac{(V_{expected} - V_{achieved})}{V_{expected}} \tag{4.5}$$

Scenario

We expect to evaluate the control theoretical metrics of the system behavior. Therefore, the system must be subject to variations in the response due to controller actuation. Thus, the components behavior is prone to a stochastic method that generates failures, illustrated by the minimum value around 35 seconds in Figure 4.10.

The Figure 4.10 shows that after 40 seconds the $G3\_T1\_X$ tasks' reliability converges to the $r_{ref} = 0.70$. This is due to the fact that at this instant the adaptation mechanism was triggered. It is notable that the convergence happens in proportional steps since the

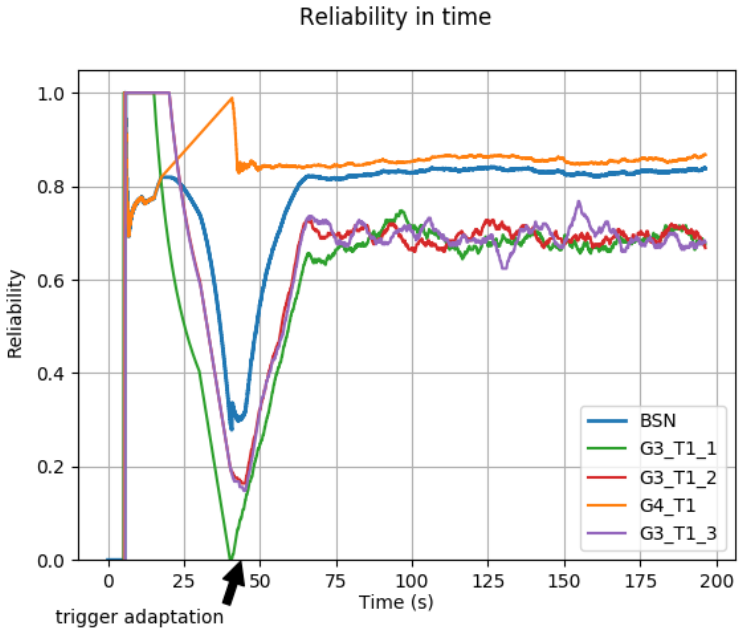


Figure 4.10: Reliability behavior for tasks collection replication control

step size is calculated by the relation between the *error* and a pre-defined gain. This characterizes the response for proportional feedback control strategies.

Results

This experiment aims at building evidence on the trust on the parametric formula for verifying whether the system runtime behavior complies with control theoretical metrics. Therefore, we have collected data regarding the component invocations in respect to success or failure at each execution instant. The data is processed according to the procedures detailed in the experimental setup for both curves. Finally, the analyzer component extracts information on the response stability, its convergence point, settling time, overshoot and steady-state error. It is noteworthy that the analysis is performed on the subset of the timeseries that begins at the moment when the adaptation was triggered to the ending of it. Figure 4.11 depicts the comparison between the behavior and metrics in respect to the parametric formula and the monitored behavior.

The scenario in the BSN was executed 10 times and the control theoretical metrics computed and compared. The summary of the executions are presented in Table 4.2.

	convergence point	**settling time (s)**	**overshoot (%)**	**steady-state error (%)**
formula	0.818 ± 0.0215	2.055 ± 0.447	0.017 ± 0.003	0.091 ± 0.023
monitored	0.816 ± 0.0178	2.329 ± 0.604	0.021 ± 0.001	0.093 ± 0.020
error	0.25%	11.76%	18.90%	2.77%

Table 4.2: Control theoretical metrics summary

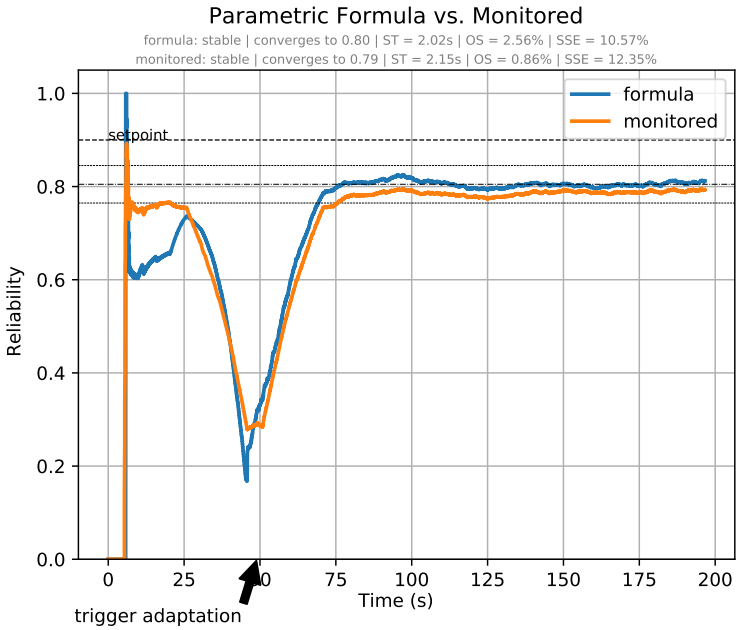


Figure 4.11: Qualitative comparison between parametric formula and monitored behavior

Discussion

The comparison between the parametric formula and monitored reliability behavior w.r.t to the control theoretical metrics are satisfactory. Given that the highest mean error is 18.90% in overshoot metric, followed by settling time with 11.76% and the others does not reach beyond 3%. The above average error is due to peaks in the reliability behavior during the executions resulting in values which trespassed the stability margin . Also, it seems that the parametric formula is less sensitive to components local reliability fluctuation in comparison with the estimated monitored behavior, see Figure 4.12 from instants [150s, 175s]. Where the orange curve follows the arc induced by the red, purple and green colored curves, and the dark blue just follows along a linear behavior.

