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

A Goal-Oriented Approach to Support the Assurance Process of Self-Adaptive Systems under Uncertainty

Gabriela Félix Solano

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2019

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

A Goal-Oriented Approach to Support the Assurance Process of Self-Adaptive Systems under Uncertainty

Gabriela Félix Solano

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof.a Dr.a Genaína Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof.a Dr.a Bozena Wozna
Jan Dlugosz University

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 11 de julho de 2019

Acknowledgements

I would like to thank my supervisor, Prof.a Dr.a Genaína Nunes Rodrigues, for the patience, wisdom and guidance throughout this work. I would also like to thank our research group for all the collaboration and valuable insights. Finally, I would like to thank my family for the never-ending support, and CAPES for the financial aid received.

Abstract

Goals are first-class entities in a self-adaptive system (SAS) as they guide the self-adaptation. A SAS often operates in dynamic and partially unknown environments, which cause uncertainty that the SAS has to address to achieve its goals. Moreover, besides the environment, other classes of uncertainty have been identified. However, these various classes and their sources are not systematically addressed by current approaches throughout the life cycle of the SAS. Recognizing the different classes and sources of uncertainty contributes to a clear understanding of how they impact system goals and behavior, therefore assisting the assurance process of SAS. We propose a goal-oriented approach that models SAS within uncertainty and generates verifiable models for system verification at both design- and runtime. At design time, based on a goal model augmented with uncertainty annotations, we automatically generate: (i) a Markov Decision Process (MDP) model in PRISM language, and (ii) reliability and cost parametric formulae with parameterized uncertainties. The MDP is used by probabilistic model checking activity to support system analysis and verification at design time. The parametric formulae provide means for efficient runtime analysis of SAS and guide the synthesis of adaptation policies by engineers. In this work, we focus on reliability and cost properties, for which we evaluate our approach on the Tele Assistance System (TAS) and the Body Sensor Network (BSN) system. The results of the validation are promising and show that our approach is able to generate scalable and trustworthy verifiable models of SAS under uncertainty.

Keywords: Self-adaptive systems, uncertainty, goal modeling, symbolic model checking, Markov decision process, non-determinism

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern software systems constantly operate under changing conditions due to new requirements and dynamic system context. Nonetheless, these systems are expected to maintain their functional and nonfunctional requirements without interruption. In this sense, a self-adaptive system (SAS) is designed to modify its behaviour at runtime in order to keep satisfying certain objectives. Engineering SAS shows great difficulty as the knowledge available at design time is often missing or it is inaccurate to predict all operating conditions that the system will encounter at runtime. Such challenge derives from the key fact that self-adaptation is subject to uncertainty [4].

Uncertainty in a software system is defined as the circumstances in which the system's behavior deviates from expectations due to dynamicity and unpredictability of a variety of factors existing in such systems [5]. For this reason, uncertainty is a fundamental challenge for a SAS and pervades from the system's requirements to its infrastructure and runtime environment [4]. Many sources of uncertainty have been identified and further grouped to four classes: (i) the system itself, (ii) the system goals, (iii) the environment, and (iv) human aspects [5, 1]. These various classes of uncertainty have to be addressed by SAS assurance processes to comprehensively and systematically address such uncertainties. Otherwise, the provided assurances may render themselves inaccurate or incomplete. Therefore, uncertainty should be leveraged as a first-class concept in self-adaptation [4] and in the assurance process [1]. We generally consider an assurance process as all design-time and runtime activities providing evidence that the adaptation goals are satisfied.

1.1 Problem Definition

A SAS faces a paradoxical challenge in the presence of uncertainties [1]: "*how can one provide guarantees for the goals of a system that is exposed to continuous uncertainties?*". There has been extensive research to address uncertainty in SAS [5, 6], but with no focus

on proposing solutions to systematically tackle classes of uncertainty and its sources [5]. Also, the main focus has been on environmental uncertainty (regarding the variability of execution contexts), and on system goals uncertainty (regarding goal changes) [5]. Moreover, most of the existing approaches postpone the treatment of uncertainty to the runtime phase of the system's life cycle [5] while rather neglecting the explicit management of uncertainty right from the start with the requirements.

Because of uncertainty, design-time knowledge of SAS operating conditions might be missing or inaccurate. Therefore, the assurance process of SAS at this stage may render itself untrustworthy. At runtime, knowledge is available but there are limitations over time and space resources. In this sense, an assurance process that traverses throughout the whole system's life cycle needs to consider the inherent characteristics of each stage and guarantee the system's goals accordingly. To the best of our knowledge, there is no end-to-end goal-oriented approach that supports SAS assurance process while considering different classes and sources of uncertainty. Considering, we summarize the objective of this work in the following research question.

RQ: Is it possible to support the assurance process of SAS at both design- and runtime, while considering different classes and sources of uncertainty, using a goal-oriented approach?

1.2 Proposed Solution

To recognize and manage uncertainties in the assurance process from early on, we propose a goal-oriented approach based on Goal-Oriented Requirements Engineering (GORE) [7]. GORE 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 [8, 7, 9, 10]. Based on GORE, our approach leverages goal modeling to support the modeling of different sources of uncertainty within the following classes: (1) system itself, (2) system goals, and (3) environment. We should note that by system we mean the managed system that comprises application code to realize domain functionality [1]. The approach further supports the automatic generation of trustworthy verifiable models *parameterized with uncertainties*, which are used at design time and runtime to assist the assurance of a SAS.

We augment goal modeling with uncertainty annotation and verification support, in which the leaf tasks represent executable components in the system and have their respective reliability and cost properties. Then, we automatically translate the resulting

goal model into a Markov Decision Process (MDP), further generated as reliability and cost parametric formulae using our symbolic model checking compositional process. The MDP is a verifiable model used by probabilistic model checking techniques to analyze properties of interest at design time. The formulae are used as runtime models to express probabilities over the fulfillment of SAS goals. Specifically, they take different classes of uncertainty into account while supporting self-adaptation. To overcome the state-space explosion problem of traditional model checking [11], we compositionally generate such parametric symbolic formulae from MDPs with parameterized uncertainties based on our augmented goal model.

1.3 Evaluation

We evaluate our work by the scalability of our verification process and by the trustworthiness of our parametric formulae to efficiently represent SAS reliability and cost properties. For this purpose, we use the Tele Assistance System (TAS) [12] and the Body Sensor Network (BSN) [13] system. For our first evaluation goal, we provide a scalability analysis of our approach with respect to the time and space complexity of generating and analyzing the MDP model and parametric formulae. The results show that our approach can be quite affordable for modeling and analyzing real SAS, but it requires further reasoning over modeling more scalable non-deterministic models in PRISM language. For our second evaluation goal, we build an augmented CGM for both TAS and BSN. Then, we experimentally simulate these systems, retrieving their leaf tasks' reliability and cost value, and mapping them into our parametric formulae. Finally, we compare the systems' overall reliability and cost given by the experiments with the results given by our formulae to assure the trustworthiness of our generated runtime models. Results show consistency between reliability evaluation of both systems, but, regarding cost evaluation, there were discrepancies between the evaluation methods (i.e., parametric formula in PARAM [14]). Using ePMC [15], however, showed itself a more accurate symbolic model checker for our experiments.

1.4 Contribution

In a nutshell, to support the assurance process of SAS under multiple classes of uncertainties, we propose an augmented goal-oriented approach that blends over the high-level representation of goals and low-level representation of synthesis and verification models of SAS through symbolic model checking with parameterized uncertainties. The contributions of this work are threefold:

1. We introduce annotations to semantically enhance goal models with different classes of uncertainty, thus *providing first class support for uncertainty* in goal modeling.
2. We develop an algorithm to automatically generate parametric formulae from our enhanced goal model. These formulae are compositional, parameterized with uncertainties groups and represent both the reliability and cost properties of the modeled system.
3. We provide a new framework by extending the goal-oriented dependability analysis framework (GODA) [3] with the generation of parametric formulae with parameterized uncertainties.

1.5 Organization

The remaining chapters of the work are structured as follows. In Chapter 2, we provide background knowledge on self-adaptive system, goal modeling, Markov decision process, model checking and GODA framework. In Chapter 3 we present the core of our proposal, and in Chapter 4 we evaluate the approach. Chapter 5 highlights related work. Finally, Chapter 6 concludes the work along with future work.

Chapter 2

Background

2.1 Self-Adaptive Systems and Uncertainty

Modern software systems must deal with uncertainty and changes in the environment. In this sense, a self-adaptive systems (SAS) should be able to modify its behavior in response to its perception of the environment, the system itself, and its goals [6]. SAS is determined by two complementary principles [1]: (i) external principle, and (2) internal principal. The former reflects the view of the software engineer of the system, while the latter focuses on how the system is conceived. According to Weyns [1], in the external principle, SAS is a system able to handle changes and uncertainties in its environment, in the system itself and in its goals autonomously, that is, with minimal or no human interference. In the internal principle, SAS comprises two parts: the first one interacts with the environment and is responsible for domain concerns; the second part interacts with the first part and is responsible for adaptation concerns. Figure 2.1 depicts a conceptual model of a SAS.

The conceptual model incorporates four elements: (i) environment, (ii) managed system, (iii) adaptation goals, and (iv) managing system. The *environment* refers to the part of the external world with which the SAS interacts with and in which the effects of the system will be observed and evaluated [16]. It can be sensed and effected but the SAS cannot manage the environment. The *managed system* is the application code that realises the system's domain functionality, while the *managing system* is the system responsible for comprising the adaptation logic that deals with one or more adaptation goals [1]. *Adaptation goals* are concerns of the managing system over the managed system. They usually relate to software qualities, such as self-configuration, self-optimisation, self-healing, and self-protection [17].

Uncertainty in a software system is defined as the circumstances in which the system's behavior deviates from expectations due to dynamicity and unpredictability of a variety of factors existing in such systems [5]. Uncertainty is an important concern that SAS

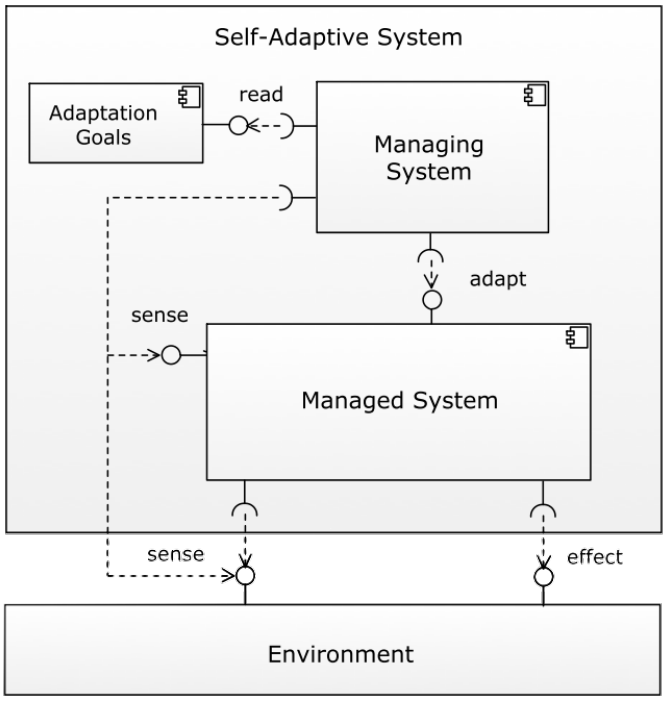


Figure 2.1: Conceptual model of a self-adaptive system [1].

needs to deal with. The execution context in which the system operates can change or become unpredictable, causing negative effects on the system, its goals, and its environment. Therefore, managing uncertainty can increase the level of assurance in a SAS [18]. According to Mahdavi-Hezavehi *et al.* [5], uncertainty is classified in four different classes: (i) uncertainty related to the system itself, (ii) uncertainty related to the system goals, (iii) uncertainty in the execution context, and (iv) uncertainty related to human aspects. Uncertainty related to the *system itself* refers to the managed system in SAS. Uncertainty related to the *system goals* refers to the specification, modeling and alteration of system goals. Uncertainty related to the *execution context* refers to the environment in which SAS operates and interacts with. Uncertainty related to *human aspects* refers to the unpredictability of human behavior interacting with the system.

2.2 Goal-Oriented Requirements Engineering

Goal-Oriented Requirements Engineering (GORE) is concerned with the use of goals for eliciting, structuring, specifying, analyzing, and, among others, modifying requirements [7]. Goals are entities that capture the objectives a system should achieve [7]. Since many failures in software systems stem from poor requirements elicitation [19], a

proper understanding of what the system is supposed to do is key for its trustworthiness. To this end, GORE [7] offers proved means to decompose technical and non-technical requirements into well-defined entities (goals) and reason about the alternatives to meet them [3]. Moreover, GORE has been used as means to model and reason about the systems' ability to adapt to changes in dynamic environments [7, 8, 9]. It has also been used as runtime model to respond to dynamic changes. Thus, goal modeling itself is a mean to mitigate uncertainties related to system goals, such as elicitation of requirements, specification of goals and goal changes.

There are different goal-oriented frameworks and methodologies, such as KAOS [19], i* [20] and TROPOS [21]. Despite some syntax differences, most goal-oriented approaches share a set of common and important concepts [8]:

- **Actor:** actors are humans or software with the ability to decide autonomously on how to achieve their goals.
- **Goal:** goals are abstractions to represent stakeholders' needs and expectations, while offering an intuitive way to elicit and analyze requirements.
- **Task:** tasks are responsible for the operationalization of actor's goals, that is, an operational means to reach them.
- **Resource:** a resource is an entity data or physical device that is generated or required by an actor. For example: a vital sign measurement, a sensor, the power from the battery component, etc.
- **AND-decomposition:** an AND-decomposition is a refinement link that decomposes goals/task into sub-goals/sub-tasks, meaning that all of the sub-goals/sub-tasks must be fulfilled/executed to satisfy its parent entity.
- **OR-decomposition:** an OR-decomposition is a refinement link that decomposes goals/task into sub-goals/sub-tasks, meaning that at least one of the sub-goals/sub-tasks must be fulfilled/executed to satisfy its parent entity.
- **Means-end:** a relation that indicates a means to fulfill a goal through the execution of a task.

2.2.1 Contextual Goal Model

Because SAS operates in dynamic environments, the requirement engineering processes of these systems must take into consideration not only the requirements and means to achieve them, but also the contextual information related to the system's operation. To this end, a Contextual Goal Model (CGM) is able to represent the requirements to meet,

the ways to meet requirements, and factors that can affect the quality and behavior of a system.

Besides the common elements of goal models described in Section 2.2, a CGM is also composed of *contexts*. According to Ali *et al.* [8], contexts are partial states of the world that are relevant to a goal fulfillment. Context changes may influence the goals, their quality and the means of achieving them. We should note the difference between resources and contexts. For instance, on the one hand, a resource is a sensor that measures the system's environmental conditions; on the other hand, specific behaviors of the sensor can be defined as contexts (e.g., the sensor availability or unavailability).

Figure 2.2 shows an excerpt of the CGM for the Body Sensor Network (BSN) [13]. The main objective of the BSN is to keep track of a patient's health status, continuously classifying it into *low*, *medium*, or *high risk*, and to send an emergency signal to a central unit in the case of an anomaly. The structure of the BSN is as follows: several wireless sensors are connected to a person to monitor her vital signs, namely, an electrocardiograph sensor (ECG) for heart rate beats measurement, a pulse oximeter for blood oxygen saturation (SaO2), a thermometer for body temperature (TEMP) in Celsius, and a sphygmomanometer for measuring diastolic and systolic arterial blood pressure (ABP). Additionally, the central node can preprocess the collected data, filter redundancy, translate communication protocols, and fuse data. Table 2.1 shows how the sensor values and thus, how the context relates to the patient's health risk as specified by a domain expert.

Table 2.1: Context Operationalization for a Patient's Health Status.

Sensor Info	**Data Ranges**
Oxygen saturation:	100 > low > 65 > medium > 55 > high > 0
Heart beats:	300 > high > 115 > medium > 97 > low > 85 > medium > 70 > high > 0
Temperature:	50 > high > 41 > medium > 38 > low > 36 > medium > 32 > high > 0
Systolic Pressure:	300 > high > 140 > medium > 120 > low > 0
Diastolic Pressure:	300 > high > 90 > medium > 80 > low > 0

In Figure 2.2, the root goal of the actor "Body Sensor Network" is "G1: Emergency is detected", which is refined into "G2: Patient status is monitored". G2 is divided into two sub-goals: "G3: Vital signs are monitored" and "G4: Vital signs are analyzed". Such goals are realized by tasks that are refined into leaf tasks that are operational. For instance, G3 is realized by "T1: Monitor vital signs" that is refined to "T1.1: Collect SaO2 data", "T1.2: Collect ECG data", "T1.3: Collect temperature data", "T1.4: Collect ABP data",

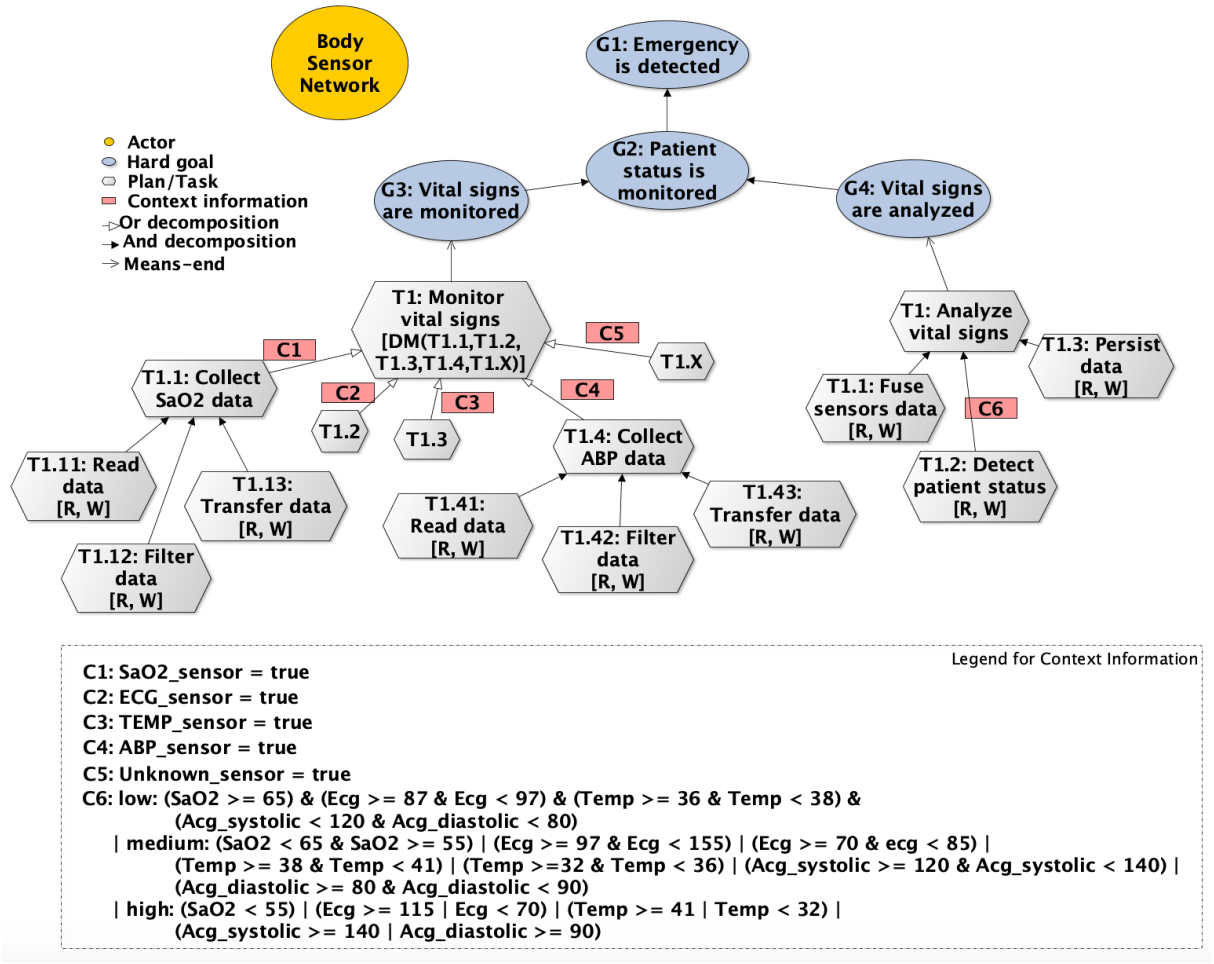


Figure 2.2: Contextual Goal Model of the Body Sensor Network.

and "T1.X". The tasks T1.2 and T1.3 are not detailed in Figure 2.2 but their further refinements to leaf tasks are analogous to task T1.1 and T1.4. The placeholder task T1.X exists because of uncertainty and will be discussed in the following chapter. The operation of the BSN is subject to six different context conditions. Contexts C1 to C5 refer to the availability of sensors that collect data related to a patient's vital signs, which may be *true* or *false*. Context C6 refers to the operationalization of collected data, stipulating ranges of valid integer or double values that each sensed data can assume when analyzing a patient's health status for risks [13].

Contexts C1 to C5 cause a non-deterministic behavior such that the fulfillment of G3 depends on the execution of any combination of tasks T1.1, T1.2, T1.3, T1.4, and T1.X. The behavior assumed at runtime is related to the different conjunction of contexts that currently hold, and each conjunction shapes the system to fulfill a goal at a different quality level.

2.3 Markov Decision Process

Markov Decision Process (MDP) models systems that exhibit both *probabilistic* and *non-deterministic* behavior [22]. In an MDP, non-deterministic choices model unknown aspects of a system's behavior. Each possible way of resolving such choices is referred to as an *strategy* (also called policy or adversary) [23]. Each strategy generates an induced Discrete-Time Markov Chain (DTMC) of the MDP [24]. To verify a property on a MDP, it is necessary to check whether it holds for all possible strategies of the MDP. Therefore, verifying quantitative properties of a MDP equates to evaluating the best- or worst-case behavior that can arise [23]. An MDP is formally defined as follows.

Definition 1 (**Markov Decision Process**). *A Markov Decision Process is a tuple* $M = (S, s_{init}, A, \delta_M, Lab, r)$ *where* S *is a finite set of states,* $s_{init} \in S$ *is an initial state,* A *is a finite set of actions,* $\delta_M : S \times A \to Dist(S)$ *is a (partial) probabilistic transition function,* $Lab : S \to 2^{AP}$ *is a labelling function assigning to each state a set of atomic propositions taken from a set* AP*, and* r *is a state reward function of the form* $r : S \to \mathbb{R}_{\geq 0}$*.* $Dist(X)$ *denotes the set of discrete probability transitions over a finite set of parameters* X*.*

Figure 2.3 shows a MDP modeling a system that aims to execute a task. The MDP states are $S = \{s_0, s_1, s_2, s_3\}$, where s_0 is the initial state. The actions are $Act = \{go, safe, risk, reset, finish, stop\}$. From each state, one or more actions can be taken. For instance, from state s_0, the action *go* should be taken, while from state s_1, actions $risk$ or $safe$ can be taken. The sum of transition probabilities for each action taken is always equal to 1. The reward assigned to each state and/or action is represented as an underlined number next to the labelling functions and/or the model's actions. The non-determinism is clear on state s_1. From this state, the system should make a decision of which action to take: $risk$, with reward of 4, or $safe$, with reward of 1. Deciding to take action $risk$, there is a 0.5 probability of transitioning to final success state s_2, and a 0.5 probability of transitioning to final fail state s_3. Deciding to take action $safe$, there is a 0.7 probability of transitioning back to initial state s_0, and a 0.3 probability of transitioning to final success state s_2.

Because of uncertainty, SAS engineers cannot always know how the system will behave during execution. In other words, depending on runtime context variability the system may behave differently, leading to a non-deterministic behavior. In this work, we use MDP to model SAS under uncertainty. Therefore, we are able to represent the system behavior while taking into account uncertainty it might face and the non-deterministic behavior that uncertainty brings.

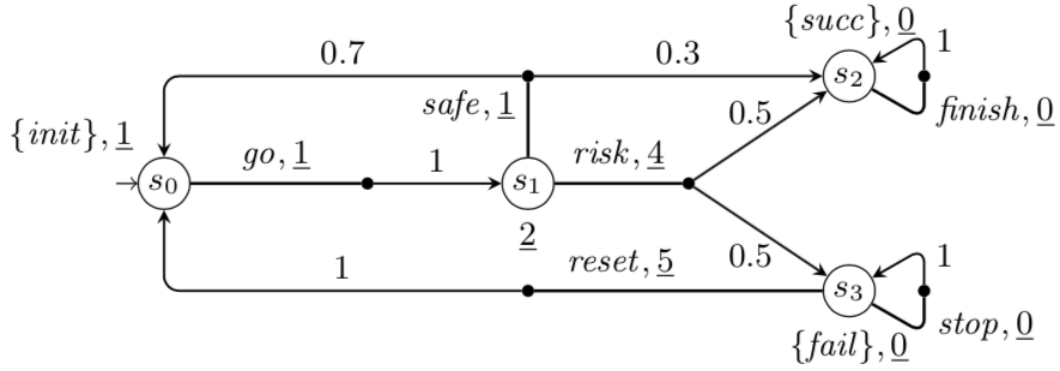


Figure 2.3: MDP Example [2].

2.4 Probabilistic Model Checking

Probabilistic model checking is an automated technique for verifying quantitative properties of stochastic systems [23]. It uses a systematic exploration and analysis of a system model to verify that certain requirements, specified in temporal logic, are satisfied by the model [23]. Alternatively to traditional model checking techniques that verifies the absolute correctness of a system [22], probabilistic model checking computes the probability with which the properties of interest are verified on the system [3]. In probabilistic model checking techniques, the system model is annotated with probabilities. Markov chains and MDPs are commonly used models. In this work we will use MDP for its ability to model both non-deterministic and probabilistic behaviour. Here, we use Probabilistic Computational Tree Logic (PCTL) [25] to specify our properties of interest and we verify them against the MDP that models the system.

Since our properties of interest are overall reliability and cost, we specify them based on the *probabilistic existence property* [26]. Therefore, the reliability is defined as *"the probability a system will eventually reach a state that satisfies its root goal"*; and the cost is defined as *"the system's cumulative cost when reaching a state that satisfies its root goal"*. Because we use MDP, we must specify the maximum and minimum PCTL properties, which are $Pmax =?[F(\phi)]$ and $Pmin =?[F(\phi)]$ for reliability; $Rmax =?[F(\phi)]$ and $Rmin =?[F(\phi)]$ for cost, where ϕ represents the satisficement of the root goal.

If, on the one hand, a key strength of probabilistic model checking is its ability to automatically analyze quantitative properties in an *exhaustive* manner, on the other hand this technique suffers from a state-explosion problem [23]. This means that the number of states needed to accurately model the system can easily exceed computer memory resources. Additionally, due to environmental variability, the modeled parameters need to be frequently evaluated over time. These problems show that probabilistic model

checking at runtime can be even more computationally expensive in terms of execution time and memory occupation. Considering the high environmental variability SAS faces and in view of the aforementioned problems, symbolic model checking [27] represents an alternative technique to improve runtime efficiency.

2.4.1 Symbolic Model Checking

Symbolic model checking is a parametric variant of probabilistic model checking. With this technique, the system model is constructed as a parametric MDP, where the parameters represent undetermined transition probabilities [27]. In this case, while conventional probabilistic model checking associates single values to transitions probabilities, symbolic model checking calls for transition probabilities specified as functions over a set of parameters [27]. Such a model checking technique enables a more flexible analysis, where verification produces parametric formulas and not single values [3], in which one can perform *parametric model checking*, i.e., check whether a formula holds for different values of the parameters [27]. In other words, by building the formulae at design time via symbolic model checking and resuming runtime verification to assigning values to parameters in formulae, the cost of model analysis is shifted from runtime to design time.

2.5 Goal-Oriented Dependability Analysis Framework

Goal-Oriented Dependability Analysis (GODA) is a framework proposed by Mendonça *et al.* [3] to model goals and analyse their fulfillment in different contexts. Figure 2.4 depicts GODA's process. First, a Goal Model is produced through conventional goal modeling. Then it is augmented with contextual and runtime information (in which context is defined as a formula of world predicates and the runtime information specifies behavioral detail about how goals are to be achieved), becoming a Contextual and Runtime Goal Model (CRGM). Once the CRGM of the system is complete, GODA automatically translates it into a DTMC. Dependability properties are then rendered as PCTL formulas, and the verification of the system can be carried out. The whole process is intrinsically iterative, and the different activities can be repeated multiple times before obtaining a complete and coherent specification of the system [3].

GODA takes into account runtime aspects and accommodates the implications that contextual information may have on goal satisfaction. GODA provides means to estimate the dependability of the strategies to fulfill goals in different contexts. At runtime, the outcome provided by GODA can be used to analyze whether the system is fulfilling its dependability goals. If it turns out that the obtained dependability is under a cer-

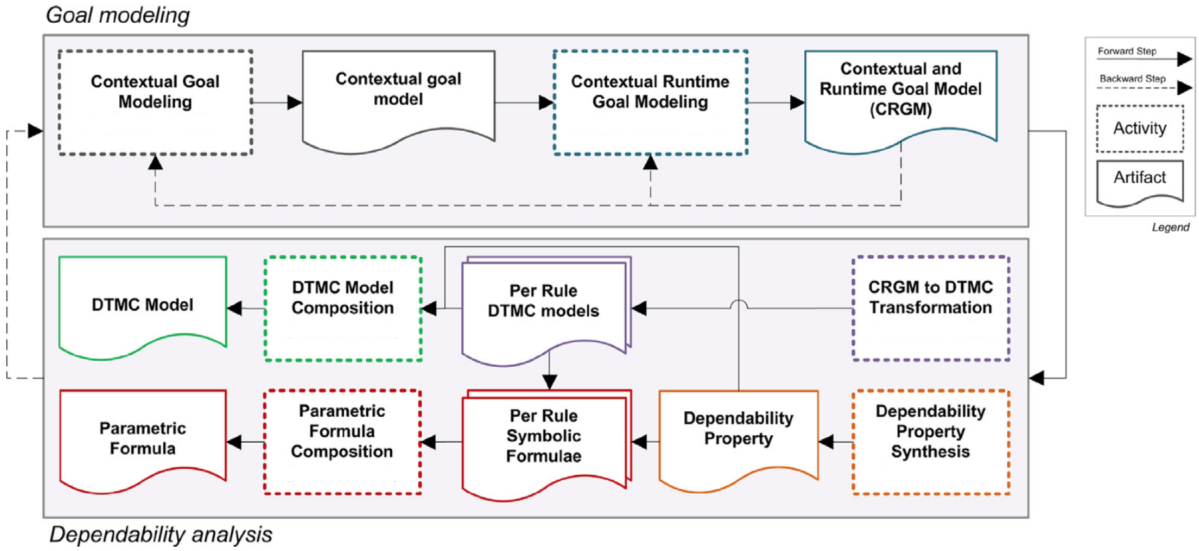


Figure 2.4: GODA Process [3].

tain threshold, the system should consider the strategy that provides the most suitable dependability measure.

The framework only supports the generation of DTMC as verification models. Despite the context information modeled, the system is deterministic, that is, once the parameters are evaluated, its behavior can be entirely determined and predicted. In SAS, however, one cannot assert such behavior. In fact, the uncertainty presented in SAS arises a non-determinism in the execution flow. Using GODA to model every possible system behaviour in a deterministic manner would require hard work from software engineers (in order to predict all runtime conditions) and would mostly not scale for real life systems. Our purpose with this work is to enhance GODA's capabilities focusing on: (i) modeling SAS under uncertainty, (ii) acknowledging the non-deterministic behaviour that comes with it, (iii) specifying not only reliability, but also cost properties, so we can provide means for a trade-off in decision-making at runtime, and (iv) providing means for guiding the synthesis of adaptation policies in SAS.

Chapter 3

A Goal-Oriented Approach to Support the Assurance of SAS under Uncertainty

Our approach aims at assisting the assurance process of SAS. It relies on the model representation of a SAS under uncertainty that allows a reasoned and precise analysis of reliability and cost properties, and that guides the synthesis of adaptation policies. Figure 3.1 provides an overview of our approach.

The approach contains two main design-time phases: (1) the goal modeling phase, and (2) the automatic translation phase[1]. In the goal modeling phase, the *"Contextual Goal Modeling with Uncertainty"* activity extends the GODA framework [3] to support goal modeling that takes different classes of uncertainty into account. This results in augmented contextual goal models (CGMs) that make the uncertainty explicit. Using the augmented CGM, the approach initiates its translation process, in which it automatically generates (i) a MDP model, and (ii) reliability and cost parametric formulae of the SAS. To this end, the *"CGM to MDP Transformation"* activity defines MDP models for each node type in the CGM. Next, the *"Verification Property Synthesis"* activity specifies the reliability and cost probabilistic properties related to the fulfillment of a goal in the CGM. Using the defined per node type MDP models and verification properties, the approach defines symbolic formulae for each node type in the CGM, through symbolic model checking. Finally, the final steps are the "MDP model composition" and "Parametric Formulae Composition" activities. In the former, we associate the reliability properties with the defined per type MDP models to compose the system's MDP model. In the latter, the

[1]The implementation is available at GitHub `https://github.com/lesunb/pistarGODA-MDP`. The modeling and analyzing environment of our framework piStarGODA-MDP is available at Heroku `https://seams2019.herokuapp.com/`.

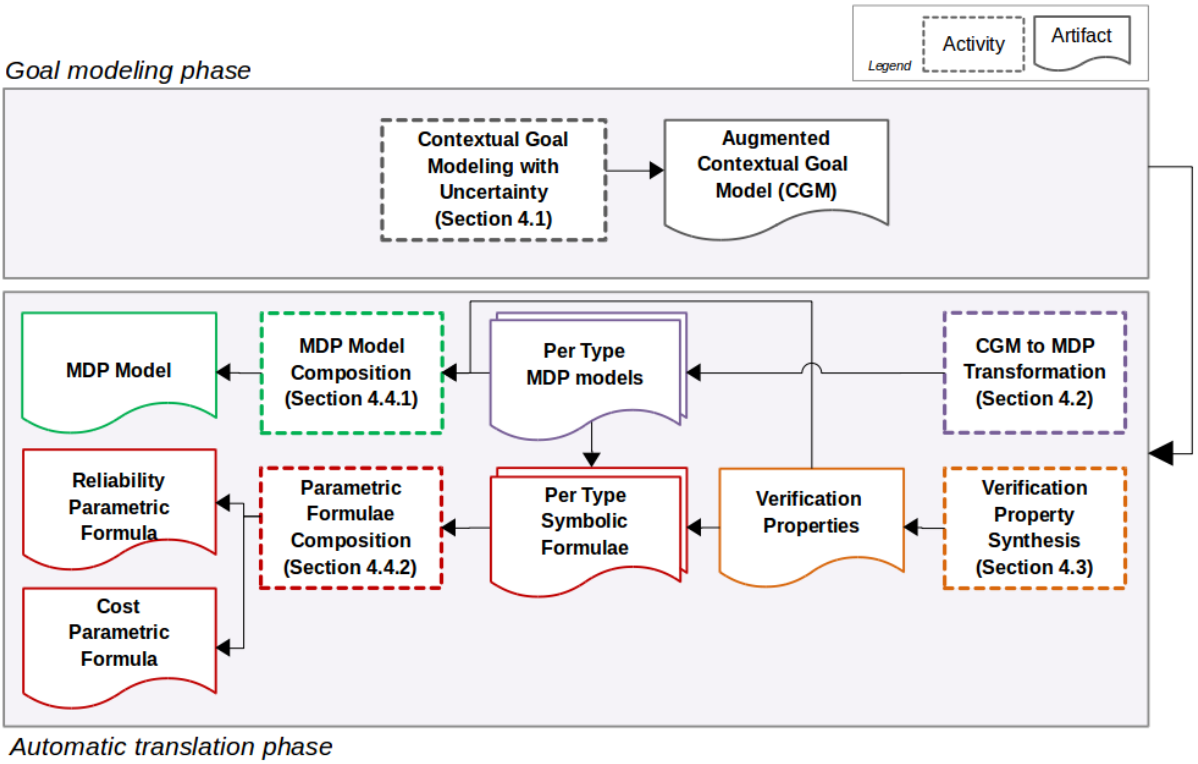


Figure 3.1: Approach's Overview to Support Uncertainty.

verification properties are associated with the per type symbolic formulae to compose the system's reliability and cost formulae. These generated models keep the uncertainty explicit.

In the remainder of this chapter, we detail the approach and discuss each of the activities shown in Figure 3.1.

3.1 Contextual Goal Modeling with Uncertainty

The first step of our approach consists of designing a goal model of a SAS that takes uncertainty into account. For this purpose, we augment CGM and introduce annotations to support three classes of uncertainty: (1) the system itself, (2) system goals, and (3) the environment, as well as non-deterministic behavior. According to Mahdavi-Hezavehi *et al.* [5], uncertainty related to the system itself refers to the managed system (i.e., the subsystem under adaptation); uncertainty related to system goals refers to the specification, modeling, and alteration of goals; and uncertainty related to the environment refers to environmental circumstances that interact with or affect the system.

In our work, each executable task (i.e., leaf task) of the CGM is implemented by a component in the managed system that has an identification label, description, execu-

tion probability (or equivalently, service usage profile), execution cost, and reliability. In particular, the execution probability indicates the execution profile over time, the cost indicates the energy consumption, and the reliability represents the probability of a successful execution of the component. Thus, the mapping of leaf tasks to components connects the CGM to the managed system. On the other hand, the context variable in the CGM is an outcome of the context operationalization process [28, 29] that has an identification label, description, and Boolean value that indicates whether a context is currently active or not, thus connecting the CGM to the managed system's environment.

In the following, we discuss the CGM extensions for uncertainty of the system itself, system goals, and environment, and the CGM extension to model non-deterministic behavior as well.

3.1.1 System Itself

Regarding uncertainty related to the system itself, we focus on two sources: *future parameter value*, and *incompleteness* [5]. *Future parameter value* relates to variables in the managed system that can affect system properties [1]. These are relevant variables for decision-making, whose value at runtime are not possible (nor practical) to predict at design time. *Incompleteness* manifests when some parts of the system or its model are knowingly missing [18] and may be added at runtime [1].

To represent *future parameter value* in the CGM, we associate the reliability and cost of the managed system's components with their corresponding leaf tasks. In Figure 3.2, these parameters are the annotation *[R,W]*, where R refers to reliability and W to cost. We should note that such parameters are intrinsic to each leaf task in our augmented CGM so that they are not required to be explicitly attributed to any particular value and neither explicitly modeled. Optionally, cost attribute can be modeled following the grammar presented in Listing 3.6. In contrast, the values can be obtained by monitoring the execution of the managed system's components, for example.

The *incompleteness* of the system is represented by a placeholder node "X" for a goal or task that is not entirely known at design time. A replacement for such a placeholder is only known at runtime. However, we should note that the replacement is not mandatory, i.e., there may or may not be a resource to implement the placeholder node during SAS execution. For instance, task "T1.X" in Figure 3.3 can be replaced at runtime by some data collection task that is unknown at design time. In this sense, if BSN can obtain vital sign data through resources different than its sensors (SaO2, TEMP, ECG, ABP) at runtime, such data collection replaces task "T1.X" in the CGM. Otherwise, "T1.X" does not become a part of the fulfillment of task "T1: Monitor vital signs".

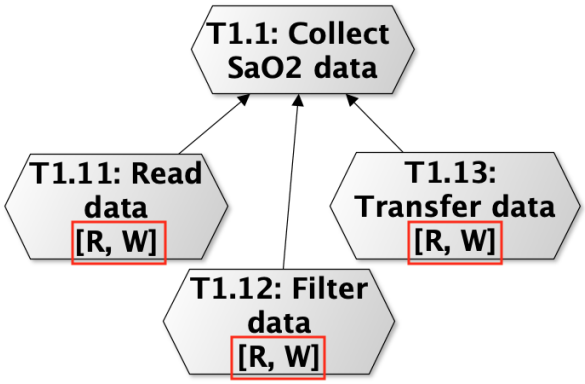


Figure 3.2: Future Parameter Value Uncertainty.

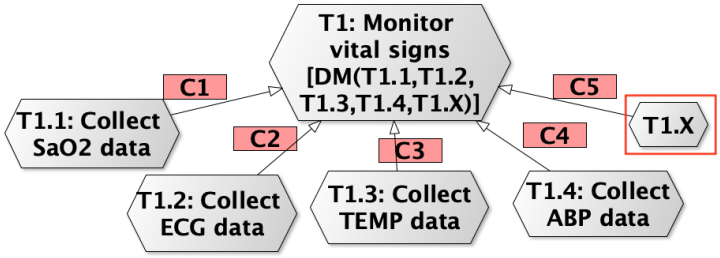


Figure 3.3: Incompleteness Uncertainty.

3.1.2 System Goals

The support for uncertainty related to system goals focuses on the *specification of goals* [5], specifically on enhancing the accuracy of the specification of stakeholders' preferences [1]. To this end, we exploit the structure of the CGM alongside the operationalization of its contexts. Since contexts influence not only on the satisfaction of goals, but also on their quality, we use context operationalization, i.e., the process of defining contexts in terms of variables and their values [29], to support the specification of stakeholders' preferences.

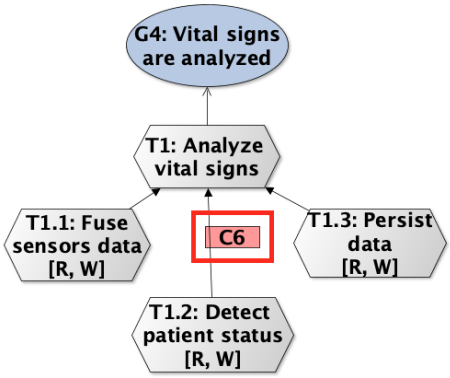
In the BSN, for instance, the detection of a patient health status is determined based on the vital signs collected and such status is individual for each patient, since a valid range for vital signs may vary differently depending on the patient profile (e.g., ECG data variability may have a greater impact on a cardiac patient then on other patients, therefore the ECG range should be adjusted accordingly). In this sense, stakeholders (such as domain experts) should update the specification of health status for each patient so that vital signs are appropriately analyzed and, hence, an emergency is detected according to the their preferences. Note that both goals (respectively G4 and G1 in Figure 2.2) could still be satisfied if a unique and default range of sensor data were to be used for all

patients. However, the *lack of accuracy when specifying such preference* could lead to a decrease of BSN trustworthiness (such as wrongly detecting, or not, an emergency).

Figure 3.4 shows an example of how context annotations in the CGM can be used to specify the preferences of stakeholders. Context C6 means that either *"low_risk"*, *"medium_risk"* or *"high_risk"* is true. Such statements, however, are represented by different combinations of the contextual variables. Considering the data presented in Table 2.1, for example, the "high_risk" is defined by the variables:

- $(SaO2 < 55)$, or
- $(Ecg \geq 115 | Ecg < 70)$, or
- $(Temp \geq 41 | Temp < 32)$, or
- $(Acg\_systolic \geq 100 | Acg\_diastolic \geq 90)$.

By replicating this process for both "low_risk" and "medium_ risk", we have the routine of the operationalization of the context C6.



Legend for Context Information

```
C6: low: (SaO2 >= 65) & (Ecg >= 87 & Ecg < 97) & (Temp >= 36 & Temp < 38) &
         (Acg_systolic < 120 & Acg_diastolic < 80)
   | medium: (SaO2 < 65 & SaO2 >= 55) | (Ecg >= 97 & Ecg < 155) | (Ecg >= 70 & ecg < 85) |
         (Temp >= 38 & Temp < 41) | (Temp >=32 & Temp < 36) | (Acg_systolic >= 120 & Acg_systolic < 140) |
         (Acg_diastolic >= 80 & Acg_diastolic < 90)
   | high: (SaO2 < 55) | (Ecg >= 115 | Ecg < 70) | (Temp >= 41 | Temp < 32) |
         (Acg_systolic >= 140 | Acg_diastolic >= 90)
```

Figure 3.4: Specification of Goals Uncertainty.

We define a grammar for context annotations and its operationalization in Listing 3.8. We should note that, for now, we do no support a dynamic change in the operationalization of contexts. In other words, the specification of such system goals preferences must be explicit in the CGM at design time.

3.1.3 Environment

Regarding uncertainty related to the environment, our approach supports *execution context* and *noise in sensing* [5]. The *execution context* represents, in the CGM, the *environmental* context in which SAS operates and its inherent unpredictability [5]. *Noise in sensing* relates to sensors' inaccuracy in providing data, since they are not ideal devices.