Errors in the formula are propagated to the analysis method and its result. In this work, the verification is subject to the goal-oriented parametric formula devised in previous works [10, 9]. In these works, evidence on the steady-state values of the formula were produced and the correctness is once again reassured by the converging point analysis.

4.2.2 Providing Guarantees in the Presence of Uncertainty

It is desired that adaptation mechanisms guide the target system into constantly fulfilling its purpose with trustworthiness. Dynamic scenarios might lead into undesired behavior due to emergent uncertainties that affect the system functionality. As a result, guaranteeing that the adaptation mechanism is robust in the presence of uncertainty is

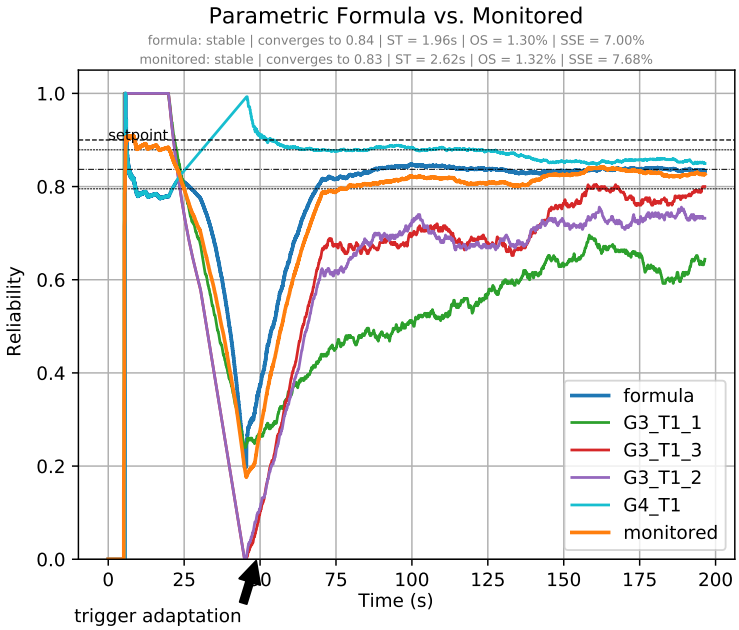


Figure 4.12: Exemplify reliability formula response and sensitivity.

key challenge on engineering assurance process. In this set of experiments we evaluate whether our approach supports the runtime analysis of the impact of emergent uncertainty through the BSN reliability response.

Experimental Setup

Following from the evaluation goal 1 the experiments of this section were run in the BSN. In addition, the designed goal-oriented adaptation engine is activated for reasoning on the composition of the entire system reliability. Since the adaptation engine reasons on a wider set of possible strategies, the executions were taken in periods of 600s. The uncertainty injection is focused on noise in sensing, which is exercised by different combinations of input signal types and its parameters. The adaptation mechanisms are incremented with a strategy to cope with message loss in the central hub. Finally, the response is analyzed w.r.t control theoretical metrics.

Scenario

We expect to evaluate whether the adaptation engine objective is guaranteed in the presence of uncertainty during the execution. Hence, the uncertainty injection mechanism is employed. We vary the types of injections and the amplitude of the signals. The step, ramp and random signals are used and the amplitude varies from 5% to 50%, covering a wide spectrum of the uncertainty injection.

In addition to the data collection task replication, a frequency adjustment strategy for message loss reduction in the central hub was employed for these experiments. As a result, the Figure 4.13 illustrates the local reliability evolution in time for each component. Where a downwards peak can be observed in the interval [50s, 100s]. The peak can be explained by the reliability recovery in the sensors, around instant t = 80s, that caused the sensors to send more messages to the central hub that, as consequence, lost messages due to a probable overflow and its reliability was degraded. Afterwards, the enactor updated the sensors frequency and from 100s on all system components tend to stabilize at 0.80.

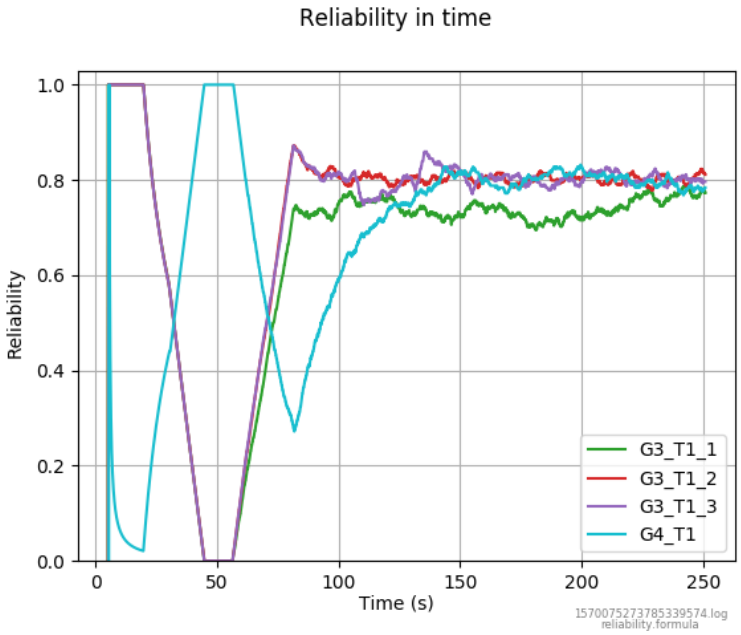


Figure 4.13: Local reliability behavior example

The adaptation engine have been setup to propagate a combination of what should be the reliability values for each component that would lead the system overall reliability to reach the desired value. In other words, the propagated strategy is a set of pairs containing the component's identification associated with a local desired reliability, the local setpoint. The property to be guaranteed in the experiment is the overall reliability, which was configured to be 95% with acceptable stability margin of ±2%.

Results

We execute the system for 24 different configurations regarding the uncertainty injection signal and the adaptation engine parameters and compute the control theoretical metrics for every generate system response, see Table 4.4. In which 7 out of the 24 were classified by our algorithm as not reaching the stability, with special remarks (*) for 5 that achieved the desired convergence point against 2 that diverged from the desired overall

reliability. A summary of the average value and error for each metric is presented in Table 4.3.

	convergence point	**settling time (s)**	**overshoot (%)**	**steady-state error (%)**
response	0.94 ± 0.04	287.70 ± 217.59	5.34% ± 0.04	1.75% ± 0.04

Table 4.3: Summary of system's response to noise in sensing injection

	input		adaptation engine parameters					system response				
# run	**type**	**amplitude (%)**	**setpoint**	**frequency (Hz)**	**quantity**	**offset (%)**	**granularity**	**stability**	**convergence point**	**settling time (s)**	**overshoot (%)**	**steady-state error (%)**
1	step	30.00%	0.95	0.02	500	50.00%	0.01	stable	0.93	426.71	3.51%	1.80%
2							0.1	stable	0.94	273.23	1.56%	1.48%
3						20.00%	0.01	stable	0.95	201.33	2.94%	0.18%
4							0.1	stable	0.95	211.45	2.82%	0.25%
5	step	2.00%	0.95	0.02	500	50.00%	0.01	not stable*	0.95	0.00	4.40%	0.02%
6							0.1	stable	0.96	528.44	4.20%	0.62%
7						20.00%	0.01	not stable*	0.94	0.00	6.04%	1.13%
8							0.1	stable	0.95	258.00	4.05%	0.05%
9	ramp	5.00%	0.95	0.02	500	50.00%	0.01	stable	0.95	414.02	5.75%	0.46%
10							0.1	stable	0.95	539.55	4.87%	0.38%
11						20.00%	0.01	stable	0.95	399.97	4.37%	0.45%
12							0.1	stable	0.95	489.82	3.91%	0.50%
13	ramp	50.00%	0.95	0.02	500	50.00%	0.01	not stable	0.80	0.00	21.55%	16.14%
14							0.1	not stable	0.82	0.00	15.92%	13.41%
15						20.00%	0.01	stable	0.94	215.16	4.17%	0.88%
16							0.1	stable	0.96	332.18	2.59%	0.60%
17	random	15.00%	0.95	0.02	500	50.00%	0.01	stable	0.95	505.46	4.94%	0.26%
18							0.1	not stable*	0.96	0.00	4.00%	0.94%
19						20.00%	0.01	not stable*	0.96	0.00	4.14%	0.91%
20							0.1	not stable*	0.96	0.00	4.48%	0.66%
21	random	40.00%	0.95	0.02	500	50.00%	0.01	stable	0.95	538.83	4.94%	0.52%
22							0.1	stable	0.95	536.30	4.39%	0.03%
23						20.00%	0.01	stable	0.95	505.24	3.96%	0.03%
24							0.1	stable	0.95	529.01	4.60%	0.31%

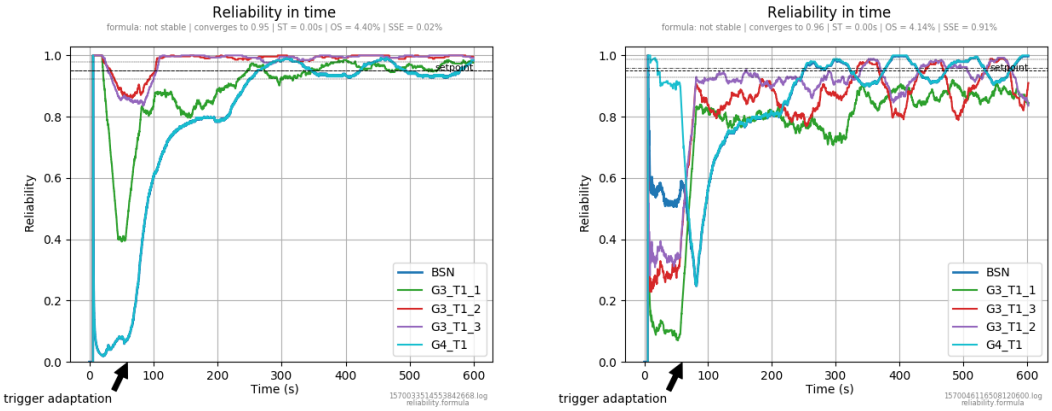
Table 4.4: System response to noise in sensing

Discussion

Fully supported by our approach, a runtime analysis of the adaptation engine operating in the presence of the noise in sensing source of uncertainty is performed as follows.