The *execution context* is inserted in the CGM as context annotations. Such annotations are associated with a node n and they can assume *integer*, *double*, or *Boolean* value types. Different context annotations can be associated with one another by using AND (&) or OR (|) operators. Different from the specification of goals, which are static annotations for each system execution to represent the preferences of stakeholders, the execution context states environmental conditions in which the system operates. It works as guard conditions towards the satisficement of a goal/task. In Figure 2.2, C1 to C5 represent the environmental contexts in which the BSN operates and that needs to be monitored to determine a suitable behavior of the BSN during execution. Figure 3.5 brings the meaning of each execution context. Listing 3.8 presents the grammar for context annotations in the CGM.

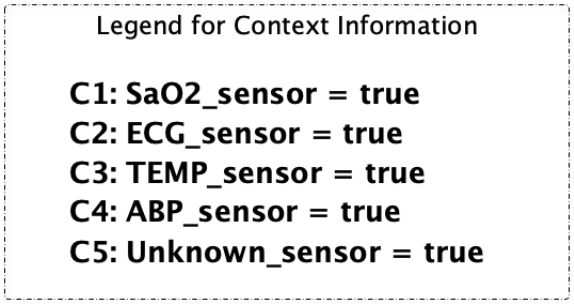


Figure 3.5: Context Conditions of BSN..

To support *noise in sensing*, we acknowledge the use of sensors by SAS (e.g., cyber-physical SAS) in its monitoring infrastructure to measure events in its environment [30]. We define a modeling guideline to take into account the data collection behavior of sensors. In this sense, a sensor's task of collecting data is decomposed into three essential sub-tasks. The first task, *read data*, is responsible for gathering data from the environment. The second task, *filter data*, is responsible for filtering and removing noise in the gathered data. Finally, the third task, *transfer data*, sends the filtered data to the managing system. For instance, in Figure 3.6, task "T1.1: Collect SaO2 data" and "T1.4: Collect ABP data" exemplify the use of our guideline for the blood oxygen saturation and arterial blood pressure sensors, respectively. We should note that the guideline sub-tasks implementation is domain dependent, i.e., we provide a guideline to account for noise in sensing but the

system engineer is responsible for detailing the actual implementation of these sub-tasks. With this guideline, we support the use of sensors in SAS while mitigating the *noise in sensing* associated with them through dedicated tasks.

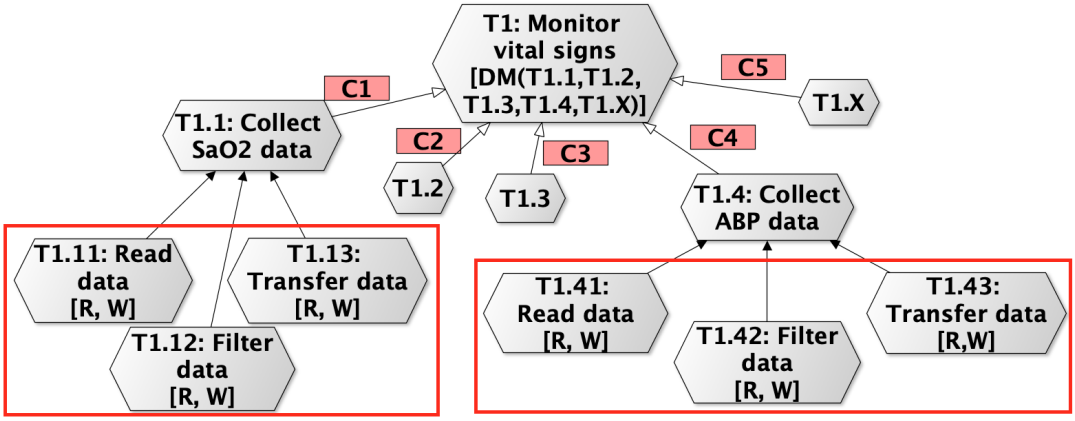


Figure 3.6: Noise in Sensing Uncertainty.

3.1.4 Non-deterministic Behavior

Another extension of the CGM is the possibility of defining a system's non-deterministic behavior, and therefore the necessity that SAS makes a decision of which action to choose to continue its execution. For this purpose, we define the *decision-making (DM)* annotation. Non-determinism in SAS is brought by uncertainty, such that the lack of runtime knowledge implicates planning multiple alternative behaviors at design time to accommodate possible configurations the system may face during execution. In this sense, the DM-annotation indicates the different paths in the CGM that satisfies a node, each one with its corresponding reliability and cost attributes. Thus, following decision theory, SAS managing system will choose the path based on the desirability of their outcome, i.e., on which path maximizes the SAS' quality preferences [31].

The DM-annotation is only inserted in non-leaf nodes, and it requires the node to be refined into context-dependent sub-trees by an OR-decomposition. For instance in Figure 2.2, task "T1: Monitor vital signs" has the annotation $DM(T1.1, T1.2, T1.3, T1.4, T1.X)$, which specifies that the successful execution of a combination of the context-dependent sub-trees T1.1, T1.2, T1.3, T1.4, or T1.X will result in the successful fulfillment of T1. In other words, T1 has up to $2^5 - 1$ different ways to be satisfied, but the decision of which way to pursue is only made at runtime, after mapping each context of T1's sub-trees to a value and eliciting the viable paths. For example, assuming that BSN is only operating with SaO2 and ECG sensors at a given time. This means that only contexts C1 and C2

are evaluated to *true*, therefore only tasks T1.1 and T1.2 may fulfill their parent node T1 (see Figure 2.2). In this sense, the viable paths to fulfill T1 are: (i) through only T1.1; (ii) through only T1.2; or (iii) through both T1.1 and T1.2. The path that maximizes the SAS' quality preferences is chosen to fulfill T1.

By providing this annotation, our work avoids the enumeration of all possible paths brought by context variability, therefore avoiding a state space explosion problem in the goal model. Moreover, it provides an explicit modeling of decision-making in SAS.

3.2 CGM to MDP Transformation

From the CGM of the system, we automatically generate its corresponding Markov Decision Process (MDP) model in PRISM language [32]. We use MDP for its ability to model both non-deterministic and probabilistic behavior [22]. The main purpose of this automation is to reduce overhead and errors caused by the manual generation of the verification model. By providing a MDP model, we aim at enabling software engineers to reason over the behavior of the system and to analyze system properties at design time, such as reliability and cost, while taking uncertainty into account. To this end, probabilistic model checking is suitable for reasoning about trustworthiness requirements of SAS.

The first step to generate a MDP model from a CGM is to define the PRISM templates that represent each node type in the CGM. The state diagram of Figure 3.7 describes the states of a leaf task in the CGM. For each of these states, there is a corresponding value for the state variable s in the PRISM model. State **Initial** (s = 0) corresponds to the initial state of a leaf task. Then, a leaf task can enter state **Running** (s = 1) if it is selected to start execution, or the final state **Skipped** (s = 3) if it does not participate in the fulfillment of the parent goal. Before, it is verified whether the context condition of a task, if existent, is satisfied or not. Once started running, the result of the task fulfillment is **Success** (s = 2) if the task has successfully been executed, otherwise **Failure** (s = 4).

Listing 3.1 presents the PRISM template for a context-dependent leaf task $N1$, while Listing 3.2 presents the PRISM template for a context-free leaf task $N1$. In particular, $c1$ is a binary placeholder that assumes value 1 (true) when the *execution context* in the leaf task $N1$ holds, and value 0 (false) otherwise; and $r1$ represents the *reliability* of the task (i.e., the success probability of a task), assuming real value in the range of 0 to 1. Moreover, $s1$ models the state of the leaf task. State **init** (s1 = 0) corresponds to the initial state. The probability of moving from **init** (s1 = 0) to **running** state (s1 = 1) is $c1$ if the leaf task is context dependent, and 1 otherwise. Consequently, $1 - c1$ represents the probability of moving from **init** (s1 = 0) to **skipped** state (s1 = 3) for a context-dependent leaf task. Once started running, the result of the task fulfillment is **success** (s

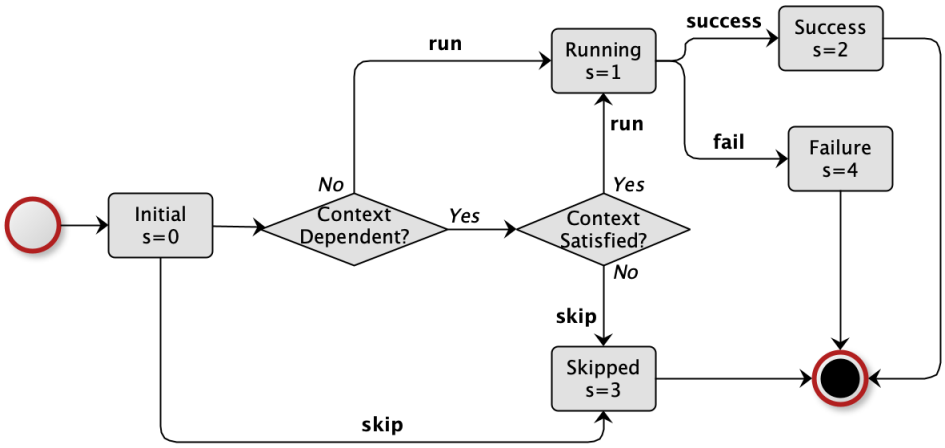


Figure 3.7: States of a CGM's Leaf Task.

= 2) with probability r, if the task has successfully been executed, otherwise **failure** (s = 4) with probability $1 - r$. The *cost* of executing the task is represented by $w1$ in the reward structure. Label *next* is used to tag a transition that needs to be synchronized with other transitions in the same or in different modules, while variable x represents an index that sequentially sets the actions of the model. The MDP for a goal G_i is obtained by composing the MDP models of its sub-trees.

```
const int c1; //context condition of n1
const double r1; //n1 probability of success

module N1
    s1 :[0..4] init 0;

    //init to running or to skipped
    [next<x>] s1 = 0 -> c1 : (s1'=1) + (1 - c1) : (s1'=3);
    [] s1 = 1 -> r1 : (s1'=2) + (1 - r1) : (s1'=4); //running to final state
    [next<x+1>] s1 = 2 -> (s1'=2); //final state success
    [next<x+1>] s1 = 3 -> (s1'=3); //final state skipped
    [next<x+1>] s1 = 4 -> (s1'=4); //final state failure
endmodule
rewards "cost"
    s1 = 1 : w1; //cost of n1 execution
endrewards
```

Listing 3.1: CGM Context-dependent Leaf Task in PRISM Language.

```
const double r1; //n1 probability of success

module N1
    s1 :[0..4] init 0;

    //init to running
    [next<x>] s1 = 0 -> (s1'=1);
    [] s1 = 1 -> r1 : (s1'=2) + (1 - r1) : (s1'=4); //running to final state
    [next<x+1>] s1 = 2 -> (s1'=2); //final state success
```

```
10      [next<x+1>] s1 = 3 -> (s1'=3); //final state skipped
11      [next<x+1>] s1 = 4 -> (s1'=4); //final state failure
12  endmodule
13  rewards "cost"
14      s1 = 1 : w1; //cost of n1 execution
15  endrewards
```

Listing 3.2: CGM Context-free Leaf Task in PRISM Language.

To represent a CGM node with a DM-annotation, we model the non-determinism in PRISM as shown in Listing 3.3. Listing 3.3 presents the non-deterministic module for a node with the $DM(N1, N2)$ annotation, where $N1$ and $N2$ are sub-trees. The CTX parameters represent each combination of context conditions modeled in the node's sub-trees. They assume integer values of 1 or 0 to indicate whether the represented context conditions are satisfied or not. Lines 11–14 present the non-determinism in the model that depends on the values of the CTX parameters to be resolved. Once the non-determinism is resolved, the global variables $c1$ and $c2$ are either set to 1, enabling the execution of its corresponding leaf task, or to 0, indicating that the corresponding leaf task will enter skipped state.

```
1  const int CTX_1; //context condition of n1
2  const int CTX_2; //context condition of n2
3  const int CTX_3; //context condition of n1 & n2
4
5  global c1: [0..1] init 0; //variable that enables n1
6  global c2: [0..1] init 0; //variable that enables n2
7  module NonDeterminism
8      s :[0..2] init 0;
9
10     [next<x>] s = 0 -> (s'=1);
11     [] s = 1 -> CTX_1 : (c1'=1) & (s'=2) + (1 - CTX_1) : (s'=1);
12     [] s = 1 -> CTX_2 : (c2'=1) & (s'=2) + (1 - CTX_2) : (s'=1);
13     [] s = 1 -> CTX_3 : (c1'=1) & (c2'=1) & (s'=2) + (1 - CTX_3) : (s'=1);
14     [] s = 1 -> (s'=2); //no uncertainty holding
15
16     [next<x+1>] s = 2 -> (s'=2);
17 endmodule
```

Listing 3.3: CGM Node with a DM-annotation in PRISM Language.

To represent the incompleteness uncertainty of the CGM in the PRISM model, we model a placeholder for the incomplete node following Figure 3.7 while acknowledging the node is optionally fulfilled. Listing 3.4 presents the PRISM template for the incomplete node NX. Assuming an available resource to fulfill such node, if the system wants to use such resource in its execution, the OPT_X parameter is valued to 1; otherwise, it is valued to 0. In case there is no available resource, OPT_X is valued to 0. Even though the MDP PRISM model is built for design-time analysis and the availability of unknown resources is only verified at runtime, by modeling incomplete nodes our approach supports simulation of runtime scenarios regarding possible resources availability.

```
const int OPT_X; //variable to represent nX's optionality
const double rX; //nX probability of success

module NX
    sX :[0..4] init 0;

    //init to running or to skipped
    [next<x>] sX = 0 -> OPT_X : (sX'=1) + (1 - OPT_X) : (sX'=3);
    [] sX = 1 -> rX : (sX'=2) + (1 - rX) : (sX'=4);//running to final state
    [next<x+1>] sX = 2 -> (sX'=2); //final state success
    [next<x+1>] sX = 3 -> (sX'=3); //final state skipped
    [next<x+1>] sX = 4 -> (sX'=4); //final state failure
endmodule
rewards "cost"
    sX = 1 : wX; //cost of nX execution
endrewards
```

Listing 3.4: CGM Incomplete Node in PRISM Language.

The reliability of goals is represented as formulas constructs in the PRISM language, and their value is given by means of the probabilistic existence property [26] presented next.

3.3 Reliability Property Synthesis

Once the MDP templates are established for each CGM node type, we define the PCTL properties for verification. In this work, we focus on reliability and cost properties. Their specifications are based on the idea of the *probabilistic existence property* [26], that is, the probability that a system will *eventually* reach a state that satisfies a goal of interest. The cost property summarizes the cumulative cost when reaching such state. Because our model exhibits non-deterministic behavior, a probability measure is feasibly defined once all non-determinism is removed [11]. Hence, PCTL properties for MDP models reason about the minimum and maximum probability, over all possible ways of resolving non-determinism, that a certain behavior is observed [11]. The maximum and minimum reliability and cost of fulfilling a goal G_i is defined in Table 3.1, where proposition ϕ represents the success of G_i and ϕ is recursively formed by composing the propositions of the nodes underlying G_i in the CGM.

Proposition ϕ varies for each node in the CGM according to the node's types: AND/OR-decomposition, DM-annotation, and incompleteness. Table 3.2 specifies the proposition for each type, where $i, j \in \mathbb{N}_{\geq 1}$ are nodes in the CGM. Note that a node specifying *incompleteness* in the system is optional to the system's overall fulfillment, since it depends on the availability of an unknown runtime resource. Note also that propositions for nodes with OR-decomposition and DM-annotation are similar, although the meaning of each type is different. On the one hand, the OR-decomposition establishes that sub-nodes are

Table 3.1: Reliability and Cost Properties for Verification.

Property	**PCTL formula**
Reliability (max)	$Pmax_{G_i} =? [F\ (\phi)]$
Reliability (min)	$Pmin_{G_i} =? [F\ (\phi)]$
Cost (max)	$Cmax_{G_i} =? [F\ (\phi)]$
Cost (min)	$Cmin_{G_i} =? [F\ (\phi)]$

executed sequentially, stopping after the first successful execution or after the last node's execution. A successful execution of one of the sub-nodes validates the proposition of success of the parent node. On the other hand, the DM-annotation establishes the execution of all sub-nodes chosen in the decision-making activity, aiming the combination that best suits SAS quality preferences. Nonetheless, if a different set of the chosen sub-nodes executes correctly, the parent node is satisfied (only with different quality attributes then expected), therefore validating its proposition of success.

Table 3.2: Proposition of Success of a Node in the CGM.

Node types	**Proposition**
AND-decomposition	$\phi = \bigwedge_{i=1}^{j} succeeded_i$
OR-decomposition	$\phi = \bigvee_{i=1}^{j} succeeded_i$
DM-annotation	$\phi = \bigvee_{i=1}^{j} succeeded_i$
Incompleteness	$\phi = succeeded_i \vee skipped_i$

Figure 3.8 illustrates the composition of proposition ϕ for a fraction of the BSN's goal model. The success of goal G3 is the success of "T1: Monitor vital signs". T1 has a DM-annotation, hence its success is defined as the corresponding proposition presented in Table 3.2. Sub-trees T1.1 and T1.2 both contain AND-decomposition, thus they succeed when all of their respective sub-trees succeed. Finally, each of the leaf nodes have their own execution module (illustrated in Figure 3.7), in which they succeed by reaching final state ($s = 2$). The success proposition for G3 is given by Equation 3.1.

$$\begin{aligned} \phi_{G3} = (&sT1\_11 = 2 \wedge sT1\_12 = 2 \wedge sT1\_13 = 2) \\ \vee\ (&sT1\_21 = 2 \wedge sT1\_22 = 2 \wedge sT1\_23 = 2) \end{aligned} \tag{3.1}$$

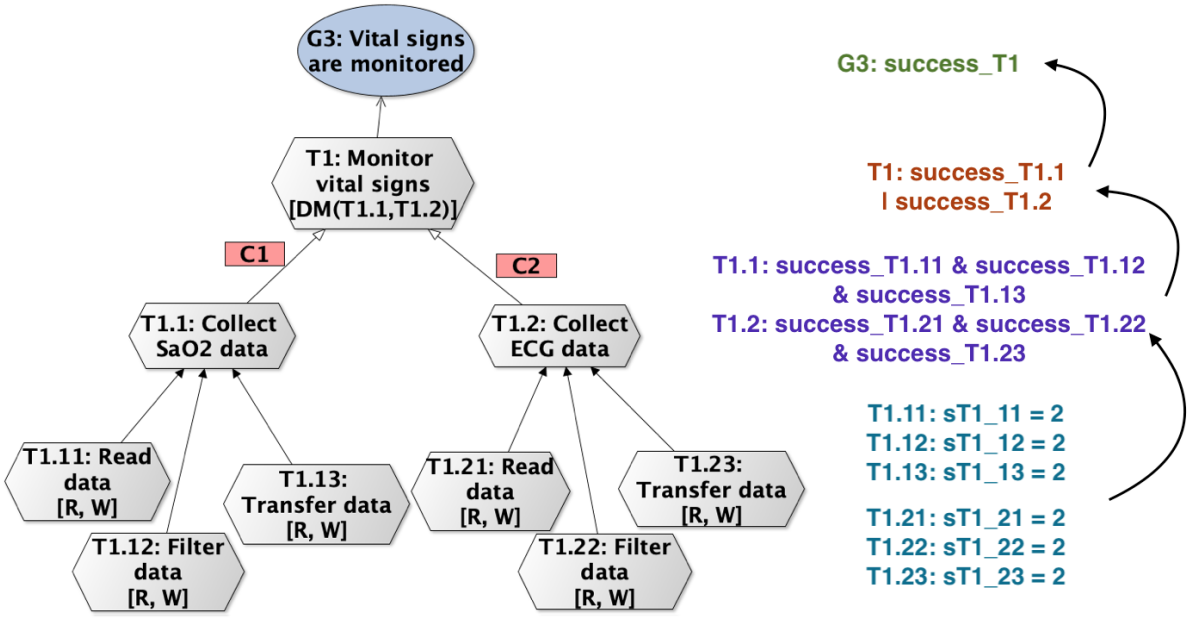


Figure 3.8: Example of Composing a Success Proposition.

3.4 Model Composition

Once the MDP models and the verification properties have been defined for each node type (i.e., AND/OR-decomposition, DM-annotation, and incompleteness) (Section 3.2 and Section 3.3, respectively), the approach initiates the model composition step. The following sections explain the composition of a MDP model for design-time verification via PRISM (Section 3.4.1) and parametric formulae for runtime verification (Section 3.4.2).

3.4.1 MDP Model Composition

Considering that CGM goals are realized by leaf tasks, the overall system behavior is the composition of these tasks. In this sense, the fulfillment of CGM goals, mapped as ϕ propositions, depends on the successful execution of their leaf tasks (following Figure 3.7). Accordingly, to generate a MDP model of a CGM, we follow a compositional approach. First, we retrieve the MDP models built for each leaf task (i.e., executable task), for each DM-annotated node and for each incomplete node (i.e., node modeling the incompleteness uncertainty) in the augmented CGM following the templates defined in Section 3.2. Second, we compose these MDP models to represent the fulfillment of the modeled goals. Third, we recursively compose propositions ϕ_{G_i} to represent the fulfillment of goal G_i by means of PRISM *formula* constructs (see Section 3.3). And finally, we build a PRISM *reward* constructs to keep track the cost of execution of each leaf task.

Take, for example, the CGM in Figure 3.8. Listing 3.5 presents its generated PRISM MDP. The MDP is composed by seven PRISM modules, one for each leaf task and one

for “T1: Monitor vital signs”, since this node contains the DM-annotation. Lines 3–5 contain the variables to represent the combination of the contexts conditions in the CGM (“SaO2_sensor = true” as C1, and “ECG_sensor = true” as C2). Lines 7 and 8 contain global variable responsible for enabling, or not, the execution of leaf tasks associated with the contexts. The non-deterministic module is presented in lines 10–21. In the sequence, the PRISM MDP model brings the leaf tasks modules (following Figure 3.7) alongside their reliability (R) parameter declaration. The fulfillment of goal G3 is defined as a PRISM formula in line 95. Lines 99–104 contain the cost parameters for each executable task (i.e., leaf task). Finally, lines 106–113 bring a PRISM reward constructs to account for the leaf tasks’ cost of execution.

```
1  mdp
2
3  const int CTX_1=1; // SaO2_sensor = true
4  const int CTX_2=1; // ECG_sensor = true
5  const int CTX_3=1; // SaO2_sensor = true & ECG_sensor = true
6
7  global CTX_G3_T1_1: [0..1] init 0; //C1
8  global CTX_G3_T1_2: [0..1] init 0; //C2
9
10 module NonDeterminism_G3_T1
11   sG3_T1 :[0..2] init 0;
12
13   [next_0] sG3_T1 = 0 -> (sG3_T1'= 1);
14
15   [] sG3_T1 = 1 -> CTX_1 : (sG3_T1'= 2) & (CTX_G3_T1_1'=1) + (1 - CTX_1) : (sG3_T1'=1);
16   [] sG3_T1 = 1 -> CTX_2 : (sG3_T1'= 2) & (CTX_G3_T1_2'=1) + (1 - CTX_2) : (sG3_T1'=1);
17   [] sG3_T1 = 1 -> CTX_3 : (sG3_T1'= 2) & (CTX_G3_T1_1'=1) & (CTX_G3_T1_2'=1) + (1 -
         CTX_3) : (sG3_T1'=1);
18   [] sG3_T1 = 1 -> (sG3_T1'=2); //no uncertainty holding
19
20   [next_1] sG3_T1 = 2 -> (sG3_T1'=2);
21 endmodule
22
23 const double R_G3_T1_11;
24 module G3_T1_11_ReadData
25   sG3_T1_11 :[0..4] init 0;
26
27   [next_1] sG3_T1_11 = 0 -> CTX_G3_T1_1 : (sG3_T1_11'=1) + (1 - CTX_G3_T1_1) : (sG3_T1_11
         '=3); //init to running or skip
28
29   [] sG3_T1_11 =  1 -> R_G3_T1_11 : (sG3_T1_11'=2) + (1 - R_G3_T1_11) : (sG3_T1_11'=4);//
         running to final state
30   [next_2] sG3_T1_11 = 2 -> (sG3_T1_11'=2);//final state success
31   [next_2] sG3_T1_11 = 3 -> (sG3_T1_11'=3);//final state skipped
32   [next_2] sG3_T1_11 = 4 -> (sG3_T1_11'=4);//final state failure
33 endmodule
34
35 const double R_G3_T1_12;
36 module G3_T1_12_FilterData
37   sG3_T1_12 :[0..4] init 0;
38
```

```
39   [next_2] sG3_T1_12 = 0 -> CTX_G3_T1_1 : (sG3_T1_12'=1) + (1 - CTX_G3_T1_1) : (sG3_T1_12
         '=3); //init to running or skip
40
41   [] sG3_T1_12 =  1 -> R_G3_T1_12 : (sG3_T1_12'=2) + (1 - R_G3_T1_12) : (sG3_T1_12'=4);//
         running to final state
42   [next_3] sG3_T1_12 = 2 -> (sG3_T1_12'=2);//final state success
43   [next_3] sG3_T1_12 = 3 -> (sG3_T1_12'=3);//final state skipped
44   [next_3] sG3_T1_12 = 4 -> (sG3_T1_12'=4);//final state failure
45 endmodule
46
47 const double R_G3_T1_13;
48 module G3_T1_13_TransferData
49   sG3_T1_13 :[0..4] init 0;
50
51   [next_3] sG3_T1_13 = 0 -> CTX_G3_T1_1 : (sG3_T1_13'=1) + (1 - CTX_G3_T1_1) : (sG3_T1_13
         '=3); //init to running or skip
52
53   [] sG3_T1_13 =  1 -> R_G3_T1_13 : (sG3_T1_13'=2) + (1 - R_G3_T1_13) : (sG3_T1_13'=4);//
         running to final state
54   [next_4] sG3_T1_13 = 2 -> (sG3_T1_13'=2);//final state success
55   [next_4] sG3_T1_13 = 3 -> (sG3_T1_13'=3);//final state skipped
56   [next_4] sG3_T1_13 = 4 -> (sG3_T1_13'=4);//final state failure
57 endmodule
58
59 const double R_G3_T1_21;
60 module G3_T1_21_ReadData
61   sG3_T1_21 :[0..4] init 0;
62
63   [next_1] sG3_T1_21 = 0 -> CTX_G3_T1_2 : (sG3_T1_21'=1) + (1 - CTX_G3_T1_2) : (sG3_T1_21
         '=3); //init to running or skip
64
65   [] sG3_T1_21 =  1 -> R_G3_T1_21 : (sG3_T1_21'=2) + (1 - R_G3_T1_21) : (sG3_T1_21'=4);//
         running to final state
66   [next_2] sG3_T1_21 = 2 -> (sG3_T1_21'=2);//final state success
67   [next_2] sG3_T1_21 = 3 -> (sG3_T1_21'=3);//final state skipped
68   [next_2] sG3_T1_21 = 4 -> (sG3_T1_21'=4);//final state failure
69 endmodule
70
71 const double R_G3_T1_22;
72 module G3_T1_22_FilterData
73   sG3_T1_22 :[0..4] init 0;
74
75   [next_2] sG3_T1_22 = 0 -> CTX_G3_T1_2 : (sG3_T1_22'=1) + (1 - CTX_G3_T1_2) : (sG3_T1_22
         '=3); //init to running or skip
76
77   [] sG3_T1_22 =  1 -> R_G3_T1_22 : (sG3_T1_22'=2) + (1 - R_G3_T1_22) : (sG3_T1_22'=4);//
         running to final state
78   [next_3] sG3_T1_22 = 2 -> (sG3_T1_22'=2);//final state success
79   [next_3] sG3_T1_22 = 3 -> (sG3_T1_22'=3);//final state skipped
80   [next_3] sG3_T1_22 = 4 -> (sG3_T1_22'=4);//final state failure
81 endmodule
82
83 const double R_G3_T1_23;
84 module G3_T1_23_TransferData
85   sG3_T1_23 :[0..4] init 0;
86
```

```
87   [next_3] sG3_T1_23 = 0 -> CTX_G3_T1_2 : (sG3_T1_23'=1) + (1 - CTX_G3_T1_2) : (sG3_T1_23
         '=3); //init to running or skip
88
89   [] sG3_T1_23 =  1 -> R_G3_T1_23 : (sG3_T1_23'=2) + (1 - R_G3_T1_23) : (sG3_T1_23'=4);//
         running to final state
90   [next_4] sG3_T1_23 = 2 -> (sG3_T1_23'=2);//final state success
91   [next_4] sG3_T1_23 = 3 -> (sG3_T1_23'=3);//final state skipped
92   [next_4] sG3_T1_23 = 4 -> (sG3_T1_23'=4);//final state failure
93 endmodule
94
95 formula G3 = ((sG3_T1_11=2 & sG3_T1_12=2 & sG3_T1_13=2) | (sG3_T1_21=2 & sG3_T1_22=2 &
       sG3_T1_23=2));
96
97 label "success" = G3;
98
99 const double W_G3_T1_11=1;
100 const double W_G3_T1_12=1;
101 const double W_G3_T1_13=1;
102 const double W_G3_T1_21=1;
103 const double W_G3_T1_22=1;
104 const double W_G3_T1_23=1;
105
106 rewards "cost"
107   sG3_T1_11 = 1 : W_G3_T1_11;
108   sG3_T1_12 = 1 : W_G3_T1_12;
109   sG3_T1_13 = 1 : W_G3_T1_13;
110   sG3_T1_21 = 1 : W_G3_T1_21;
111   sG3_T1_22 = 1 : W_G3_T1_22;
112   sG3_T1_23 = 1 : W_G3_T1_23;
113 endrewards
```

Listing 3.5: Example of PRISM MDP Model.

3.4.2 Parametric Formulae Composition

The generation of parametric formulae is a key enabler towards an affordable time-space *runtime analysis* [3]. Because of uncertainty and the corresponding non-deterministic behavior, model checking requires a Markov Decision Process (MDP) [23] to provide best- or worst-case scenarios analysis based on lower or upper probability bounds that can be guaranteed, when ranging over all possible paths [22]. Consequently, symbolic model checking of a whole CGM with uncertainty through a single MDP may present memory and time limitations due to the quantity of parameters involved. Our approach overcomes these limitations by accounting for uncertainty and by allowing the model checking of independent and smaller parts of the system, thus *compositionally* generating unique formulae and alleviating combinatorial state explosion problems.

With the MDP and property formulae for each node type (see Sections 3.2 and 3.3), we use the symbolic model checking PARAM [14] to generate their corresponding symbolic formulae. For models with context constraints, we restrict PARAM not to resolve context

variability, leaving context conditions parameterized in the formulae. In our approach, uncertainties related to context conditions are only resolved at runtime. Therefore, we obtain *unique* formulae for reliability and cost properties in terms of the context parameters, instead of maximum and minimum property formulae. For the models that do not require a context condition, PARAM results are the same for both maximum and minimum property analysis. Table 3.3 summarizes the formulae for the different types of a CGM node. Note that, in the first column, N_1, N_2, and N_x represent sub-trees. In the second and third columns, P_n represents the reliability, W_n represents the cost, and C_n represents the context condition of sub-tree n. C_n may assume values 1 (true) or 0 (false) to indicate whether the context holds or not at a given time. Note also that the symbolic formulae for N_x (that represents the incompleteness in the system) yields a variable OPT that is either 1 (true) or 0 (false) to render the existence of the node's resource (e.g., a sensor or component) at runtime.

Table 3.3: Symbolic Formulae.

Node type	**Reliability symbolic formula (P)**
AND $(N_1, ..., N_k)$	$\prod_{i=1}^{k} P_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
OR $(N_1, ..., N_k)$	$P_{n_1} + (1 - P_{n_1}) * P_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - P_{n_i})\right) * P_{n_k}$, where $k \in \mathbb{N}_{\geq 2}$
$DM(N_1, ..., N_k)$	$C_{n_1}P_{n_1} + (1 - C_{n_1}P_{n_1}) * C_{n_2}P_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - C_{n_i}P_{n_i})\right) * C_{n_k}P_{n_k}$, where $k \in \mathbb{N}_{\geq 2}$
Incompleteness (N_x)	$OPT_{n_X} * P_{n_X} - OPT_{n_X} + 1$

Node type	**Cost symbolic formula (W)**
AND $(N_1, ..., N_k)$	$P_{AND}(n_1, ..., n_k) * \sum_{i=1}^{k} W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
OR $(N_1, ..., N_k)$	$-P_{OR}(n_1, ..., n_{k-1}) * W_{n_k} + P_{OR}(n_1, ..., n_k) * \sum_{i=1}^{k} W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
$DM(N_1, ..., N_k)$	$-P_{DM}(n_1, ..., n_{k-1}) * C_{n_k}W_{n_k} + P_{DM}(n_1, ..., n_k) * \sum_{i=1}^{k} C_{n_i}W_{n_i}$, where $k \in \mathbb{N}_{\geq 2}$
Incompleteness (N_x)	$OPT_{n_X} * P_{n_X} * W_{n_X}$

Even though the formulae of only DM-annotation are in terms of context condition (C_n), since its mandatory for this node type, nodes of other types may also have context dependent formulae. In this sense, the formulae of a node with purely AND/OR-decomposition or and incomplete node (N_x), but whose sub-nodes are context dependent, follow Table 3.3 with parameter C_n multiplying the respective P_n and W_n. Also, in spite of the formulae for OR-decomposition and DM-annotation being similar, the semantics behind each structure is very different, since DM represents non-determinism in the system, therefore indicating a decision-making process in SAS.

From Table 3.3, one can note that the reliability and cost parameters of all CGM node's types grow following a geometric progression with respect to the number of sub-trees. For AND/OR-decomposition, the reliability and cost parameters grow with a common ratio 1 and 2, respectively, in the best-case scenario (i.e., when there are no context conditions involved). In the worst-case scenario (i.e., when all sub-trees are context dependent), the common ratio is 2 for reliability and 3 for cost parameters. The parameters of DM-annotation grow following a geometric progression with common ratio 2 for reliability and 3 for cost. For incomplete node N_x, in the best-case scenario the reliability and cost parameters grow with a common ratio 2 and 3, respectively. In the worst-case scenario, the common ratio is 3 and 4, for reliability and cost respectively.

To generate reliability and cost parametric formulae for the overall system, we exploit the symbolic formulae on Table 3.3 and the tree structure of goal models in a compositional way while taking into account the uncertainty modeled in the CGM: probability formulae of smaller goal models are computed and then composed to obtain the formula of its corresponding larger goal model. We follow a recursive depth-first strategy to visit the tree structure of the CGM. Each node gets a respective symbolic formulae in terms of the formulae associated with its sub-nodes. Leaf nodes get atomic formulae that are returned to rewrite their parents' formulae. Therefore, by the time the rewriting terminates, the system overall formulae will be a composition of its sub-trees formulae.

Algorithm 1 presents the compositional approach to build a parametric formulae. It starts from *node*, which is a local or the root goal of the CGM, and for which the parametric formula should be built. Line 1 stores the sub-tree nodes of *node* in the list *decNodes*. Line 2 fetches the decomposition type *decType*, either AND or OR, of *node*. Line 3 uses *dmAnnot* to store the fetched DM-annotation of *node*. Line 4 stores the context information of *node* in *ctxAnnot*. Line 5 calls function $getForm$, which returns in $nodeForm$ the parametric formula of *node* considering the decomposition type (*decType*) and DM-annotation (*dmAnnot*) according to the symbolic formulae listed in Table 3.3.

Accordingly and following a depth-first strategy, each *subNode* of the goal tree is traversed through a recursive call to $composeNodeForm$, which produces a $subNodeForm$ that replaces its corresponding ID symbol in $nodeForm$ (lines 6 to 9). Whenever the recursive approach finds a leaf task, the algorithm builds a parametric MDP of the task according to Figure 3.7, and uses the symbolic model checker PARAM to retrieve the task reliability and cost formulae. Finally, line 14 returns the parametric formula for the goal *node*.

Figure 3.9 exemplifies how the algorithm works for a fraction of the BSN's goal model to generate and compose a reliability formula. Since G3 has no specific type, its reliability will be the same as its sub-tree "T1: monitor vital signs". T1 has a DM-annotation,

Algorithm 1: composeNodeForm(Node node)

```
   Input: A node, either a root or a local goal
   Result: Parametric Symbolic formula of the node
 1 List [] decNodes ← getDecomposition(node);
 2 DecType decType ← getDecType(node);
 3 String dmAnnot ← getDecisionMakingRule(node);
 4 String ctxAnnot ← getContextInfo(node);
 5 String nodeForm ← getForm(decType, dmAnnot, node);
 6 foreach subNode in decNodes do
 7 |   String subNodeId ← getId(subNode);
 8 |   String subNodeForm ← composeNodeForm(subNode);
 9 |   replaceSubForm(nodeForm,subNodeForm,subNodeId);
10 end
11 if isLeafTask(node) then
12 |   nodeForm ← getParamForm();
13 end
14 return nodeForm;
```

therefore its reliability follows the symbolic formula defined in Table 3.3 for nodes with DM-annotation. Sub-trees T1.1 and T1.2 have both an AND-decomposition, thus the reliabilities of each sub-trees are multiplied to obtain the reliabilities of each, T1.1 and T1.2 (note that C1 and C2 are not considered in T1.1 and T1.2 symbolic formulae, but in their parent's T1 instead, as defined in Table 3.3). Finally, the leaf nodes have their reliability retrieved by PARAM, in which rT_i represents the reliability of leaf node i. Similarly, cost formulae are generated by the algorithm.

The variability of the CGM contexts at runtime will create a specific induced Discrete Time Markov Chain (DTMC) to resolve any non-determinism at runtime. Accordingly, the system overall formulae vary as well. Table 3.4 shows the different context conditions the system may face during execution, and the reliability formula for each of them. We should note that the formula is specified in terms of *reliability* and *context condition*, which are all uncertainties parameterized in the CGM. In the case of a cost formula, it would also be defined in terms of *cost* of nodes. The computation of the reliability and cost formulae is used to evaluate these attributes of the system based on the dynamic variance of (i) reliability, (ii) cost, and (iii) context conditions as reflected in the CGM.

Also, by mapping the formulae parameters, one is able to guide the synthesis of adaptation policies by retrieving valid combinations of nodes to be executed at a given time. In Figure 3.9, for example, the conjunction of holding contexts at a given time impacts on which leaf tasks could be executed, and, therefore, on G3 reliability and cost formulae (as seen in Table 3.4). Since adaptation policy synthesis in SAS requires, among other things, context and actions identification [33], our generated parametric formulae, along-

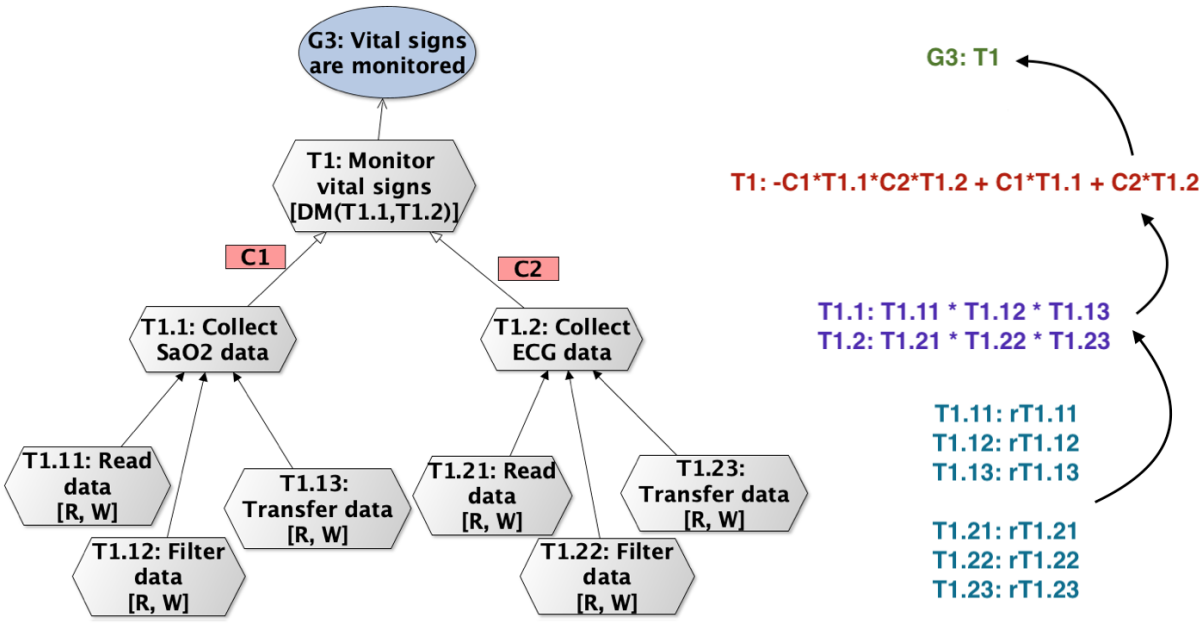


Figure 3.9: Example of Composing a Reliability Formula.

Table 3.4: Context Variability and Associated Formula

C1	C2	Reliability formula for G3
1	1	$-rT_{1.11}rT_{1.12}rT_{1.13} * C_1 * rT_{1.21}rT_{1.22}rT_{1.23} * C_2 + rT_{1.11}rT_{1.12}rT_{1.13} * C_1$ $+rT_{1.21}rT_{1.22}rT_{1.23} * C_2$
1	0	$rT_{1.11} * rT_{1.12} * rT_{1.13} * C_1$
0	1	$rT_{1.21} * rT_{1.22} * rT_{1.23} * C_2$

side augmented contextual goal model, provide means to easily retrieve the corresponding sets of executable tasks in a SAS given context variability. And, thus, supporting SAS engineers to reason about the possible actions that can be performed to modify the SAS managed system's behavior.

3.5 Automatic Generation of Verifiable Models

One of the contributions of this work is the piStarGODA-MDP framework, that automatically generates MDP models in PRISM language and parametric formulae based on a previous modeled CGM. Such automated generation aims at reducing the overhead and errors caused by a manual generation of verification models and, also, at abstracting the "know-how" of probabilistic modeling from the analysts, therefore optimizing the formal verification of systems.

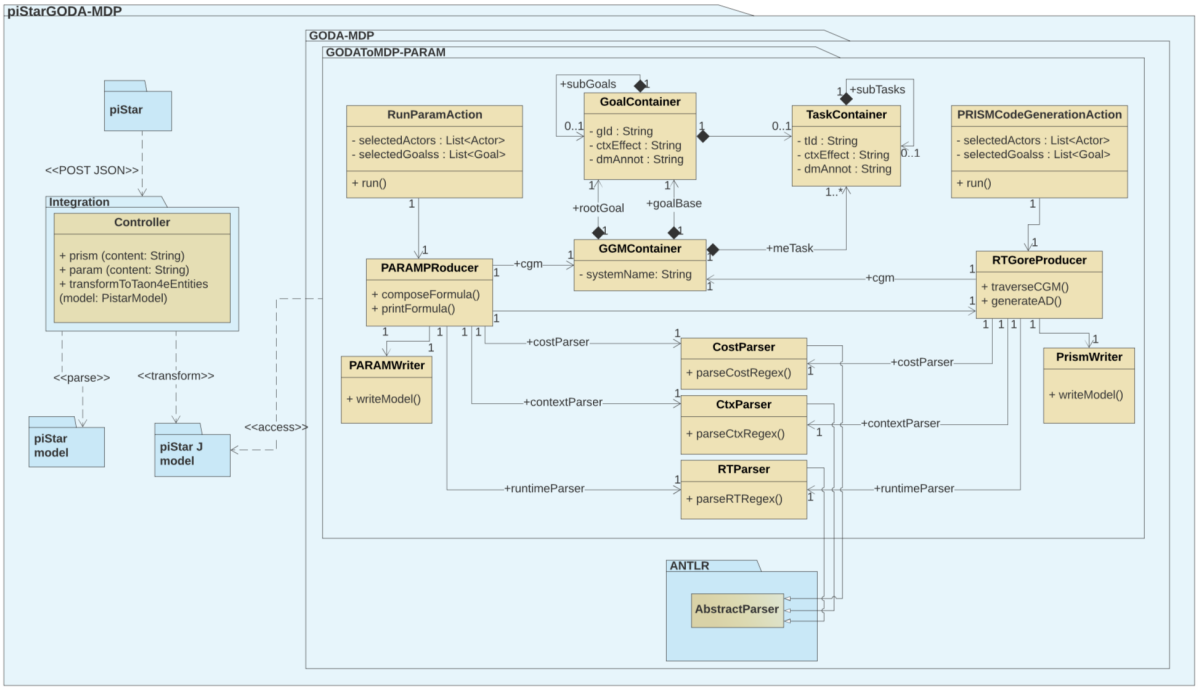


Figure 3.10: Implementation Architecture of the piStarGODA-MDP Framework.