The goal-oriented adaptation engine has fulfilled the reliability at 95% objective at the majority of the exercised scenarios. Since we evaluated the response for different sets of parameter configurations for the adaptation engine, it is possible to compare and choose the configuration that best fits the stakeholders' purposes. From the obtained results for example, we can state that using a offset of 20% and a granularity of 0.1 would render a more robust adaptation mechanism for operating in scenarios subject to the inputs exercised. In addition, a more precise analysis on each adaptation engine configuration can be placed in respect to the other metrics, e.g. the configuration 20% offset and 0.01 granularity lead to smaller settling time for the injected signals. It is noteworthy as well that, on the highest noise in sensing uncertainty amplitude injected, ramp with 50% amplitude, the 50% offset diverged from the desired behavior in contrast to the 20% configuration.

As for the marked (*) stability labels a qualitative investigation of the response is necessary, see Figure 4.14. The not stable but convergent behavior is explained by our definition of stability, that relies on the stability margin requirement of $\pm 2\%$. The examples in Figures 4.14a and 4.14b illustrates that the overall reliability converges to the desired point but due to volatility in the behavior the last few points of the execution were out of the stability margin requirement. Depending on how strict the requirements are the stability margin could be stretched or this analysis could place doubts on the responses that stabilize on the last seconds of execution, pointed out by the settling time metric. If the former is the case, then we can say that the adaptation engine is robust enough to be stable at the 91.4% (22/24) of the cases or 41.6% (10/24) of the cases for the latter, since doubt would be placed in 7 other curves that stabilize after 500s.



(a) Step response with 2% amplitude to 50% offset and 0.01 granularity

(b) Random response with 15% amplitude to 20% offset and 0.01 granularity

Figure 4.14: Responses to not stable convergent scenarios

4.2.3 Threats to validity

Construct validity.

The major threats here are the correctness of the implementation of the BSN and of the proposed approach. The BSN has been thoroughly tested as part of previous work [32, 10]. Concerning our approach, at least two authors of this work reviewed the implementation and checked the plausibility of the evaluation results based on the experience they have with the BSN. Another threat is due to the formula generation process employed in this work, which follows the GODA-MDP implementation [10]. It does not guarantee that the formula terms are directly mapped into each component properties. Even though we proposed a mapping from means-end tasks to components, its enactment is subject to human-driven thus error-prone implementation. And even if the implementation of each component is correct, there lays a discussion on whether the components collaboration behaves accordingly to the source goal model. Which may lead to discrepancies between the desired behavior, the implemented and the modeled. We insistently advise for ensuring that the system engineer implements the component following the modeled means-task id and that the messages that run through the architectural components contains the source emitter, for correctly tracking the means-end implementation with its elements in the parametric formula.

Internal validity.

Our approach showed itself effective and efficient in the evaluation. Although we comprehensively deal with the contextual uncertainties class, unveiling sources of uncertainty involved in a system's operation is inherently non-deterministic, which represents a threat

to any assurance process. Moreover, our policy strategies were specified in ad-hoc manner, but they were implemented to automate the policies enactment. This might represent a scalability threat in complex scenarios with multiple uncertainties combined as well as in catastrophic scenarios where highly rare events have to be taken into account. Also, as observed at the evaluation goal 1 results from Section 4.2.1, the transient-state overshoot and settling time control theoretical metrics resulted in higher errors when compared to the estimated behavior. This is due to discrepancies in the parametric formula sensitivity when compared to the estimated behavior. Therefore, a deeper investigation must be placed on both estimated behavior and parametric formula transient-state response.

External validity.

Although our approach is platform independent, we do reckon the limitation of the evaluation since it was applied in the specific case of the BSN. Further evaluation must be performed to generalize the results. Despite all efforts to implement the BSN with new features to tame uncertainty, further study must be done to verify the applicability in real-world scenarios with multiple uncertainties combined.

Chapter 5

Related Work

5.1 Model-based Adaptation

Cámara J. et al [33] employ Stitch [34] specification language alongside with PRISM markov decision process for generating verifiable adaptation policies, in design-time, which compose a strategies repertoire. Additionally, Cámara J. et al [35] use model checking of stochastic multiplayer games for online reasoning for the adaptation plans enactment in a runtime-fashion. In contrast, our proposal relies on design-time models for guaranteeing desired behavior and enable model updates and reasoning for runtime adaptation.

Moreover, Calinescu R. et al [6] advocate the use of a goal driven method for assurance case specification which is updated during the adaptation. However it does not fit into using goal-models for guiding the system self-adaptation, since it is a product of the correct execution of the system, and not what drives the system to behave correctly during runtime. The approach uses UPPAAL [36] at design-time and PRISM [37] at runtime for modeling and verification. On the other hand, our proposal applies goal-models as specification languages that are analyzed in both design-time and runtime, the latter in the form of a parametric formula.