The piStarGODA-MDP framework merges the back-end implementation architecture of GODA framework (Section 2.5) with the front-end implementation architecture of the piStar goal modeling tool [34]. The former is implemented in JAVA and the latter in JavaScript. The piStar goal modeling tool provides an online graphical environment for goal modeling compliant with the i* standard. Our piStarGODA-MDP framework is a product inherited from the integration of piStar and GODA projects provided by Bergmann [35]. Further information regarding the integration of these two projects is available in [35].

3.5.1 Implementation

The implementation architecture of piStarGODA-MDP is illustrated in Figure 3.10. The *Integration* package is responsible for receiving the modeled CGM from the piStar modeling tool and transforming it into JAVA objects used by the GODA-MDP package. The GODA-MDP package is responsible for the automatic generation of MDP or parametric formulae from CGM, called *CGM to MDP-PARAM*.

As originally proposed by GODA [36], the CGM to MDP-PARAM code generation process presents two phases:

1. **Parsing phase.** It is a common phase to generate both MDP model and parametric formula. In the parsing phase, the piStarJ model (i.e., input file containing

a goal model within a system actor that is transformed into JAVA objects by the Integration package) is parsed using a depth first algorithm. Container objects for goals and tasks are kept in memory with any relevant metadata. The cost annotation in any leaf task, the DM-annotation and the context effects in any goal or task are also parsed by ANTLR [37] generated APIs for the grammars in Listings 3.6, 3.7 and 3.8.

2. **Writing phase.** The writing phase is different whether the framework is generating a MDP model or parametric formulae.

   (a) **MDP model.** The container objects with all information required for the generation of a MDP model are traversed from the root goal container. MDP modules for leaf tasks and for each DM-annotated node are created and concatenated to a global variable. For each goal G_i in the model, a success formula declared as G_i is also concatenated to the model. These formulas can be part of the Probability existence PCTL properties specifying the probability in fulfilling one or more system goals, which eliminates the effort of manually building such formulas. Reliability and cost parameters of each leaf task and context conditions associated to goals and tasks are processed and their declaration is appended to the model. A reward module is concatenated to the model. Finally, the MDP model file output is made available for download by the framework.

   (b) **Parametric formulae.** The container objects with all information required for the generation of the reliability and cost parametric formulae are traversed from the root goal container. Using a depth first algorithm and based on Table 3.3, symbolic parametric formulae for each node is retrieved, always in function of its sub-nodes and possible context parameters. For each context condition in the goal model, is created a placeholder parameter whose value indicates the presence (1) or absence (0) of the corresponding context. For each leaf task in the model, a PRISM model containing only the leaf task module is built. Hence, PARAM model checker is invoked to retrieve the reliability and cost parametric formulae of each leaf task. Recursively, leaf tasks parametric formulae are returned to their parent node and replaced in the parent node symbolic formulae. The recursive approach continues the replacement back to the root goal, generating the final formulae. The formulae files output is made available for download by the framework.

The classes in Figure 3.10 interact as follows:

1. The user accesses the framework via a browser, and builds a model.

2. The user clicks on *"Generate PRISM MDP Model"* or *"Parametric formulas generation"*, in case he/she wants to generate a MDP model or the parametric formulae of the goal model built, respectively.
3. The website generates a JSON file of the goal model built by the user.
4. Through a POST request, the website sends the JSON file as a *String* to to *"prism"* or *"param"*.
5. The *Controller* class receives the request, parses the JSON file and generates the corresponding JAVA objects of the model.
6. The *Controller* class instantiates either the *"PRISMCodeGenerationAction"* or *"RunParamAction"* classes to generate the MDP model or the parametric formulae, respectively.
7. The system actor is identified and its root goal extracted.
8. The *"RTGoreProducer"* starts the depth-first algorithm by the root goal calling the *addGoal()* method.
9. Each goal or non-leaf task have their children elements extracted and saved as containers in the *CGMContainer* class instance before a recursive call to *addGoal()/addTask()*:
    - DM-annotation, all cost information and context annotations are parsed by *RTParser*, *CostParser* and *CtxParser*, respectively. The corresponding attributes are setted in the current node container.
    - A recursive call to *addGoal()/addTask()* is performed.
10. The recursive call return of a given child element is added to the parent element attributes to propagate nested time increments to subsequent goals/tasks.
11. The *AgentDefinition* file with all goals and tasks and a reference to the root goal is passed to the *PrismWriter* class or back to the *PARAMPRoducer* class to generate the MDP model or parametric formulae, respectively.
12. By the end of the verifiable model generation, the browser makes available for download a *"prism.zip"* or *"param.zip"* file, depending on the requested model.

3.5.2 ANTLR Grammars

Listing 3.6: Grammar for cost attribute in the CGM leaf tasks.

```
1 grammar CostRegex;
```

```
2
3  cost: 'W' op='=' FLOAT          # gFloat
4     | 'W' op='=' FLOAT VAR       # gExpression
5     | 'W' op='=' VAR             # gVariable
6     | NEWLINE                    # blank
7     ;
8
9  FLOAT        : DIGIT+'.'?DIGIT*          ;
10 VAR          : ('a'..'z'|'A'..'Z'|'_')+  ;
11 NEWLINE      : [\r\n]+                   ;
12 WS           : [\t]+ -> skip             ;
13
14 fragment
15 DIGIT        : [0-9]                     ;
```

Listing 3.7: Grammar for DM-annotation in the CGM.

```
1  grammar RTRegex;
2
3  rt:      expr NEWLINE                    # printExpr
4    |      NEWLINE                        # blank
5    ;
6
7  expr:    t=('G'|'T') id                  # gId
8      |    'DM(' expr ')'                   # gDecisionMaking
9      |    expr op=',' expr                 # gDM
10     ;
11
12 id:      FLOAT
13    |  FLOAT X
14    |  X
15    ;
16
17 FLOAT        : DIGIT+'.'?DIGIT*  ;
18 TASK         : 'T'               ;
19 GOAL         : 'G'               ;
20 X            : 'X'               ;
21 NEWLINE      : [\r\n]+           ;
22 WS           : [\t]+ -> skip     ;
23
24 fragment
```

```
25 DIGIT     : [0-9]                    ;
```

Listing 3.8: Grammar for context annotations in the CGM.

```
1  grammar CtxRegex;
2
3  ctx:     ctx NEWLINE                    # printExpr
4      |    'assertion condition 'expr     # condition
5      |    'assertion trigger 'expr       # trigger
6      |    NEWLINE                        # blank
7      ;
8
9  expr:    expr op='<' num                # cLT
10     |    expr op='<=' num               # cLE
11     |    expr op='>' num                # cGT
12     |    expr op='>=' num               # cGE
13     |    expr op='=' value              # cEQ
14     |    expr op='!=' value             # cDIFF
15     |    expr op='&' expr               # cAnd
16     |    expr op='|' expr               # cOr
17     |    VAR                            # cVar
18     |    '(' expr ')'                   # cParens
19     ;
20
21 value:   num                            # cNum
22     |    BOOL                           # cBool
23     ;
24
25 num:     INT                            #cInt
26     |    FLOAT                          #cFloat
27     ;
28
29 BOOL     : ('false'|'true')                ;
30 VAR      : NAME+                           ;
31 NAME     : ('a'..'z'|'A'..'Z'|'_')+DIGIT*  ;
32 INT      : DIGIT+                          ;
33 FLOAT    : DIGIT+'.'DIGIT*                 ;
34 NEWLINE  : [\r\n]+                         ;
35 WS       : (' '|'\t')+ -> skip             ;
36
37 fragment
```

```
38 DIGIT    : [0-9]                                    ;
```

Chapter 4

Evaluation

In this chapter, we evaluate our approach by means of a Goal-Question-Metric (GQM) methodology [38]. First, we focus on the scalability of the approach's translation process at design- and runtime. Second, we focus on the trustworthiness of our generated runtime models[1]. We evaluate the approach on the BSN and Tele Assitance System (TAS) [12] case studies. We use the example of a BSN [13] enriched with features to incorporate uncertainties that affect reliability and cost (e.g., power consumption) of the BSN. The BSN prototype used by this work was originally developed in [39] and continuously evolved by [40] and [33].

4.1 Tele Assistance System

Tele Assitance System (TAS) [12] is a service-based system exemplar that provides home health support to chronic ill patients. It was originally introduced in [41], and has already been used in the evaluation of different self-adaptation solutions [12]. In our work, though, we do not focus on providing solutions for self-adaptation, but on the evaluation of TAS' reliability and cost quality attributes.

The structure of TAS is as follows: a combination of sensors embedded in a wearable device and remote services (e.g., health care, pharmacy and emergency service providers) regularly measures the patient's vital signs. A third-party medical service is responsible for their analysis. The analysis result may trigger the invocation of: (i) a pharmacy service to deliver new medication to the patient or to change the medication dose, or (ii) an alarm service. The alarm service can also be invoked by the patient himself/herself via

[1]Conducted experiments used version prism-4.4-osx64 of PRISM and 2-3-64α of PARAM on an Intel Core i5-4278U CPU @ 2.60 GHz processor with 3.9 GB of RAM virtual machine, running Linux Ubuntu 16.04 LTS as operating system.

a panic button. When triggered, this service leads, e.g., to an ambulance being dispatched to the patient.

Figure 4.1 depicts a CGM that represents TAS behavior and intents following the requirements in [12]. The root goal "G1: Provide health support" can be realized by two different sub-goals: "G2: Provide automated support" or "G3: Provide self-diagnosed support". The choice of which sub-tree to pursue depends on whether or not the patient has triggered the panic button. Goal G2 is divided into three sub-goals: "G4: Get vital params", "G5: Analyze data", and "G6: Enact treatment". G4 is realized by operation task "T1: Get vital params". G5 is realized by a medical service containing three execution alternatives: medical services 1, 2 and 3. In case of adaptation, TAS changes from one alternative to another to keep the system running successfully. The same strategy is used with the alarm service (that fulfills goals "G8: Send alarm" and "G2: Provide self-diagnosed support"). Goal "G6: Enact treatment" is triggered when the patient is not fine, and it is fulfilled by the satisficement of goals "G7: Administer medicine' and "G8: Send alarm". A drug service is used for medicine administration, which could be a change of drug or a change of dose. Note that most of the operational tasks in TAS are performed by a remote service (such as alarm service, medical service and drug service).

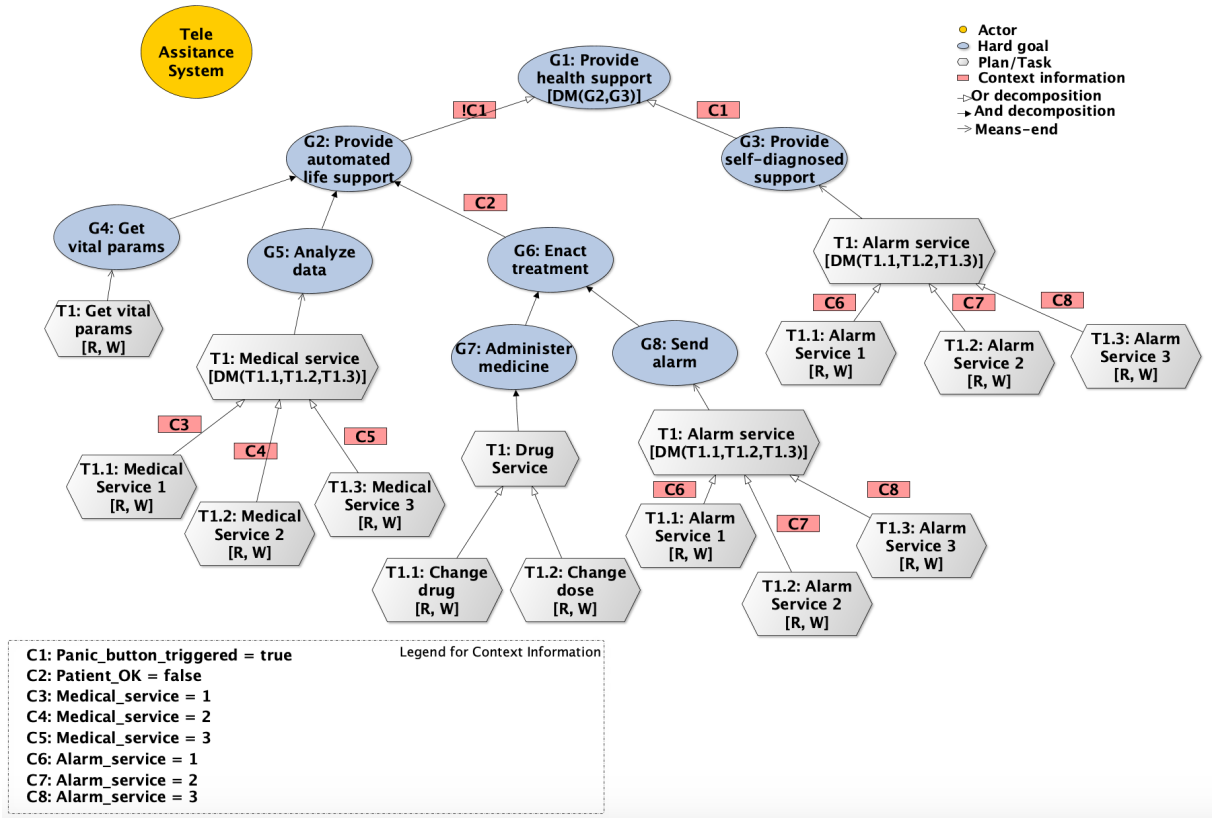


Figure 4.1: Contextual Goal Model of the Tele Assitance System.

The operation of TAS is subject to eight different context conditions. Context C1

refers to the enabling of the panic button by the patient himself/herself, which defines how the health support will be provided. Context C2 refers to the need of treatment for a patient, depending on his/her vital parameters analysis. Both C1 and C2 assume Boolean values. Contexts C3 to C5 refer to which medical will be responsible for analyzing the patient's data. Contexts C6 to C8 refer to which alarm service will be triggered, if necessary. Contexts C3 to C8 assume integer values in the range of 1 to 3. At runtime, TAS behavior changes depending on the the currently holding contexts, leading the system to fulfill its goals at different quality levels.

4.2 Scalability Analysis

The first evaluation goal **EG1** aims to provide a more comprehensive time-space scalability analysis of our translation process and its generated models. In order to better explore the scalability of the approach, we create generic CGMs with a growing number of leaf tasks for different structures. In this sense, we not only analyze the scalability for real systems like BSN and TAS, but we also investigate the approach's limits of scalability. As such, the following questions and metrics have been defined:

- **Q1**: What is the time-space trend of the automated generation of MDP models?
  - M1.1: The file size (KB) of the MDP model.
  - M1.2: The number of states in the MDP model.
  - M1.3: The number of transitions in the MDP model.
  - M1.4: The time consumed (s) to model check the MDP model.
- **Q2**: What is the time trend to generate the parametric formulae?
  - M2.1: The time consumed (ms) to build the reliability parametric formula.
  - M2.2: The time consumed (ms) to build the cost parametric formula.
- **Q3**: What is the time-space trend of a runtime verification using the formulae?
  - M3.1: The size (KB) and number of parameters of the parametric formulae.
  - M3.2: The time consumed (s) to evaluate the reliability parametric formula.
  - M3.3: The time consumed (s) to evaluate the cost parametric formula.

Setup

In order to assess the limits of the scalability analysis, we automatically create generic CGM models containing a single node parent with a growing number of its own leaf tasks.

Hence, in the evaluation, whenever the number of leaf tasks is mentioned, it should be clear that they are all sub-nodes refined from the same node parent, as illustrated in Figure 4.2.

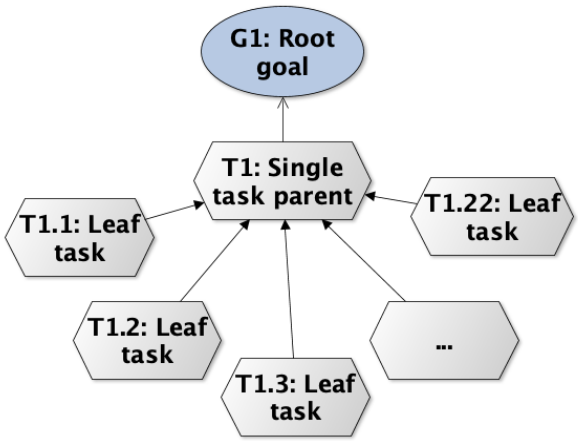


Figure 4.2: CGM Example Used by Scalability Analysis.

Question 1: What is the time-space trend of the automated generation of MDP models?

As for question Q1, Figure 4.3 shows that the size of the textual MDP model in PRISM language (M1.1) has a linear growth with respect to the number of leaf tasks when considering AND/OR-decomposition, but it grows exponentially when considering the DM-annotation. This trend is due to the 2^n context combinations that are modeled when using the DM-annotation, in which n is the number of leaf tasks. In our approach, we can generate MDP models in PRISM language with up to 3000 leaf tasks refined from AND/OR-decomposition. But, due to DM-annotation's exponential growth, we can generate at most 18 leaf tasks refined from such annotation. This means that the parent node has 18 context conditions to analyze and up to 2^{18} different ways to be fulfilled.

Figures 4.4a and 4.4b show that the number of states and transitions of the resulting MDP grows exponentially with the number of leaf tasks for both DM-annotation and AND/OR-decomposition. These growths reflect the state-space explosion problem faced by probabilistic model checking. Probabilistic model checking combines probabilistic analysis and conventional reachability in a single tool, providing full coverage of the executions and therefore exact answers, but the drawback is the state-space explosion [11].

Regarding BSN, which is a research prototype inspired by real system requirements and composed by different annotation and decomposition, its CGM has 17 leaf tasks refined from different parent nodes, as shown in Figure 2.2. Moreover, BSN has a MDP

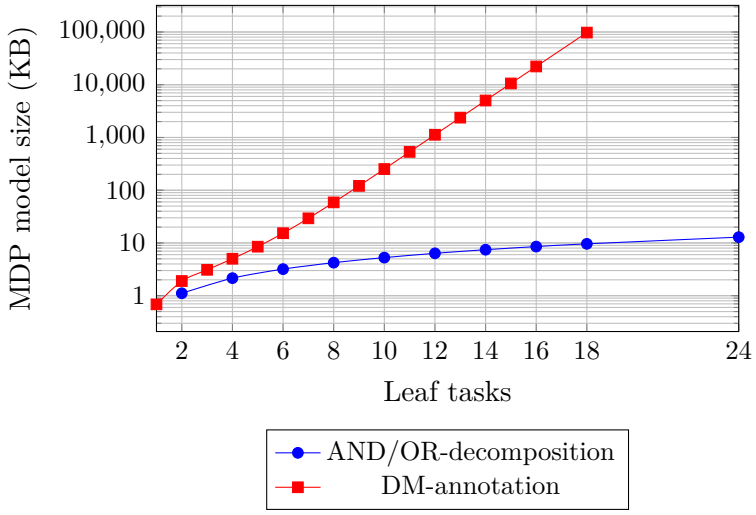
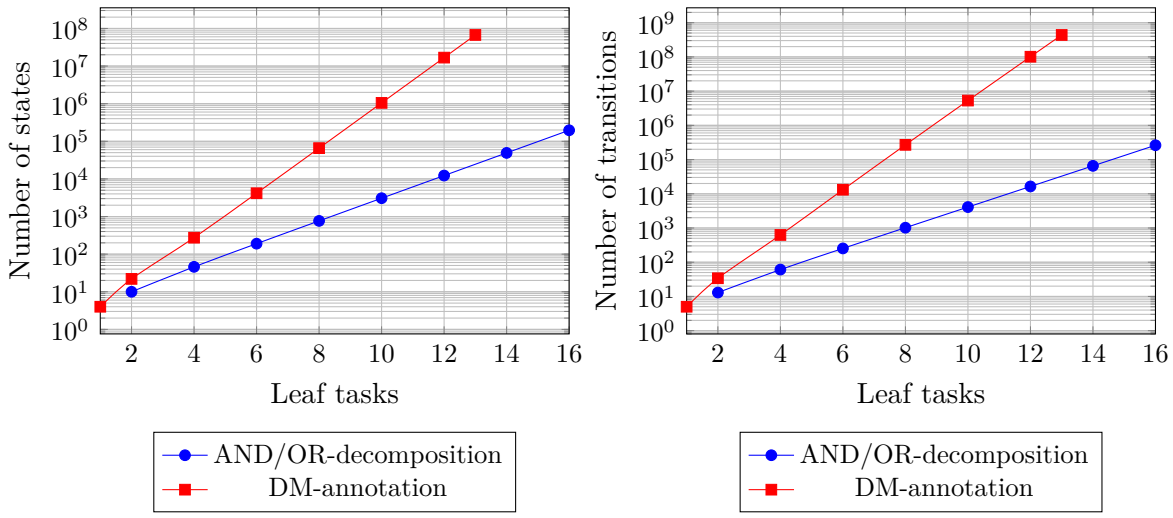


Figure 4.3: Size of the textual MDP model with respect to the increasing number of leaf tasks (M1.1).



(a) Number of MDP states.

(b) Number of MDP transitions.

Figure 4.4: Number of states and transitions of MDP model with respect to increasing number of leaf tasks (M1.2 and M1.3).

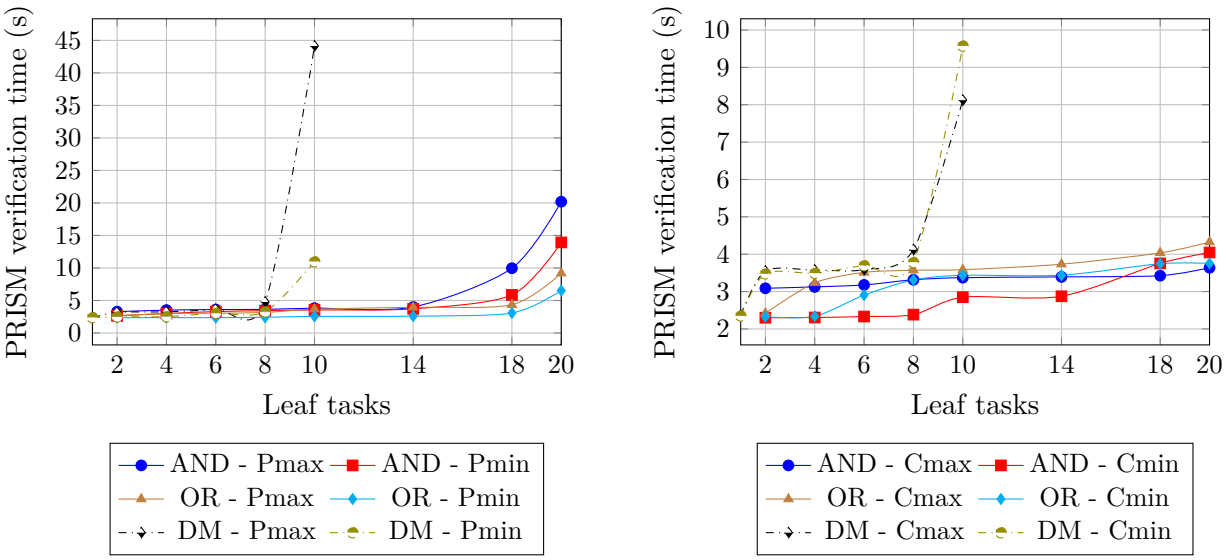
model file of size 18 KB, with the dimension of 10^8 states, transitions and different choices of paths to achieve the root goal *"G1: Emergency is detected"*. As for TAS, its CGM has 12 leaf tasks refined from different parent nodes (see Figure 4.1), which generates a MDP model file size of 12 KB, with the dimension of 10^6 states and 10^7 transitions to achieve the root goal *"G1: Provide health support"*. Therefore, despite the exponential growth that our new DM-annotation shows regarding the MDP model size, the use of this annotation is still affordable for research prototypes inspired by real requirements since they will hardly require 2^{18} context combinations to fulfill some node (BSN, for example, only needs a DM-annotation with 5 leaf tasks, meaning 2^5 different context combinations to

satisfy a node).

As for the time consumed to model check the generated MDP, Figure 4.5 shows the design-time PRISM verification time for reliability and cost properties. To evaluate the reliability verification time using PRISM tool, we assume that every context condition is valued to 1, and a reliability of 99% for every leaf task, therefore evaluating scenarios in which the MDP models follow the space trend previously discussed. From Figure 4.5a, we can note that traditional probabilistic model checkers take minimum of 2s to verify even the smallest and simplest models (such as AND-decomposition with 2 leaf tasks). The reliability verification time grows exponentially for all type nodes. But due to the non-determinism modeled by DM-annotation (in which 2^n non-deterministic transitions are modeled, being n the number of leaf tasks), this node type reliability verification time grows faster that simply AND/OR-decomposition, reaching over 40s to model check 10 leaf tasks. To evaluate the cost verification time using PRISM tool, we assume that every context condition holds, but now the leaf tasks' reliability is 100% so that PRISM can evaluate the cost to a real value. In this case, leaf tasks always succeed and, therefore, the paths for PRISM to provide a full coverage of the model are always constant and related to the number of leaf tasks. Figure 4.5b shows the linear growth of cost verification time for AND/OR-decomposition. As for DM-annotation, the growth is exponential since non-deterministic transitions are intrinsic to this node type. The minimum time for cost verification is over 2s for any model. For the BSN, it takes over 6,25s to verify its reliability property through PRISM, and over 2,2s to verify its cost property. For TAS, the reliability and cost properties are evaluated in over than 21s, and 2,8s, respectively, through PRISM. For design-time analysis, these values may be acceptable, but for runtime analysis it is a long time for SAS to wait for a verification, specially safe-critical SAS.

Question 2: What is the time trend to generate the parametric formulae?

To generate the parametric formulae, we use Algorithm 1. The time trend, that is, the time required for the generation of the parametric formulae is presented in Fig. 4.6. Figure 4.6 shows the generation time of the formuale grows linearly for AND-decomposition and exponentially for the other types. This is due to the complexity of symbolic formulae for each node type, as shown in Table 3.3. Nonetheless, the parametric formulae generation is a design-time activity and it takes approximately 12 leaf tasks to build the formulae in over 1s, which is quite affordable. Moreover, for systems with multiple different node types and, naturally, DM-annotation of less sub-nodes, the formulae generation time tends to be even smaller. Take the BSN, for example, whose CGM has 17 leaf tasks, with AND-decomposition and DM-annotation (with 5 sub-nodes). Its reliability formulae generation time is, in average, of 0,14s (standard deviation of 0,060s); while the cost



(a) Reliability verification time.

(b) Cost verification time.

Figure 4.5: Design-time PRISM verification time (M1.4).

formulae generation time is of 0,17s, in average (standard deviation of 0,071s). As for TAS, whose CGM contains 12 leaf tasks and different node types, such AND/OR-decomposition and 4 nodes with DM-annotation (with, at most, 3 sub-nodes), its reliability formula average generation time is 0,070s (standard deviation of 0,027s), and its cost formula average generation time is 0,076s (standard deviation of 0,029s).

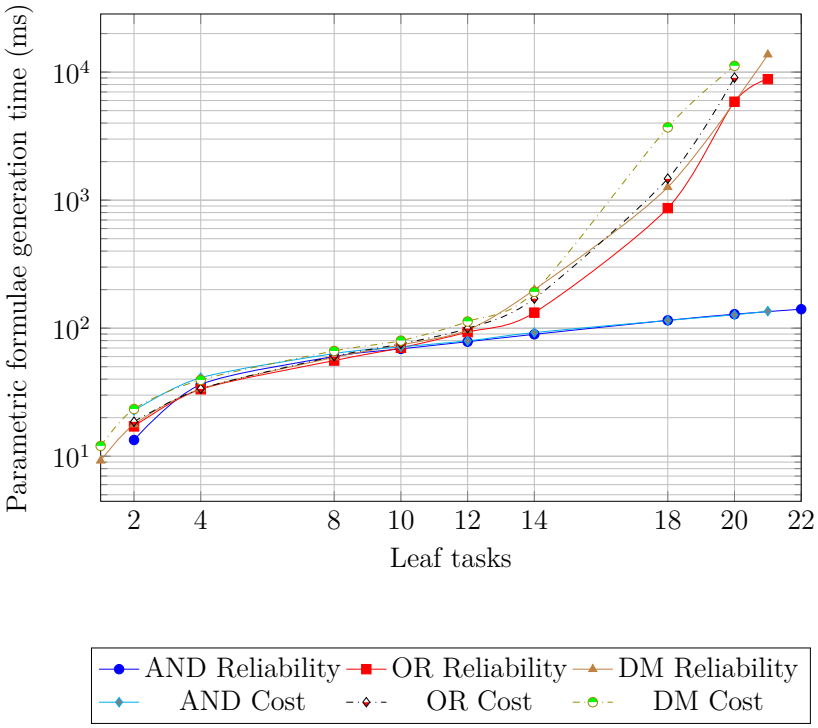


Figure 4.6: Generation time of the parametric formulae (M2.1 and M2.2).

Question 3: What is the time-space trend of a runtime verification using the formulae?

At runtime, our approach provides reliability and cost parametric formulae to assist the decision-making process of SAS. Table 4.1 details the number of parameters in the formulae for each node type (i.e., decomposition or annotation a node may have) with respect to the number of leaf tasks. The common parameters to every formula is the reliability associated with each leaf task. Cost formulae also contain the cost parameter, which refers to the cost of execution of each leaf task. Since a node with the DM-annotation requires context conditions associated with its sub-nodes, the formulae for this annotation also contains the leaf tasks' context parameters.

Leaf tasks	**DM-annotation**		**AND/OR-decomposition**	
	Reliability	**Cost**	**Reliability**	**Cost**
1	2	3	-	-
2	4	6	2	4
4	8	12	4	8
6	12	18	6	12
8	16	24	8	16
10	20	30	10	20
12	24	36	12	24
14	28	42	14	28
16	32	48	16	32
18	36	54	18	36
20	40	60	20	40
21	42	-	21	42
22	44	-	22	44/-

Table 4.1: Number of parameters in the formulae (M3.1).

The formulae sizes with respect to each node type and the number of leaf tasks are shown in Figure 4.7. Figure 4.7 shows that the size of reliability and cost formulae grows exponentially with the number of leaf tasks, with the exception of leaf tasks refined from AND-decomposition. This is due to the complexity of the symbolic formulae for each node type, as presented in Table 3.3. Because of the exponential growth that DM-annotation and OR-decomposition present, our approach faces memory challenges when generating reliability formulae for nodes with more than 22 leaf tasks refined from either DM-annotation or simply OR-decomposition. To generate cost formulae, the refinement limit is 20 leaf tasks for DM-annotation and 21 leaf tasks for OR-decomposition. As for AND-decomposition, we are able to generate reliability and cost formulae for over 1000 leaf tasks.

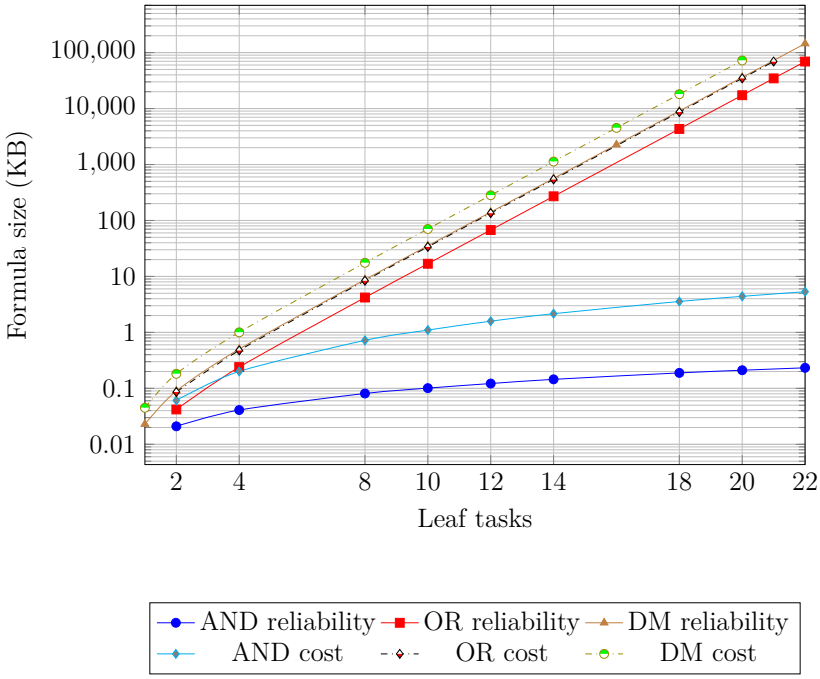


Figure 4.7: Size of the parametric formulae (M3.1).

As for the time consumed to evaluate our generated formulae at runtime, Figure 4.8 shows that AND-decomposition presents linear complexity with respect to the number of leaf tasks, remaining under 0.02s with 22 leaf tasks. DM-annotation and OR-decomposition present exponential complexity with a peak growth when reaching 12 leaf tasks. However, it is not until over 17 leaf tasks that their cost formulae take 1s for evaluation at runtime. On the other hand, by using traditional probabilistic model checking techniques at runtime, the smallest and simplest models already take over than 2s to be evaluated (see Figure 4.6). And, even though PRISM presents better evaluation time for cost formula than us, it requires that reliability parameters are evaluate to 100%, otherwise PRISM considers the cost as *"infinity"* [42]. In contrast, our formula is able to evaluate SAS cost for all value reliability parameters may assume during execution. Additionally, considering real systems like our running examples: (i) the BSN reliability (with 24 parameters and 3,4 KB) and cost (with 41 parameters and 15,6 KB) formulae are evaluated in 0.010s and 0.012s, respectively, at runtime; and (ii) the TAS reliability (with 24 parameters and 1,5 KB) and cost (with 36 parameters and 4,8 KB) formulae are evaluated in 0.012s and 0.013s, respectively, at runtime, we can conclude that our approach is efficient to optimize symbolic model checking of SAS with multiple sources of uncertainty parameterized, assuming a reasonable number of parameters due to space restrictions. We acknowledge, though, that optimization must be in place so formulae of larger system can be evaluated at runtime.

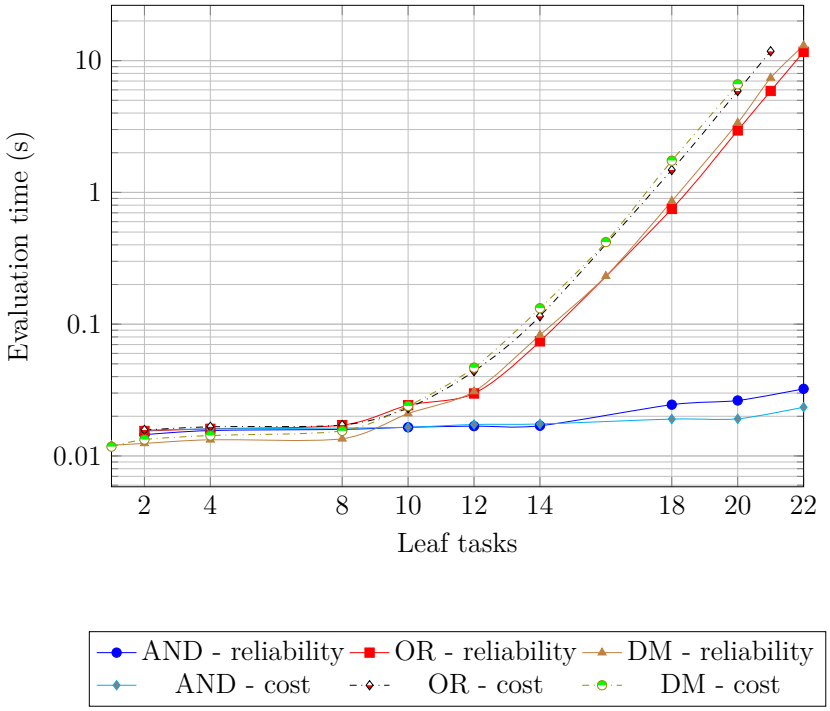


Figure 4.8: Evaluation time of the parametric formulae (M3.2 and M3.3).

Discussion

The space scalability limits of our approach for generating MDP PRISM models containing DM-annotated nodes and for generating DM-annotation and OR-decomposition formulae are relatively low. Therefore, we investigated the use of Reduced Ordered Binary Decision Diagrams (ROBDD) [43] to reduce the size of such models based on the logical relationship between context conditions in the CGM. The main idea is divided in three steps: (i) to observe how context conditions in the CGM interact logically, (ii) to build a ROBDD considering such interactions, and (iii) to retrieve the sets of context combinations that requires the generation of different verifiable models. In this sense, we could generate multiple but smaller models and, hence, we would be able to expand our scalability limits.

For instance, suppose we have a CGM with the following logical structure of contexts: $(c_1 \vee c_2 \vee c_3) \wedge c_4$. These four contexts can generate up to 2^4 combinations of context conditions, but we can reduce this number to the satisfying assignments in Table 4.2. These are the combinations that would satisfy the CGM. Hence, our approach would generate not one set of reliability and cost parametric formulae (that encompass all possible context combination), but actually seven different sets of parametric formulae (each one corresponding to a different conjecture of contexts). Most of these parametric formulae contain a smaller number of parameters since the contexts that do not hold (valued to 0) indicates that the leaf tasks associated to them do no execute, hence, their parameters can

be removed from the parametric formulae. A smaller number of parameters indicate these formulae are also smaller in size, in comparison with the global formulae our approach current generates. The context conjecture [0,0,1,1], for instance, contains only parameters of the leaf tasks related to contexts c_3 and c_4.

Table 4.2: Satisfying Context Combinations.

C1	**C2**	**C3**	**C4**
0	0	1	1
0	1	1	1
1	1	1	1
0	1	0	1
1	0	1	1
1	1	0	1
1	0	0	1

Although its usefulness on synthesizing valid context combinations to the satisficement of a CGM, the ROBDD approach do not improve our current scalability issues. This is due to the fact that, in the worst-case scenario (i.e., the conjecture in which all context conditions are holding) the parametric formulae generated is the same as the current global formulae our approach already generates. In other words, the approach would still need to generate these models with jeopardized scalability. Also, it is important to note that even though this worst-case scenario formulae may not be essential to the CGM's root goal fulfillment (i.e., other simple conjectures could satisfy while using smaller formulae), each conjecture in Table 4.2 may impact the CGM reliability and cost differently. Therefore the option to chose such context combination should be available for self-adaptation purposes.

Continuing the discussion, we should note that, a main aspect that interferes on the scalability limits verified by our evaluation is the the modeling structure adopted to create the generic CGMs analyzed. As presented in Figure 4.2, we increase the number of leaf tasks in a CGM but always associating them with a single node parent. However, such modeling structure is not common in real systems. In fact, systems are mostly composed by the combination of different goals/tasks and modeling types (e.g., AND/OR-decomposition, DM-annotation) rather then a single goal/task with multiple sub-nodes decomposed by only one node type.

BSN, for example, contains 17 leaf tasks and presents both AND-decomposition and DM-annotation (see Figure 2.2). The generated reliability and cost formulae for BSN have, respectively, 24 and 41 parameters. The reliability formula is 3,4 KB and the cost formula is 15,6 KB. Our other running example, TAS, contains 12 leaf tasks and presents AND/OR-decomposition and DM-annotation (see Figure 4.1). TAS generated

reliability formula has 24 parameters and 1,5 KB, while the generated cost formula has 36 parameters and 4,8 KB. Therefore, our approach manages to be quite affordable when fragmenting large DM-annotation constructs into smaller ones. Take, for example, the augmented CGM presented in Figure 4.9. This CGM contains 40 leaf tasks and another 40 context conditions, but instead of being modeled with a single node with the DM-annotation (with 40 sub-nodes), it has 4 DM-annotated nodes (with 10 sub-nodes each) that are united by AND-decomposition. Reliability formula for this CGM contains 80 parameters, while cost formula contains 120 parameters. Yet, formulae sizes are of 153 KB for reliability and 920 KB for cost, which are over 100 times smaller than the formulae sizes of our maximum limit model (DM-annotated model with 22 leaf tasks for reliability, and DM-annotated model with 20 leaf tasks for cost), as presented in Table 4.1. In this sense, we can conclude that our approach can be quite efficient for the modeling of real systems with multiple nodes and different modeling types.

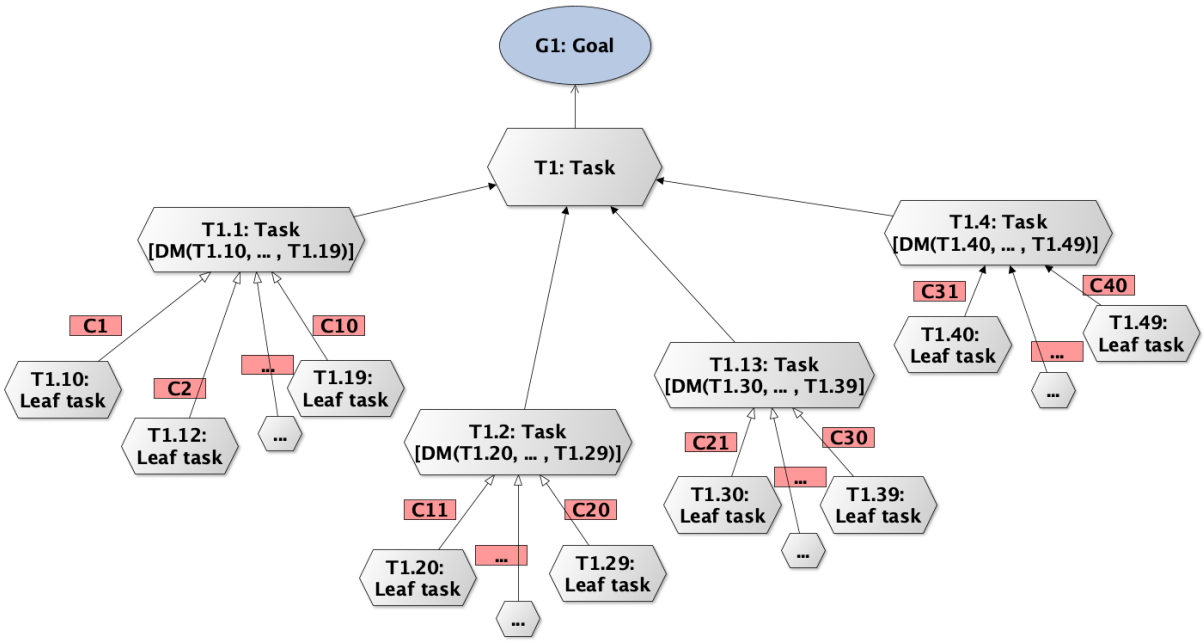


Figure 4.9: CGM Example with Fragmented DM-Annotation.

4.3 Parametric Formulae Analysis

Our second evaluation goal **EG2** aims to analyze whether or not the runtime formulae correctly represent the modeled SAS reliability and cost properties. As such, we perform a number of experiments in both TAS exemplar and BSN prototype, retrieving their leaf tasks' and the overall system's failure rates and execution costs. Then, we map the retrieved values into the formulae and compare whether or not our generated formulae

are able to similarly evaluate the systems' reliability and cost. The following question and metrics have been defined:

- **Q1**: Do the reliability and cost values provided by our runtime formulae accurately compute the values provided by experimenting on TAS exemplar?
  - M1.1: Wilcoxon statistical test.
- **Q2**: Do the reliability and cost values provided by our runtime formulae accurately compute the values provided by experimenting on the BSN prototype?
  - M2.1: Wilcoxon statistical test.

Q1: Do the reliability and cost values provided by our runtime formulae accurately compute the values provided by experimenting on TAS exemplar?

To evaluate the automatically generated reliability and cost formula for TAS, we performed 10 experiments in the TAS exemplar provided by Weyns and Calinescu [12]. The experiments had the following settings: (i) goal "G2: Provide automated life support" is the one chosen to fulfill goal "G1: Provide health support" (i.e., "Panic_button_triggered = false", hence C1 is 0 (false)); (ii) context C2 ("Patient_OK = false") holds at all times; and (iii) the *"no adaptation"* strategy is always used, i.e., the same medical and alarm services are chosen whenever a decision is made (such that, in Figure 4.1, C3, C5, C6 and C8 are 0 (false), and C4 and C7 are 1 (true) at all times). The values obtained in each experiment were mapped into our generated formula to compute reliability and cost.

Reliability Evaluation: Table 4.3 presents reliability for TAS' leaf tasks and TAS overall system obtained through experimenting, and the overall reliability provided by our formula.