Shevtsov S. et al [38] rely on Setpoint, Threshold and Optimization requirements (STO-reqs) for representing user related adaptation policies in mathematical notation, what furthers the gap between non-specialist stakeholders' and the system in comparison with goal-modeling. Which is the basis of our work, despite that, we rely on goal-oriented notation for requirement specification.

Vogel T. et al [19] rely on an extensible *CompArch* language to express the runtime model that seamlessly interfaces with model-driven adaptation engines. The language designed for architectural adaptation ensembles mechanisms for issue injection, execution time measurement, impact analysis and adaptation strategy enactment. However, it does not give support for goal-oriented approaches without effort.

The present work employs the specification methods from Solano G. et al [10] and makes use of the goal-model to parametric formula. Such approach provides guarantee of correctness over goal-models at design-time and a formula that can be used for goal-oriented adaptation at runtime for reliability and cost evaluation. Ours is complementary to theirs, aiming at the provenience of guarantees of correctness for the runtime behavior w.r.t. control theoretical metrics.

5.2 Guarantees under Uncertainty

In Cámara J. et al [33] the system properties are discretized and modeled in PRISM with markov decision processes for optimal policies synthesis over state space representations, where the guarantees under uncertainty are provided by the probabilistic model-checking. However, they are limited to uncertainty modeled in design-time which also depends on the discretization factor. On the other hand, Cámara J. et al [35] employ model checking of stochastic games wherein the system and the environment are modeled as interacting players along with the desire to maximize an utility reward. Falling in the same limitations of uncertainty modeling capabilities.

Calinescu R. et al [6] employ either design-time and runtime (ActivFORMS [39]) techniques for provisioning assurance against uncertainty. The seven-step methodology comprises design-time modeling and verification in UPPAAL the relevant aspects of the controlled software system, environment and controller. Then, at runtime, performs verification with PRISM model checker. It is noteworthy that the models verified in design time are taken to runtime with a model-to-architecture transformation, placed by ActivFORMS. Thus, the unknowns left to runtime can be resolved through monitoring the system and the environment. Our work follows the same direction by suggesting a goal model to runtime architecture mapping.

Shevtsov S. et al [38] reportedly tackle uncertainty from four sources, (i) disturbances from the execution environment, (ii) system parameters, (iii) component interaction and (iv) requirement change. The approach relies on learning the software model and semi-automatically synthesizing the controller with manual poles choice that, at runtime, will cope with the aforementioned uncertainty. The manual support for controller synthesis settles the uncertainty management at design time, which is not the case of our proposal since there is room for model update at runtime, and, therefore, dynamic uncertainty mitigation at runtime.

Vogel T. et al [19] provide support for issues injection and an utility function usage for deriving the behavior of a system property through simulation in the form of a timeseries. Thus, uncertainty can be analytically overcome by stressing distinct uncertainty scenarios

which demands adaptation engine improvements during design-time that are taken to runtime in the same designed language. Such approach has inspired ours which does the same despite the presence of another layer of goal model verification at both design-time and runtime.

5.3 Control-based Metrics Analysis

In Cámara J. et al [33] an evaluation of the impact of the synthesized adaptation policies was taken. The method takes into consideration the solution's (1) optimality, by evidencing the percentage of constraints satisfaction, (2) robustness, by evaluating the repertoire in several distinct scenarios and (3) stability, since not converging scenarios are discarded. The evaluation method is manual, along with the system analysis. Accordingly, Cámara J. et al [35] assess (1) stability, (2) cost of control (by measuring the execution time to generate a plan) as well as (3) robustness (by simulating runtime disturbances). Thus, it partially fulfills the control-based metrics at design-time and at runtime.

Calinescu R. et al [6] reportedly issue that the control theoretical paradigm is not supported. Despite that, the evaluation step assesses (1) stability and (2) controller robustness by simulating randomly selected scenarios of execution. Their engine analysis process is manual and liable only to design-time.

On the other hand, Shevtsov S. et al [38] claims that control-based adaptation engines can be used for STO-reqs. Employing SimCA [40] the control theoretical adaptations provide formal guarantees for controller properties, such as (1) stability, (2) absence of overshoot, (3) zero steady-state error, (4) tuneable settling time and (5) tuneable robustness at runtime. The guarantees are addressed at both design- and runtime since not only the design-time synthesis provides guarantees, but tunnings at runtimes contributes to it as well.

Vogel T. et al [19] assess the adaptation engine effectiveness in terms of the utility function provided, and efficiency, in terms of the adaptation execution time. Their work supports the analysis of control-based metrics, but does not implements a verification mechanism that checks whether the control-based metrics satisfy the requirements.

Finally, Solano G. et al [10] manually analyze the effectiveness in terms of stability and robustness of the adaptation policies through simulations where the tackled uncertainty are stressed. Our proposal, in contrast, advocates the use of dynamic goal-models at runtime, enabling assessment of both reliability and performance metrics at design-time and at runtime.