Given the overall reliability values provided by both TAS exemplar and our generated formula, we used the paired Wilcoxon statistical test [44] to verify whether or not these methods' results follow a similar distribution. The null hypothesis (H_0) is that there is no difference between the distribution of TAS' reliability evaluation and our formula's. The alternative hypothesis is that there is a difference between these two methods. We consider a 5% of significance level. Therefore, the Wilcoxon test will calculate a p-value so that, if the p-value is greater than the significance level, we accept the null hypothesis; otherwise we reject the null hypothesis. To calculate the p-value of Wilcoxon test, we use the following R script:

```
x<-c(.92,.95,.89,.84,.94,.97,.92,.89,.94,.94)
y<-c(.92,.93,.87,.80,.94,.98,.90,.90,.92,.93)
```

Table 4.3: Reliability Formula Evaluation - TAS.

Experiment	Get Vital Params	Medical Service	Drug Service	Alarm Service	TAS	GODA-MDP
1	1.0	0.92	1.0	1.0	0.92	0.92
2	1.0	0.96	1.0	0.971	0.95	0.93
3	1.0	0.92	0.973	0.947	0.89	0.87
4	1.0	0.88	0.981	0.914	0.84	0.80
5	1.0	0.96	0.982	0.975	0.94	0.94
6	1.0	0.98	0.984	1.0	0.97	0.98
7	1.0	0.94	0.985	0.962	0.92	0.90
8	1.0	0.9	0.983	1.0	0.89	0.90
9	1.0	0.95	1.0	0.966	0.94	0.92
10	1.0	0.96	0.984	0.971	0.94	0.93

```
3 wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.1: R Script of Wilcoxon Test for TAS and GODA-MDP Reliability Formula Evaluation.

where x contains the reliability values obtained through the TAS exemplar experimentation, and y contains the reliability values obtained through exercising our generated reliability formula (Table 4.3). In case the p-value calculated is greater than 0.05, which is our significance level, then there is no reason to reject H_0. Otherwise, the null hypothesis is rejected.

From the script in Listing 4.1, we get that $p\text{-}value = 0.07895$. Since the p-value calculated is greater than our significance level, i.e., $p\text{-}value > 0.05$, we have no reason to reject $H0$. Therefore, we may say that our generated reliability formula evaluates reliability similar to TAS exemplar.

Cost Evaluation: As for the cost evaluation, Table 4.4 presents the individual cost for each leaf task in TAS, as well as the system overall cost given by TAS experimentation and by our generated cost formula. Since our cost formula is in function of reliability parameters as well, we used the reliability presented in Table 4.3 corresponding to each experiment. Leaf tasks' costs were obtained by getting the average cost of each of them and multiplying by their frequency of invocations, so that we have the actual cost that sums up at each of their execution.

By using the Wilcoxon test, we define the null hypothesis (H_0) as that there is no difference between evaluating the system overall cost through TAS experimentation or through our generated cost formula. The alternative hypothesis is that there is a difference between these two methods. We keep considering a 5% of significance level. The script

Table 4.4: Cost Formula Evaluation - TAS.

Experiment	Get Vital Params	Medical Service	Drug Service	Alarm Service	TAS	GODA-MDP
1	0	7.0	1.66	2.02	10.68	9.31
2	0	7.04	1.60	2.14	10.77	9.63
3	0	7.12	1.96	1.19	10.27	8.29
4	0	7.16	1.51	2.23	10.91	7.69
5	0	7.07	1.45	2.46	10.98	9.88
6	0	7.04	1.56	2.22	10.82	10.49
7	0	7.08	1.80	1.61	10.49	8.95
8	0	7.04	1.62	2.08	10.74	9.05
9	0	7.04	1.75	1.78	10.56	9.21
10	0	7.07	1.61	2.08	10.76	9.63

shown is Listing 4.2 was used to calculate the p-value. In this case, we have that $p\text{-}value = 0.001953$. Since the p-value calculated is smaller than the significance level (i.e., $p\text{-}value < 0.05$) we have to reject the null hypothesis. In this sense, we conclude that our generated cost formula evaluates cost differently from TAS exemplar.

```
x<-c(10.68,10.77,10.27,10.91,10.98,10.82,10.49,10.74,10.56,10.76)
y<-c(9.31,9.63,8.29,7.69,9.88,10.49,8.95,9.05,9.21,9.63)
wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.2: R Script of Wilcoxon Test for TAS and GODA-MDP Cost Formula Evaluation.

Discussion: In our approach, we generate both reliability and cost parametric formulae based on the parametric model checker PARAM, which is a well established tool in the literature. However, from the paired Wilcoxon statistical test, we observe discrepancies between TAS exemplar and our formula regarding the cost evaluation of the SAS (which does not occur when evaluating the reliability formula). In this sense, we compare the cost evaluation of TAS with ePMC, an alternative and recent efficient parametric model checking technique provided by Calinescu *et al.* [15].

In ePMC, the cost symbolic formula for a node with an OR-decomposition is defined as:

$$W_{OR}(n_1, ..., n_k) = W_{n_1} + (1 - P_{n_1}) * W_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - P_{n_i})\right) * W_{n_k}, \tag{4.1}$$

where $n_1, ..., n_k, k \in \mathbb{N}$, are sub-trees of the node in question, W_n is the cost of a sub-tree n and P_n is the reliability of a sub-tree n. Since our reliability formula evaluates reliability similar to the TAS exemplar, P_n is generated by our approach. Analogously, we define the cost symbolic formula for a node with an AND-decomposition or a DM-annotation as:

$$W_{AND}(n_1, ..., n_k) = W_{n_1} + P_{n_1} * W_{n_2} + ... + \left(\prod_{i=1}^{k-1} P_{n_i}\right) * W_{n_k}, \tag{4.2}$$

$$W_{DM}(n_1, ..., n_k) = C_{n_1}W_{n_1} + (1 - C_{n_1}P_{n_1}) * C_{n_2}W_{n_2} + ... + \left(\prod_{i=1}^{k-1}(1 - C_{n_i}P_{n_i})\right) * C_{n_k}W_{n_k}. \tag{4.3}$$

The cost formula for a node with the *incompleteness* node type remains the same presented in Table 3.3. Using the ePMC-based cost formula generated for TAS' CGM (see Figure 4.1), we perform the Wilcoxon test to compare the cost evaluation of TAS exemplar and the results provided by the new ePMC cost formula. Listing 4.3 presents the R Script used. The null hypothesis (H_0) is that these two methods are similar in evaluating TAS cost, while the alternative hypothesis is that they are different. Table 4.5 presents the cost value for TAS' leaf tasks and TAS overall system (computed by both TAS exemplar and the ePMC-based cost formula). From these data, we have a $p\text{-}value = 0.001953$. Considering a 5% of significance level, we have that the p-value calculated is also smaller than the significance level ($p\text{-}value < 0.05$). Then, we should reject the null hypothesis and conclude that the new ePMC-based cost formula does not evaluate cost similar to the TAS exemplar.

```
x<-c(10.68,10.77,10.27,10.91,10.98,10.82,10.49,10.74,10.56,10.76)
y<-c(10.39,10.63,10.07,10.48,10.85,10.77,10.31,10.39,10.39,10.64)
wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.3: R Script of Wilcoxon Test for TAS and ePMC Cost Formula Evaluation.

Considering an error perspective, i.e., a percentage difference between the cost values calculate by TAS exemplar and other approach, we can verify whether such other approach can be acceptable or not as similar to TAS when evaluating the cost property. We evaluate the percentage error using Equation 4.4.

$$Error(\%) = \left(\frac{Value_{AP1} - Value_{AP2}}{Value_{AP1}}\right) * 100 \tag{4.4}$$

where $Value_{AP1}$ is the cost evaluated by TAS exemplar and $Value_{AP2}$ is the cost evaluated by a different approach (such as GODA-MDP or ePMC) in a given experiment.

Table 4.5: ePMC Cost Formula Evaluation - TAS.

Experiment	Get Vital Params	Medical Service	Drug Service	Alarm Service	TAS	ePMC Formula
1	0	7.0	1.66	2.02	10.68	10.39
2	0	7.04	1.60	2.14	10.77	10.63
3	0	7.12	1.96	1.19	10.27	10.07
4	0	7.16	1.51	2.23	10.91	10.48
5	0	7.07	1.45	2.46	10.98	10.85
6	0	7.04	1.56	2.22	10.82	10.77
7	0	7.08	1.80	1.61	10.49	10.31
8	0	7.04	1.62	2.08	10.74	10.39
9	0	7.04	1.75	1.78	10.56	10.39
10	0	7.07	1.61	2.08	10.76	10.64

Considering the experimental data presented in Tables 4.4 and 4.5, we have that our approach presents error percentages up to 30% when evaluating TAS cost, while the ePMC based cost formula presents errors up to 4%. Hence, we acknowledge that the cost parametric formula generated by our approach, which is based on the PARAM model checker, requires further validation. But the cost formulae based on ePMC, even though failed the Wilcoxon test, presents reasonable error percentage. Next, we validate both reliability and cost formulae for BSN to analyze the behavior of such models.

Q2: Do the reliability and cost values provided by our runtime formulae accurately compute the values provided by experimenting on the BSN prototype?

Similar to TAS evaluation, to evaluate the automatically generated reliability and cost formula for BSN we performed 10 experiments in a BSN prototype [39]. The experiments had the following settings: the available sensors are the electrocardiograph sensor, the pulse oximeter, the thermometer and the sphygmomanometer (i.e., contexts C1 to C4 are evaluated to 1 (true) at all times) Context C5 never holds, and context C6 always holds. The values obtained in each experiment were mapped into our generated formula to compute reliability and cost.

Reliability Evaluation: Table 4.6 presents reliability for BSN's leaf tasks and overall system obtained through experimenting, and the overall reliability provided by our formula. For the sake of space in the table, we do not present the reliability of leaf tasks "T1.1: Fuse sensors data", "T1.2: Detect patient status", and "T1.3: Persist data" in Table 4.6, but, instead, we present their parent node "T1: Analyze vital signs" reliability.

Table 4.6: Reliability Formula Evaluation - BSN.

Experiment	Collect SaO2 data			Collect ECG data			Collect TEMP data			Collect ABP data			Analyze	BSN	GODA
	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	vital signs		MDP
1	0.891	0.885	0.797	0.906	0.875	0.833	0.896	0.865	0.839	0.88	0.89	0.839	0.885	0.88	0.872
2	0.895	0.875	0.825	0.89	0.875	0.835	0.895	0.87	0.785	0.8	0.915	0.84	0.88	0.86	0.864
3	0.923	0.886	0.813	0.912	0.898	0.82	0.927	0.879	0.865	0.788	0.897	0.821	0.901	0.89	0.889
4	0.901	0.894	0.790	0.914	0.894	0.804	0.904	0.866	0.801	0.863	0.883	0.89	0.911	0.9	0.897
5	0.914	0.884	0.814	0.911	0.911	0.804	0.917	0.899	0.78	0.856	0.862	0.859	0.899	0.887	0.886
6	0.91	0.877	0.823	0.918	0.905	0.823	0.914	0.877	0.848	0.889	0.897	0.852	0.868	0.86	0.859
7	0.922	0.895	0.753	0.949	0.919	0.791	0.912	0.892	0.743	0.821	0.949	0.831	0.902	0.878	0.887
8	0.954	0.954	0.784	0.945	0.95	0.807	0.945	0.908	0.899	0.867	0.95	0.798	0.89	0.876	0.884
9	0.946	0.889	0.85	0.95	0.942	0.792	0.95	0.904	0.881	0.815	0.992	0.615	0.876	0.869	0.868
10	0.949	0.897	0.817	0.949	0.938	0.846	0.930	0.899	0.81	0.894	0.956	0.759	0.927	0.916	0.919

Given the overall reliability values provided by both BSN prototype and our generated formula, we used the paired Wilcoxon statistical test to verify whether or not these methods provide similar results. The null hypothesis (H_0) is that there is no difference between the reliability evaluation performed by BSN and by our generated formula. The alternative hypothesis is that there is a difference between these two methods. Once again, we consider a 5% of significance level. Listing 4.4 presents the R script to calculate the p-value, where x contains the reliability values obtained through BSN simulation and y contains the reliability values obtained by our generated reliability formula (see Table 4.6).

```
x<-c(.88,.86,.89,.9,.887,.86,.878,.876,.869,.916)
y<-c(.872,.864,.889,.897,.886,.859,.887,.884,.868,.919)
wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.4: R Script of Wilcoxon Test for BSN and GODA-MDP Reliability Formula Evaluation.

From the script in Listing 4.4 we get that $p\text{-}value = 0.7579$. Since the p-value calculated is greater than the significance level, i.e., $p\text{-}value > 0.05$, we have no reason to reject $H0$. Therefore, we can say that our generated reliability formula evaluates reliability similar to the BSN prototype.

Cost Evaluation: As for the cost evaluation, Table 4.7 presents the individual cost for each leaf task in BSN, as well as the system overall cost given by BSN experimentation and by our generated cost formula. To evaluate our cost formula, we used the reliability parameters presented in Table 4.6 corresponding to each experiment. Leaf tasks' costs were obtained by getting the total cost of each of them during an experiment, and multiplying by their frequency of invocations, so that we have the actual cost that sums up at each of their execution.

Using the Wilcoxon test, the null hypothesis (H_0) is that there is no difference between evaluating the system overall cost through BSN experimentation or through our generated cost formula; and the alternative hypothesis is that there is a difference between these two methods. We keep considering a 5% of significance level. Listing 4.5 presents the R script to calculate the p-value. In this case, we have that $p\text{-}value = 0.001953$. Since the p-value calculated is smaller than the significance level (i.e., $p\text{-}value < 0.05$) we have to reject the null hypothesis. In this sense, we conclude that our generated cost formula evaluates cost differently from BSN prototype.

Table 4.7: Cost Formula Evaluation - BSN.

Experiment	Collect SaO2 data			Collect ECG data			Collect TEMP data			Collect ABP data			Analyze	BSN	GODA
	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	vital signs		MDP
1	19.2	15.23	12.35	19.2	15.77	12.68	19.2	15.41	11.72	19.2	14.88	11.56	19.2	81.84	95.63
2	20	16.02	12.64	20	16.02	12.48	20	16.02	12.48	20	12.8	10.66	20	81.2	96.47
3	27.3	23.26	18.05	27.3	22.71	18.54	27.3	23.45	18.71	27.3	16.93	13.79	27.3	113.65	148.1
4	29.1	23.77	18.98	29.1	24.31	19.79	29.1	23.77	18.5	29.1	21.65	16.94	29.1	121.43	149.92
5	32.7	27.34	22.13	32.7	27.16	22.29	32.7	27.52	22.46	32.7	23.98	17.76	32.7	134.99	171.47
6	24.3	20.1	16.13	24.3	20.46	17.13	24.3	20.28	15.65	24.3	19.2	15.81	24.3	103.15	126.05
7	29.6	25.18	20.78	29.6	26.68	23.37	29.6	24.63	20.11	29.6	19.95	17.87	29.6	123.62	155.62
8	21.8	19.85	17.98	21.8	19.47	17.44	21.8	19.47	16.04	21.8	16.39	14.86	21.8	97.47	133.32
9	26	23.28	18.62	26	23.47	21.06	26	23.47	19.3	26	17.29	17.12	26	123.41	152.17
10	27.3	24.57	20.4	27.3	24.57	21.81	27.3	23.63	19.72	27.3	21.81	20.06	27.3	119.61	165.43

```
x<-c(81.84,81.2,113.65,121.43,134.99,103.15,123.62,97.42,123.41,1
   19.61)
y<-c(95.63,96.47,148.1,149.92,171.47,126.05,155.62,133.32,152.17,
   165.43)
wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.5: R Script of Wilcoxon Test for BSN and GODA-MDP Cost Formula Evaluation.

Discussion: From the paired Wilcoxon statistical test, we observe discrepancies between evaluating BSN cost through experimenting the prototype and calculating using our generated cost parametric formula. Such discrepancies do not occur when evaluating BSN reliability property. Therefore, we also compare the cost evaluation of BSN with the ePMC technique. Based on Equations 4.1, 4.2 and 4.3, we generate the ePMC cost formula for BSN's CGM and performed the Wilcoxon test to compare the cost evaluation of BSN prototype simulation and the results provided by the new ePMC cost formula. The null hypothesis (H_0) is that these two methods are similar in evaluating BSN cost, while the alternative hypothesis is that they are different. We consider a 5% of significance level. Table 4.8 presents the cost value for BSN's leaf tasks and BSN overall system (computed by both BSN prototype simulation and the ePMC-based cost formula). Listing 4.6 contains the script used to calculate the p-value. From this script, we have a $p\text{-}value = 0.03711$. Since the p-value calculated is smaller than the significance level ($p\text{-}value < 0.05$), we reject the null hypothesis and conclude that the new ePMC-based cost formula does not evaluate cost similar to the BSN prototype.