We summarize the major work related to ours in Table 5.1. First, we evaluate which languages are used for modeling the system and its inherent uncertainty and whether

they are used at design-time or runtime. Then, we split the techniques for guaranteeing the desired behavior under uncertainty into model-checking and algebraic. Finally, we categorize the metrics used for evaluating the adaptation into control-based metrics in respect to stability, settling time, overshoot and steady-state error. The tag 'partial' indicates that the work evaluates at least one metric but not all. The lifecycle phase (i.e. design-time and runtime) is put in perspective as a third axis in the comparison provided. The phase in which each characteristic is resolved determines the assurance process of the solution.

	Model-base Adaptation	**Guarantees under Uncertainty**		**Control-based Metrics Analysis**
	Language(s)	**Model-Checking**	**Algebraic**	**Stability, OS, ST SSE**
Cámara J. et al. [33]	design-time	design-time	no	design-time (partial)
Cámara J. et al. [35]	runtime	runtime	no	runtime (partial)
Calinescu et al. [6]	design-time runtime	design-time runtime	no	runtime (partial)
Shevtsov et al. [38]	design-time	no	design-time	design-time
Vogel T. [19]	runtime	no	runtime	runtime (partial)
Solano G. [10]	design-time runtime	design-time	runtime	design-time runtime (partial)
Our Work	**design-time runtime**	**design-time**	**runtime**	**design-time runtime**

Table 5.1: Related work summary table

Chapter 6

Conclusions and Future Work

Uncertainty pervades all phases of software lifecycle, from requirements elicitation in design-time through runtime behavior. Several methods were developed within the software engineering for minimizing uncertainty at design-time and their impact in the system behavior. Lately, approaches advocate for runtime adaptation for reducing the costs of taming emergent uncertainty during the execution. In this work, we contribute to the claim that uncertainty must be consistently tamed on the lifecycle as a whole. We present an architecture that supports the verification of goal-oriented adaptation engines w.r.t widely adopted control theoretical metrics. Moreover, the verified system can seamlessly be taken to runtime with minimal or no changes bundled with evidence that the it behaves accordingly to its goals under the influence of certain kinds of uncertainty.

Our approach relies on a layered architecture implemented upon the Robot Operating System (ROS) middleware, which is widely adopted by academy and industry, for supporting reconfiguration and behavioral adaptation of software systems. Based on philosophy that goal models should pervade all the software lifecycle for the provision of trustworthy behavior, we focus on the analysis of adaptation engines based on goal-oriented reasoning processes. Which composes a step of the verification process of the system behavior under the influence of uncertainty w.r.t to control theoretical metrics.

The architecture is instantiated by an implementation of the Body Sensor Network (BSN) case study, in which reliability requirements are of extreme importance given the medical nature of the system. Mechanisms for adapting the BSN parameters which lead to reliability improvement, the system instrumentation for collecting data related to component failures and an adaptation engine that uses a parametric formula, derived from a goal model, for reasoning over the adaptations are designed and implemented accordingly to the proposed conceptual architecture. Finally, scenarios prone to uncertainty are designed and with the uncertainty injection component, also implemented on ROS, executions are performed for collecting evidence that the system behaves as expected in

a control theoretical analysis fashion.

Furthermore, the hypotheses that define the work solution are validated through empirical observation. That was taken in two phases, a correlation comparison between the parametric formula and the expected and monitored system behavior and a verification process on whether the implemented adaptation engine behaves as expected in face of uncertainty. The satisfactory results on both phases pave the way for further investigation on solutions that unite accessible GORE software development processes to control theoretical systematic methods. Given this work outcomes we elicit envisioned next steps which shall contribute to the ongoing research topic:

- **Explore the injection of other sources uncertainty**: Hezavehi et al. [5] classified uncertainty studied in software communities into 7 groups. We explored one in a specific scenario, noise in sensing. The architecture is ready for exploring other groups with not much effort, a straight-forward step would be to extend the components into permitting other uncertainty injections for providing more evidence and robustness for developed adaptation engines.

- **Sensitivity analysis of parametric formula**: The experiments from the evaluation goal 1 drew our attention to the fact that the parametric formula overall reliability is more sensible to components when decomposed by AND, which could lead us to develop heuristics for optimizing the adaption engine reasoning process. A sensitivity analysis on the on the parametric formula can help sharpening its accuracy and the adaptation engine itself, we then would suggest efforts on this duty.

- **Improvement on adaptation engine reasoning process**: The adaptation engine reasoning process presented in this work is specific for the situation it was developed to, as it stands upon the shoulders of strong assumptions on how the system behaves. Methods for developing more robust and independent from the system goal-oriented adaptation engines would be a follow up of this work, which could even use the presented architecture and case study for test-bed experiments.

- **Adaptation strategies characterization**: The adaptation strategies employed in the work were hardcoded in lower level language, that demanded specific implementation of strategy enactors, effectors and mechanisms inside the target system components for supporting the adaptation. The used of proper specific languages would empower the architecture and the application of the method. Thus it is a possible point of improvement for the work.

- **Control theoretical properties characterization**: The control theoretical properties verification against the system behavior is non-automated as in a model-checking approach, due specially to the fact that the properties to be verified followed no specific language. Specifying the properties in a logical language and a mechanism for automatic verification could be a follow up to the current work.

- **Evaluate the approach with other systems**: Even though the evaluation contributed with evidence on the applicability our approach, the evaluation was performed on a specific case study at specific scenarios, which would hinder threats to the validity as aforementioned. A straight-forward follow up would be to extend the evaluation to other SAS exemplars.

- **Explore other control theoretical metrics**: In this work we define, calculate and analyze the system response in respect to convergence point/stability, overshoot, settling time and steady-state error. However, there are other relevant variables when it comes to analyzing the a control theoretical solution: controller effort, controller robustness, delay time, rise time, peak time [22]. A differentiation from maximum overshoot to minimum undershoot could hinder more precision on the transient-state analysis and on the adaption engine.

References

[1] Camara, J., D. Garlan, and G. Eakman: *Building long-lived adaptive systems.* IEEE Software, 36(2):70–72, March 2019. 1

[2] Kephart, J. O. and D. M. Chess: *The vision of autonomic computing.* Computer, 36(1):41–50, Jan 2003. 1, 5

[3] Cheng, Betty H., Rogério Lemos, Holger Giese, Jeff Inverardi, and et al.: *Software engineering for self-adaptive systems.* chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009. 1, 5

[4] Lemos, Rogério de, Holger Giese, Hausi A. Müller, Mary Shaw, and et. al: *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 1, 2, 5

[5] Mahdavi-Hezavehi, S., P. Avgeriou, and D. Weyns: *Chapter 3 - a classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements.* In Mistrik, Ivan, Nour Ali, Rick Kazman, John Grundy, and Bradley Schmerl (editors): *Managing Trade-Offs in Adaptable Software Architectures*, pages 45 – 77. Morgan Kaufmann, Boston, 2017. 1, 19, 56

[6] Calinescu, Radu, Danny Weyns, Simos Gerasimou, M. Usman Iftikhar, Ibrahim Habli, and Tim Kelly: *Entrust: Engineering trustworthy self-adaptive software with dynamic assurance cases.* In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 495–495, New York, NY, USA, 2018. ACM. 1, 2, 51, 52, 53, 54

[7] Whittle, J., P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. Bruel: *Relax: Incorporating uncertainty into the specification of self-adaptive systems.* In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, Aug 2009. 1, 2, 23

[8] Filieri, Antonio, Martina Maggio, Konstantinos Angelopoulos, and et. al: *Software engineering meets control theory.* In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 71–82, Piscataway, NJ, USA, 2015. IEEE Press. 1, 2

[9] Mendonça, Danilo Filgueira, Genaína Nunes Rodrigues, Raian Ali, Vander Alves, and Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis.* Information and Software Technology, 80:245 – 264, 2016. 2, 11, 13, 14, 43

[10] Solano, Gabriela Félix, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione: *Taming uncertainty in the assurance process of self-adaptive systems: a goal-oriented approach.* 2019. 2, 11, 14, 15, 18, 23, 30, 43, 49, 52, 53, 54

[11] Ali, Raian, Fabiano Dalpiaz, and Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis.* Requir. Eng., 15(4):439–458, November 2010. 2, 12

[12] Van Lamsweerde, Axel: *Goal-oriented requirements engineering: A guided tour.* In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 249–, Washington, DC, USA, 2001. IEEE Computer Society. 2, 17

[13] Weyns, Danny: *Engineering self-adaptive software systems - an organized tour.* In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018*, pages 1–2, 2018. 2, 19

[14] Lemos, Rogério de, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, Yuriy Brun, Javier Cámara, Radu Calinescu, Myra B. Cohen, Alessandra Gorla, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jézéquel, Sam Malek, Raffaela Mirandola, Marco Mori, Hausi A. Müller, Romain Rouvoy, Cecília M. F. Rubira, Éric Rutten, Mary Shaw, Giordano Tamburrelli, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, and Franco Zambonelli: *Software engineering for self-adaptive systems: Research challenges in the provision of assurances.* In *Software Engineering for Self-Adaptive Systems*, volume 9640 of *Lecture Notes in Computer Science*, pages 3–30. Springer, 2013. 2

[15] Weyns, Danny, Nelly Bencomo, Radu Calinescu, Javier Cámara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean-Marc Jézéquel, Sam Malek, Raffaela Mirandola, Marco Mori, and Giordano Tamburrelli: *Perpetual assurances for self-adaptive systems.* In *Software Engineering for Self-Adaptive Systems*, volume 9640 of *Lecture Notes in Computer Science*, pages 31–63. Springer, 2013. 2

[16] Weyns, Danny, Nelly Bencomo, Radu Calinescu, Javier Camara, Carlo Ghezzi, Vincenzo Grassi, Lars Grunske, Paola Inverardi, Jean Marc Jezequel, Sam Malek, *et al.*: *Perpetual assurances for self-adaptive systems.* In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 31–63. Springer, 2017. 2

[17] Filieri, Antonio, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel: *Control strategies for self-adaptive software systems.* ACM Trans. Auton. Adapt. Syst., 11(4):24:1–24:31, February 2017. 2, 3