```
x<-c(81.84,81.2,113.65,121.43,134.99,103.15,123.62,97.42,123.41,1
   19.61)
y<-c(84.73,86.72,120.31,129.88,145.22,107.08,138.36,99.86,115.05,
   122.89)
wilcox.test(x,y,paired=T,correct=T)
```

Listing 4.6: R Script of Wilcoxon Test for BSN and ePMC Cost Formula Evaluation.

Once again, we evaluate the error percentage between the cost values obtained through simulating BSN and through other methods. We use Equation 4.4, where $Value_{AP1}$ is the cost value obtained from BSN, and $Value_{AP2}$ is the cost value obtained through either GODA-MDP or ePMC, in a given experiment. Considering the data in Tables 4.7 and 4.8, we have that our generated cost formula presents error percentages up to 38% when evaluating BSN cost, while the ePMC based cost formula presents errors up to 12%. From theses results, we can conclude that although our approach, which is based on the PARAM model checker, generates trustworthy reliability parametric formulae for both

TAS and BSN, our generated cost parametric formulae could not appropriately evaluate these systems cost. As for the alternative ePMC technique, it showed itself more accurate in evaluating both TAS and BSN overall cost. Even though failing the Wilcoxon statistical test, the formulae generated based on ePMC presented relatively small and acceptable error percentages.

Table 4.8: ePMC Cost Formula Evaluation - BSN.

Experiment	Collect SaO2 data			Collect ECG data			Collect TEMP data			Collect ABP data			Analyze	BSN	GODA
	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	Read	Filter	Transfer	vital signs		MDP
1	19.2	15.23	12.35	19.2	15.77	12.68	19.2	15.41	11.72	19.2	14.88	11.56	19.2	81.84	84.73
2	20	16.02	12.64	20	16.02	12.48	20	16.02	12.48	20	12.8	10.66	20	81.2	86.72
3	27.3	23.26	18.05	27.3	22.71	18.54	27.3	23.45	18.71	27.3	16.93	13.79	27.3	113.65	120.31
4	29.1	23.77	18.98	29.1	24.31	19.79	29.1	23.77	18.5	29.1	21.65	16.94	29.1	121.43	129.88
5	32.7	27.34	22.13	32.7	27.16	22.29	32.7	27.52	22.46	32.7	23.98	17.76	32.7	134.99	145.22
6	24.3	20.1	16.13	24.3	20.46	17.13	24.3	20.28	15.65	24.3	19.2	15.81	24.3	103.15	107.08
7	29.6	25.18	20.78	29.6	26.68	23.37	29.6	24.63	20.11	29.6	19.95	17.87	29.6	123.62	138.36
8	21.8	19.85	17.98	21.8	19.47	17.44	21.8	19.47	16.04	21.8	16.39	14.86	21.8	97.47	99.86
9	26	23.28	18.62	26	23.47	21.06	26	23.47	19.3	26	17.29	17.12	26	123.41	115.05
10	27.3	24.57	20.4	27.3	24.57	21.81	27.3	23.63	19.72	27.3	21.81	20.06	27.3	119.61	122.09

4.4 Threats to validity

Construct validity. The major threats here are the mapping of leaf tasks, representing operations of the system, on their concrete implementation in the system; and the mapping of leaf tasks properties on the formulae parameters. We assume that leaf tasks can be traced back to software operations and that the reliability and cost of the former ones represents the probability of a successful execution of the later elements and the energy taken to successfully execute them. Moreover, our generated formulae contain parameters for each of the leaf tasks' reliability and cost properties. Another threat for construct validity is that we do not exercise all possible scenario combinations for TAS nor BSN. In TAS, the experimenting scenario in the exemplar is not modifiable, so we use the fixed scenario to conduct our evaluation. In BSN, by simulating a scenario in which all context conditions are holding except one, we are able to exercise the formulae in a next to worst-case scenario, in which almost all formulae parameters are evaluated, and also observe the formulae behavior with a not holding context condition.

Internal validity. The suitability of our approach to reason over TAS and BSN reliability and cost properties has been presented. We performed a number of simulations on our running examples to collect data regarding these properties. We used a non-parametric statistical method to compare independent groups (i.e., independent reliability/cost formula evaluation approaches) and, therefore, to be able to conclude whether or not these approaches are similar. Although the evaluations show consistent results, we notice individual discrepancies over cost evaluation, to which we used a different symbolic model checker (ePMC) to generate cost formulae for both TAS and BSN, and evaluate them.

External validity. We evaluate our approach on two research prototypes inspired by real self-adaptive system requirements: BSN and TAS, both belonging to the healthcare domain. Further evaluation must be performed not only to evaluate the applicability of our approach in SAS' prototypes of different domains, but also to evaluate real industrial-strength self-adaptive systems.

Reliability. Scalability assessment can be reproduced by generating the goal models (as specified in Section 4.2) on the piStarGODA-framework, and by using open source tools such as PRISM. The experimentation data regarding formulae evaluation is presented in Section 4.3, alongside the scripts used to evaluate the formulae. The formulae themselves, the goal models of the running examples, and the piStarGODA-framework used to generated all of them are publicly on GitHub[2].

[2] `https://github.com/lesunb/pistarGODA-MDP`

Chapter 5

Related Work

5.1 Modeling SAS under Uncertainty

Regarding the works that focus on modeling requirements of SAS under uncertainty, Whittle *et al.* [10] argue that requirements languages for SAS should include constructs for specifying and dealing with the uncertainty inherent in SAS. They present RELAX, a language that supports the expression of environmental uncertainty in system requirements, so that adaptation modules can reason about system satisfaction at runtime without jeopardizing critical requirements. Cheng *et al.* [45] introduce a variation of threat modeling to identify sources of uncertainty and demonstrate how RELAX can be used to specify more flexible requirements within a goal model to handle the uncertainty. Baresi and Ghezzi [46] presents FLAGS (Fuzzy Live Adaptive Goals for Self-adaptive systems), an innovative goal model augmented with adaptive goals, which define countermeasures that one must perform if one or more goals are not fulfilled satisfactorily. FLAGS also propose fuzzy constraints to specify the degree of a goal satisfaction. In our work, we also explore goal-oriented requirements engineering to model SAS. Additionally, we augment goal modeling with numerous sources of different classes of uncertainty, not limited to environmental uncertainty.

5.2 Supporting Self-Adaptation

Regarding the works that support self-adaptation through system verification, Filieri *et al.* [47] present a framework to overcome model checking scalability issues by generating runtime symbolic expressions of system requirements from rewarded Discrete-Time Markov Chains, while accounting for uncertainty in the managed system (such as failure probability and energy cost of an operation). At design time, Matrix-based methods are used to generate formulas representing the system properties of interest. At runtime, once

the real values of the variables are known, the formulas can be evaluated. The framework shifts the the model verification cost to design time and shows substantial improvements in terms of efficiency.

Cailliau and Lamsweerde [48] use goal models through an obstacle-driven runtime technique to support adaptation aiming at increasing the actual satisfaction rate of probabilistic system goals in spite of environment changes. They use probabilistic LTL_3 monitors to continuously evaluate goals satisfaction and find most appropriate countermeasures at runtime. Bencomo and Belaggoum [49] map goal models onto Dynamic Decision Networks (DDNs) to provide a principled approach to make rational decisions in the face of uncertainty within changing environments. The generated DDNs combine probability and utility theory in order to make a decision based on the realization strategies with the highest utilities.

Weyns and Iftikhar [50] propose a modular approach for decision making that supports changing goals at runtime. The approach requires distinct models capturing the essential elements of the managed system and its environment, and models capturing the characteristic of different qualities that are subject to adaptation. When an adaptation is in need at runtime, a pre-defined number of simulations over the models is performed and the best configuration is chosen to adapt the system. Authors acknowledge that, although simulation is inherently more efficient compared to exhaustive approaches, the consequence is a reduction of accuracy, which may lead to temporal violations of requirements.

In our work, we combine goal models and symbolic model checking to generate reliability and cost formulae parameterized with uncertainty. Similar to Filieri *et al.*[47], we use symbolic model checking to overcome model checking scalability issues at runtime. However, besides the different classes of uncertainty that are used as parameters in our parametric formulae, they are also able to encapsulate non-deterministic behavior of SAS. Moreover, our parametric formulae not only are used to verify system properties at runtime, thus supporting decision-making and adaptation in SAS, but also to guide the synthesis of adaptation policies, alongside the goal model, by SAS engineers.

5.3 Supporting the Synthesis of Adaptation Policies

Among the works that focus on synthesizing adaptation policies, Su *et al.* [51] use a parametric MDP to apply the value iteration method and select a confidently optimal adaptation policy at runtime with focus on a trade-off among accuracy, data usage, and computational overhead. Cámara *et al.* [52] and Moreno *et al.* [53] use the PRISM language [32] to model SAS as MDPs and synthesize adaptation policies. They both use

the probabilistic model checker PRISM [42], but while Cámara *et al.* [52] focus on a design-time approach to synthesize optimal repertories considering uncertainty, Moreno *et al.* [53] use PRISM at runtime to synthesize policies that maximize an expected accumulated utility over a finite horizon. In our work, we provide design-time generated parametric formulae with *multiple classes of uncertainty*, along with a goal model of the system, to support the synthesis of adaptation policies. In this sense, we avoid the model exploration that is required by traditional model checking techniques while supporting various types of uncertainty.

5.4 Supporting the Assurance Process of SAS

Regarding the support of SAS assurance process in an end-to-end methodology, Calinescu *et al.* [54] propose QoSMOS, a framework to develop service-based systems that achieve their Quality of Service (QoS) requirements through dynamically adapting to changes in the system state, environment, and workload. QoSMOS keeps an updated model of the system at runtime and monitors changes. The model (a Markov model) depends on parameters whose values can be unknown or imprecise at design time, and even change during the operating life of the system. In case an adaptation need is detected, QoSMOS performs a number of PRISM experiments to analyse system properties and to choose optimal values for the configurable parameters. Different from our work, QoSMOS uses probabilistic model checking at runtime, causing significant computational overhead. Therefore, the framework should be applied to systems where time efficiency is not a problem.

Filieri *et al.* [55] provide an approach for continuous verification of reliability and performance properties of evolving and adaptive software. The work focuses on environmental changes that may affect the quality of an application, and also non-functional requirements. Uncertainty is stated in probabilistic terms. The approach is based on formal probabilistic models. At design time, assumptions are made on the external environment, and probabilistic model checking is used in order to prove that the specified machine satisfies its performance and reliability requirements. At runtime, relevant environmental data is continuously collected and Bayesian inference technique is used to automatically update the system model. The updated model, which is kept alive at run time, is re-run to perform continuous verification.

Ghezzi *et al.*[56] present ADAM, a model-driven framework conceived to support the development and runtime operation of SAS, aiming at mitigating uncertainty concerning response time and faulty behavior. Initially, the framework automatically generates a MDP representation of the system from an Activity Diagram. At design time, it relies on

probabilistic model checking to compute the minimum and maximum cumulative rewards from each state of the MDP. At runtime, the framework executes the application by navigating through the MDP's state space and invoking the corresponding implementation. Probability Theory is used to choose the transition with greater probability of success, in case of non-deterministic outgoing transitions.

Mendonça *et al.* [3] propose GODA, a framework for goal-oriented dependability analysis. GODA provides a modeling environment to build goal models augmented with contextual and runtime annotation. The former annotation is related to different environmental conditions the system faces at runtime, and the latter is related to the different behavior the system assumes during execution. From such goal model, GODA automatically generates a formal model for design-time verification, a Discrete-Time Markov Chain (DTMC), and a dependability parametric formula for runtime analysis. However, none of these generated models contain uncertainty parameters other then the reliability of leaf tasks in the goal model. In our work, we extend GODA in order to represent different classes of uncertainty not only in the goal model, but also to pervade this information into the verifiable models we generate. In this sense, the assurance process of the system, which is made through the verification of the generated models, becomes more accurate. Additionally, our work focuses on the reliability and cost properties for evaluation, which enables a trade-off in the decision making process for self-adaptation.

In [57], Calinescu *et al.* propose the ENTRUST methodology to provide assurance evidence, cases, and arguments for SAS at design- and runtime while supporting internal and environmental changes. ENTRUST combines design-time modeling and verification, and industry-adopted assurance processes. At design time, due to environmental and internal uncertainty, incomplete models for both the managing and managed system are developed. At runtime, the implanted system dynamically adjusts its parameters and architecture in face of internal or environmental changes. Every time a system reconfiguration is produced, probabilistic model checking analyses the compliance of the self-adaptive system with its runtime-assured requirement.

In our work, we exploit GORE into the modeling of SAS under uncertainty. We go further on supporting different classes of uncertainty, since we support three classes with focus on five sources of uncertainty: (i) future parameter value, and (ii) incompleteness, regarding uncertainty in the system itself; (iii) specification of goals, regarding uncertainty in system goals; (iv) execution context, and (v) noise in sensing, regarding environmental uncertainty. Moreover, we advocate the use of symbolic model checking as means to continuously verify the model at runtime with minimal overhead.

5.5 Final Considerations About the Related Work

In this chapter we listed some related work on self-adaptive system, specifying the classes of uncertainty it supports, in which state of the system life cycle they focus, and the modeling and verification techniques used to assist the system adaptation process. Table 5.1 summarizes these properties and characteristics of each related work and what they differ from our work.

Table 5.1: Comparative Table of Related Work.

Work by:	**Class of uncertainty**	**System stage**	**Techniques applied**	**Goal-oriented**
Filieri *et al.*, 2016 [47]	System itself	Runtime	Symbolic model checking	No
Cailliau and Lamsweerde, 2017 [48]	Environment	Runtime	LTL_3 monitoring technique	Yes
Bencomo and Belaggoum, 2013 [49]	Environment	Design-time Runtime	Dynamic Decision Network (DDN) evaluation	Yes
Weyns and Iftikhar, 2016 [50]	System goals	Runtime	Simulation	No
Su *et al.*, 2016 [51]	Not specified	Runtime	Value iteration method	No
Cámara *et al.*, 2018 [52]	System itself, Environment	Design-time	Probabilistic model checking	No
Moreno *et al.*, 2015 [53]	Environment	Runtime	Probabilistic model checking	No
Filieri *et al.*, 2012 [55]	Environment	Design-time Runtime	Probabilistic model checking, Bayesian inference	No
Ghezzi *et al.*, 2013 [56]	System itself	Design-time Runtime	Probability theory	No
Mendonça *et al.*, 2016 [3]	Environment (design-time only), System itself	Design-time Runtime	Symbolic model checking	Yes
Calinescu *et al.*, 2011,2017 [54, 57]	System itself, Environment	Design-time Runtime	Probabilistic model checking	No
Present work	**System itself, System goals, Environment**	**Design-time Runtime**	**Symbolic model checking**	**Yes**

Chapter 6

Conclusion and Future Work

In this work, we presented an approach to support the assurance process for trustworthy SAS that operates under different classes of uncertainty. Uncertainty is a first-class concept in SAS since it can influence the system towards an unforeseen behavior. Thus, assuring that SAS' goals are continuously satisfied *even in face of uncertainty* is a research challenge.

Based on Goal-Oriented Requirements Engineering (GORE), our approach supports the goal modeling of SAS augmented with uncertainty. From the resulting goal models, we generate MDP models in PRISM language for design time system verification and simulation. Also, we compositionally generate reliability and cost parametric formulae provided with parameterized uncertainties. These formulae are then suitable for guiding the synthesis of adaptation policies and, at runtime, for automatically evaluating the fulfillment of SAS goals taking different classes of uncertainty into account while supporting self-adaptation.

The scalability assessment shows that parametric model checking (i.e., evaluating our parametric formulae parameterized with multiple classes of uncertainty) is a better technique then traditional probabilistic model checking for runtime analysis due to time limitations that occur in this stage, assuming a reasonable number of parameters. However, our generated models present space limitation when considering specific node types, such as the DM-annotation. Such limitations can be overcome by combining different node types in a model, such as AND-decomposition of DM-annotated nodes with a smaller number of sub-nodes, which is the usual behavior of most real life systems. The formulae evaluation on TAS and BSN shows consistency while evaluating the reliability property but discrepancies were found while evaluating the cost property. Hence, we should validate the approach in different SAS and investigate new parametric model checking techniques that could present better results, such as ePMC. Nonetheless, we can conclude that our empirical assessment on two SAS exemplars has presented itself effective in supporting

the assurance process of SAS while taking uncertainty into account.

Some improvements for this proposal and potential new contributions regarding the applicability of GORE to assure the fulfillment of SAS goals are planned for future work, more specifically:

- **Improve scalability of generated models:** study how we could increase the scalability limits for generating both MDP models in PRISM language and parametric formulae, especially regarding the use of DM-annotation.
- **Augment our current CGM with runtime annotations:** provide runtime information to be annotated onto our CGM with uncertainty. Runtime information regards the system execution behavior (such as parallel, alternative or sequential execution), and fault tolerant strategies (such as retry, try, and multiple cardinality strategies).
- **Generate parametric formulae based on different methodologies:** explore the new method of efficient parametric model checking (ePMC) [15] and define symbolic formulae for our current modeling types and possible new runtime annotations, so we can generate parametric formulae based on this approach (as well as based on PARAM tool).
- **Generate parametric formula for different metrics:** go further on providing verifiable models regarding only reliability and cost metrics. Explore the different properties the PCTL language allow specifying, compare them with the most relevant metrics for SAS self-adaptation and verification processes. Generate parametric formulae for these relevant metrics, such as expected time of execution.
- **Update the parametric formulae according to system evolution:** SAS may evolve and change its behavior during execution (e.g., changing, adding or removing goals), and a static generated parametric formula may not be suitable anymore. We should investigate how to update the parametric formulae, at runtime, accordingly.
- **Extend piStarGODA-MDP framework to support multiple agents:** SAS generally operates under distributed systems. In this sense, piStarGODA-MDP should support the modeling of multiple agents and the relationship between them to satisfy a main goal.
- **Evaluate the approach with multiple exemplars of SAS:** extend our evaluation to multiple SAS exemplars so we can assure that our approach is sound in providing trustworthy runtime models for SAS evaluation.

Reference List

[1] Weyns, Danny: *Software engineering of self-adaptive systems: an organised tour and future challenges*. Chapter in Handbook of Software Engineering, 2017. vii, 1, 2, 5, 6, 16, 17

[2] Forejt, Vojtěch, Marta Kwiatkowska, Gethin Norman, and David Parker: *Automated verification techniques for probabilistic systems*. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 53–113. Springer, 2011. vii, 11

[3] 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. vii, 4, 7, 11, 12, 13, 14, 29, 67, 69

[4] Esfahani, Naeem and Sam Malek: *Uncertainty in self-adaptive software systems*. In *Software Engineering for Self-Adaptive Systems II*, pages 214–238. Springer, 2013. 1

[5] Mahdavi-Hezavehi, Sara, Paris Avgeriou, and Danny Weyns: *A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements*. In *Managing Trade-Offs in Adaptable Software Architectures*, pages 45–77. Elsevier, 2017. 1, 2, 5, 6, 15, 16, 17, 19

[6] Lemos, Rogério de, David Garlan, Carlo Ghezzi, Holger Giese, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Danny Weyns, Luciano Baresi, Nelly Bencomo, *et al.*: *Software engineering for self-adaptive systems: research challenges in the provision of assurances*. In *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 3–30. Springer, 2017. 1, 5

[7] Lamsweerde, A. van: *Goal-oriented requirements engineering: a guided tour*. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262, 2001. 2, 6, 7

[8] Ali, Raian, Fabiano Dalpiaz, and Paolo Giorgini: *A goal-based framework for contextual requirements modeling and analysis*. Requirements Engineering, 15(4):439–458, 2010. 2, 7, 8

[9] Mendonça, Danilo F., Raian Ali, and Genaína N. Rodrigues: *Modelling and analysing contextual failures for dependability requirements*. In *Proc. of the 9th Inter, Symp, on Soft. Eng. for Adaptive and Self-Managing Systems*, SEAMS 2014. ACM, 2014, ISBN 978-1-4503-2864-7. 2, 7

[10] Whittle, Jon, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean Michel Bruel: *Relax: Incorporating uncertainty into the specification of self-adaptive systems.* In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009. 2, 64

[11] Kwiatkowska, Marta, Gethin Norman, and David Parker: *Quantitative analysis with the probabilistic model checker prism.* Electronic Notes in Theoretical Computer Science, 153(2):5–31, 2006. 3, 24, 43

[12] Weyns, Danny and Radu Calinescu: *Tele assistance: a self-adaptive service-based system examplar.* In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 88–92. IEEE Press, 2015. 3, 40, 41, 52

[13] 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. 3, 8, 9, 40

[14] Hahn, Ernst Moritz, Holger Hermanns, Björn Wachter, and Lijun Zhang: *Param: A model checker for parametric markov models.* In *International Conference on Computer Aided Verification*, pages 660–664. Springer, 2010. 3, 29

[15] Calinescu, Radu, Colin Alexander Paterson, and Kenneth Johnson: *Efficient parametric model checking using domain knowledge.* IEEE Transactions on Software Engineering, 2019. 3, 54, 71

[16] Hellerstein, Joseph L, Yixin Diao, Sujay Parekh, and Dawn M Tilbury: *Feedback control of computing systems.* John Wiley & Sons, 2004. 5

[17] Weyns, Danny and Tanvir Ahmad: *Claims and evidence for architecture-based self-adaptation: a systematic literature review.* In *European Conference on Software Architecture*, pages 249–265. Springer, 2013. 5

[18] 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. 6, 16

[19] Dardenne, Anne, Axel van Lamsweerde, and Stephen Fickas: *Goal-directed requirements acquisition.* Science of Computer Programming, 20(1):3–50, 1993. 6, 7

[20] Yu, Eric: *Modelling strategic relationships for process reengineering.* Social Modeling for Requirements Engineering, 11:2011, 2011. 7

[21] Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos: *Tropos: An agent-oriented software development methodology.* Autonomous Agents and Multi-Agent Systems, 8(3):203–236, 2004. 7

[22] Baier, Christel and Joost Pieter Katoen: *Principles of model checking.* MIT press, 2008. 10, 11, 21, 29

[23] Kwiatkowska, Marta and David Parker: *Automated verification and strategy synthesis for probabilistic systems.* In *International Symposium on Automated Technology for Verification and Analysis*, pages 5–22. Springer, 2013. 10, 11, 29

[24] Kwiatkowska, M. and D. Parker: *Advances in probabilistic model checking.* In Nipkow, T., O. Grumberg, and B. Hauptmann (editors): *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 126–151. IOS Press, 2012. 10

[25] Ghezzi, Carlo and Amir Molzam Sharifloo: *Model-based verification of quantitative non-functional properties for software product lines.* Information and Software Technology, 55(3):508–524, 2013. 11

[26] Grunske, Lars: *An effective sequential statistical test for probabilistic monitoring.* Information and Software Technology, 53(3):190 – 199, 2011. 11, 24

[27] Daws, Conrado: *Symbolic and parametric model checking of discrete-time markov chains.* In *International Colloquium on Theoretical Aspects of Computing*, pages 280–294. Springer, 2004. 12

[28] Farias, Arthur José Rodrigues: *Integrating data mining into contextual goal modeling to tackle context uncertainties at design time.* Master's thesis, University of Brasília, 2017. `http://repositorio.unb.br/handle/10482/31637`. 16

[29] Knauss, Alessia, Daniela Damian, Xavier Franch, Angela Rook, Hausi A Müller, and Alex Thomo: *Acon: A learning-based approach to deal with uncertainty in contextual requirements at runtime.* Information and software technology, 70:85–99, 2016. 16, 17

[30] Ramirez, Andres J, Adam C Jensen, and Betty HC Cheng: *A taxonomy of uncertainty for dynamically adaptive systems.* In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 99–108. IEEE Press, 2012. 19

[31] Russell, Stuart J and Peter Norvig: *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016. 20

[32] *The prism language.* `https://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction`. Accessed: 2018-12-26. 21, 65

[33] 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.* In *Proceedings of the 14th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '19, 2019. 32, 40

[34] Pimentel, Joao and Jaelson Castro: *pistar tool–a pluggable online tool for goal modeling.* In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 498–499. IEEE, 2018. 34

[35] Farias, Leandro Santos Bergmann: *pistar-goda: Integração entre os projetos pistar e goda*, 2018. http://bdm.unb.br/handle/10483/20428. 34

[36] Mendonça, Danilo F.: *Dependability verification for contextual/runtime goal modelling*. Master's thesis, University de Brasilia, 2015. 34

[37] Parr, Terence: *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. 35

[38] Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach: *The goal question metric approach*. In *Encyclopedia of Software Engineering*. Wiley, 1994. 40

[39] Caldas, Ricardo Diniz: *Prototipação e verificação formal de sistema autônomo com propriedades tempo-real: Um estudo de caso no body sensor network*. Master's thesis, University of Brasília, 2017. http://bdm.unb.br/handle/10483/19223. 40, 56

[40] 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, ISBN 978-1-4503-5715-9. http://doi.acm.org/10.1145/3194133.3194147. 40

[41] Baresi, Luciano, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini: *Validation of web service compositions*. IET software, 1(6):219–232, 2007. 40

[42] Kwiatkowska, M., G. Norman, and D. Parker: *PRISM 4.0: Verification of probabilistic real-time systems*. In Gopalakrishnan, G. and S. Qadeer (editors): *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 48, 66

[43] Andersen, Henrik Reif: *An introduction to binary decision diagrams*. Lecture notes, available online, IT University of Copenhagen, page 5, 1997. 49

[44] Wilcoxon, Frank: *Individual comparisons by ranking methods*. Biometrics bulletin, 1(6):80–83, 1945. 52

[45] Cheng, Betty HC, Pete Sawyer, Nelly Bencomo, and Jon Whittle: *A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty*. In *International Conference on Model Driven Engineering Languages and Systems*, pages 468–483. Springer, 2009. 64

[46] Baresi, Luciano, Liliana Pasquale, and Paola Spoletini: *Fuzzy goals for requirements-driven adaptation*. In *2010 18th IEEE International Requirements Engineering Conference*, pages 125–134. IEEE, 2010. 64

[47] Filieri, Antonio, Giordano Tamburrelli, and Carlo Ghezzi: *Supporting self-adaptation via quantitative verification and sensitivity analysis at run time*. IEEE Transactions on Software Engineering, (1):75–99, 2016. 64, 65, 69

[48] Cailliau, Antoine and Axel van Lamsweerde: *Runtime monitoring and resolution of probabilistic obstacles to system goals.* In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*, pages 1–11. IEEE, 2017. 65, 69

[49] Bencomo, Nelly and Amel Belaggoun: *Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks.* In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 221–236. Springer, 2013. 65, 69

[50] Weyns, Danny and M Usman Iftikhar: *Model-based simulation at runtime for self-adaptive systems.* In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 364–373. IEEE, 2016. 65, 69

[51] Su, Guoxin, Taolue Chen, Yuan Feng, David S Rosenblum, and PS Thiagarajan: *An iterative decision-making scheme for markov decision processes and its application to self-adaptive systems.* In *International Conference on Fundamental Approaches to Software Engineering*, pages 269–286. Springer, 2016. 65, 69

[52] Cámara, Javier, Bradley Schmerl, Gabriel A Moreno, and David Garlan: *Mosaico: offline synthesis of adaptation strategy repertoires with flexible trade-offs.* Automated Software Engineering, pages 1–32, 2018. 65, 66, 69

[53] Moreno, Gabriel A, Javier Cámara, David Garlan, and Bradley Schmerl: *Proactive self-adaptation under uncertainty: a probabilistic model checking approach.* In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 1–12. ACM, 2015. 65, 66, 69

[54] Calinescu, Radu, Lars Grunske, Marta Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli: *Dynamic qos management and optimization in service-based systems.* IEEE Transactions on Software Engineering, 37(3):387–409, 2011. 66, 69

[55] Filieri, Antonio, Carlo Ghezzi, and Giordano Tamburrelli: *A formal approach to adaptive software: continuous assurance of non-functional requirements.* Formal Aspects of Computing, 24(2):163–186, 2012. 66, 69

[56] Ghezzi, Carlo, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli: *Managing non-functional uncertainty via model-driven adaptivity.* In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 33–42. IEEE, 2013. 66, 69

[57] Calinescu, Radu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli, and Tim Kelly: *Engineering trustworthy self-adaptive software with dynamic assurance cases.* IEEE Transactions on Software Engineering, 2017. 67, 69