[18] Shevtsov, S., M. Berekmeri, D. Weyns, and M. Maggio: *Control-theoretical software adaptation: A systematic literature review.* IEEE Transactions on Software Engineering, 44(8):784–810, Aug 2018. 2, 8, 9

[19] Vogel, Thomas: *mrubis: An exemplar for model-based architectural self-healing and self-optimization.* In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 101–107, New York, NY, USA, 2018. ACM. 3, 18, 51, 52, 53, 54

[20] Braberman, Victor, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel: *Morph: A reference architecture for configuration and behaviour self-adaptation.* In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, CTSE 2015, pages 9–16, New York, NY, USA, 2015. ACM. 5, 6

[21] Avizienis, A., J. . Laprie, B. Randell, and C. Landwehr: *Basic concepts and taxonomy of dependable and secure computing.* IEEE Transactions on Dependable and Secure Computing, 1(1):11–33, Jan 2004. 6

[22] Ogata, Katsuhiko: *Modern Control Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001. 7, 8, 57

[23] Horkoff, Jennifer, Fatma Başak Aydemir, Evellin Cardoso, Tong Li, Alejandro Maté, Elda Paja, Mattia Salnitri, Luca Piras, John Mylopoulos, and Paolo" Giorgini: *Goal-oriented requirements engineering: an extended systematic mapping study.* Requirements Engineering, 24(2):133–160, Jun 2019. 10

[24] Horkoff, Jennifer, Tong Li, Feng Lin Li, Mattia Salnitri, Evellin Cardoso, Paolo Giorgini, and John Mylopoulos: *Using goal models downstream: A systematic roadmap and literature review.* International Journal of Information System Modeling and Design, 6(2):1–42, 2015. 10, 11

[25] Hahn, Ernst Moritz, Holger Hermanns, Bjorn Wachter, and Lijun Zhang: *Param: A model checker for parametric markov models.* In Touili, Tayssir, Byron Cook, and Paul Jackson (editors): *Computer Aided Verification*, pages 660–664, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 14

[26] Baresi, Luciano and Carlo Ghezzi: *The disappearing boundary between development-time and run-time.* In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 17–22, New York, NY, USA, 2010. ACM. 23

[27] Cheng, Betty H. C., Pete Sawyer, Nelly Bencomo, and Jon Whittle: *A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty.* In Schürr, Andy and Bran Selic (editors): *Model Driven Engineering Languages and Systems*, pages 468–483, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 23

[28] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002, ISBN 0201745720. 26

[29] Pessoa, Leonardo, Paula Fernandes, Thiago Castro, Vander Alves, Genaína N. Rodrigues, and Hervaldo Carvalho: *Building reliable and maintainable dynamic software product lines: An investigation in the body sensor network domain*. Information and Software Technology, 86:54 – 70, 2017, ISSN 0950-5849. 30

[30] Caldas, Ricardo: *Prototipação e verificação formal de sistema autônomo com propriedades tempo-real : um estudo de caso no body sensor network*. Universidade de Brasilia, 2017. Available at `http://bdm.unb.br/handle/10483/19223`. 30

[31] Caldiera, Victor R Basili1 Gianluigi and H Dieter Rombach: *The goal question metric approach*. Encyclopedia of software engineering, pages 528–532, 1994. 39

[32] Rodrigues, Arthur, Ricardo Diniz Caldas, Genaína Nunes Rodrigues, Thomas Vogel, and Patrizio Pelliccione: *A learning approach to enhance assurances for real-time self-adaptive systems*. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 206–216, New York, NY, USA, 2018. ACM. 49

[33] Cámara, Javier, Bradley Schmerl, Gabriel A. Moreno, and David Garlan: *Mosaico: Offline synthesis of adaptation strategy repertoires with flexible trade-offs*. Automated Software Engg., 25(3):595–626, September 2018. 51, 52, 53, 54

[34] Cheng, Shang-Wen and David Garlan: *Stitch: A language for architecture-based self-adaptation*. Journal of Systems and Software, 85(12):2860–2875, 2012. 51

[35] Cámara, Javier, David Garlan, Bradley Schmerl, and Ashutosh Pandey: *Optimal planning for architecture-based self-adaptation via model checking of stochastic games*. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 428–435, New York, NY, USA, 2015. ACM. 51, 52, 53, 54

[36] Bengtsson, Johan, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi: Uppaal — *a Tool Suite for Automatic Verification of Real–Time Systems*. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in *Lecture Notes in Computer Science*, pages 232–243. Springer–Verlag, October 1995. 51

[37] Kwiatkowska, Marta, Gethin Norman, and David Parker: *Prism: Probabilistic symbolic model checker*. In Field, Tony, Peter G. Harrison, Jeremy Bradley, and Uli Harder (editors): *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 200–204, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. 51

[38] Shevtsov, Stepan, Danny Weyns, and Martina Maggio: *Simca*: A control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees*. ACM Trans. Auton. Adapt. Syst., 13(4):17:1–17:34, July 2019. 51, 52, 53, 54

[39] Iftikhar, M. Usman and Danny Weyns: *Activforms: Active formal models for self-adaptation*. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 125–134, New York, NY, USA, 2014. ACM. 52

[40] Shevtsov, Stepan and Danny Weyns: *Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems.* In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 229–241, New York, NY, USA, 2016. ACM. 53