GODA: A Goal-Oriented Requirements Engineering Framework for Runtime Dependability Analysis

Danilo Filgueira Mendonça[a,c,*], Genaína Nunes Rodrigues[a,*], Vander Alves[a], Raian Ali[b], Luciano Baresi[c]

[a] *Department of Computer Science, University of Brasilia, Brazil*
[b] *Faculty of Science and Technology, Bournemouth University, United Kingdom*
[c] *Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy*

Abstract

Context: Many modern software systems must deal with changes and uncertainty. Traditional dependability requirements engineering is not equipped for this since it assumes that the context in which a system operates be stable and deterministic, which often leads to failures and recurrent corrective maintenance. The Contextual Goal Model (CGM), a requirements model that proposes the idea of context-dependent goal fulfillment, mitigates the problem by relating alternative strategies for achieving goals to the space of context changes. Additionally, the Runtime Goal Model (RGM) adds behavioral constraints to the fulfillment of goals that may be checked against system execution traces.

Objective: This paper proposes GODA (Goal-Oriented Dependability Analysis) and its supporting framework as concrete means for reasoning about the dependability requirements of systems that operate in dynamic contexts.

Method: GODA blends the power of CGM, RGM and probabilistic model checking to provide a formal requirements specification and verification solution. At design time, it can help with design and implementation decisions; at runtime it helps the system self-adapt by analyzing the different alternatives and selecting the one with the highest probability for the system to be dependable. GODA is integrated into TAO4ME, a state-of-the-art tool for goal modeling and analysis.

Results: GODA has been evaluated against feasibility and scalability on Mobee: a real-life software system that allows people to share live and updated information about public transportation via mobile devices, and on larger goal models. GODA can verify, at runtime, up to two thousand leaf-tasks in less than 35ms, and requires less than 240 kbytes of memory.

Conclusion: Presented results show GODA's design-time and runtime verification capabilities, even under limited computational resources, and the scalability of the proposed solution.

Keywords: Goal modeling, Dependability, Probabilistic Model Checking, Runtime Analysis

1. Introduction

Many failures in software systems stem from poor requirements elicitation [14] and thus a proper understanding of what the system is supposed to do is key for its *dependability*. To this end, GORE (Goal-Oriented Requirements Engineering, [45]) offers proved means to decompose technical and non-technical requirements into well-defined entities (goals) and reason about the alternatives to meet them.

More recently, GORE has been used as means to model and reason about the systems' ability to adapt to changes in dynamic environments [1, 33]. Goals have been used as both design and runtime artifacts. Goal modeling has been used to customize software systems with respect to the characteristics of the organization they are deployed in [3], to derive high-variability designs, and to maximize the resilience and adaptivity of deployed systems [49, 44]. It has also been used as runtime model to respond to dynamic changes — while maintaining dependability. For example, goals become live entities that can self-adapt according to

*Corresponding authors
Email addresses: danilo.filgueira@polimi.it (Danilo Filgueira Mendonça), genaina@unb.br (Genaína Nunes Rodrigues), valves@unb.br (Vander Alves), rali@bournemouth.ac.uk (Raian Ali), luciano.baresi@polimi.it (Luciano Baresi)

the context [7, 8], or are complemented with meta-requirements that refer to their success or failure and can recover from errors [42]. The Runtime Goal Model (RGM) [13] augments goals and tasks with runtime specifications to verify whether their instances behave correctly, that is, they are dependable.

In previous work [33], we proposed the Dependability Contextual Goal Model, which exploits fuzzy logic to reason about the effects the actual context of operation has on both dependability requirements and dependability attributes. However, after studying some further real-life case studies, we have understood that the approach could become prohibitively heavy and time-consuming due to the effort required in providing declarative rules for each different goal, attribute, and context. Defined rules could also be corrupted by imprecise domain knowledge.

In addition, most of the solutions for eliciting dependability requirements do not take into account the history of failures. This is mandatory to be able to foresee probabilities of success and failing trends, and thus to support decision making procedures that can identify appropriate strategies to keep the system dependable. As a consequence, we advocate that dependability requirements models must be probabilistic, and that sound approaches and new tools be developed to guide self-adaptive capabilities and guarantee the fulfillment of goals.

In this context, *probabilistic model checking* (PMC) is suitable for reasoning about dependability requirements since it helps compute the probabilities with which these properties are satisfied [6]. PMC has been largely supported by tools such as PRISM [29] and PARAM [22]. The challenge is thus the conceptualization and formulation of dependability requirements in a way suitable for PMC.

This paper proposes the Goal-Oriented Dependability Analysis framework (GODA) to model goals and analyze their fulfillment in different contexts. GODA takes into account runtime aspects and accommodates the implications that contextual information may have on goal satisfaction. Since the overall goal satisfaction may be impacted by context restrictions, GODA provides a means to specify the interplay between them and to estimate the dependability of the strategies to fulfill goals in different contexts. At runtime, the outcome provided by GODA can be used to analyse whether the system is fulfilling its dependability goals. If it turns out that the obtained dependability is under a certain threshold, the system should consider the strategy (or strategies) that provide the most suitable dependability measure.

The proposed analysis solution relies on discrete-time PMC, where required specifications are obtained automatically from contextual runtime goal models. These goal models borrow concepts from contextual goal models [1] and runtime goal models [13]. Obtained models are verified through parametric PMC to take into account the possible variability of the probabilities in the model. However, since parametric PMC solutions are not fast enough and do not scale as needed, we propose an innovative solution for computing the parametric formulae. Our solution uses model checking to precompute the formulae that define the dependability of any node of the goal tree and takes into account the type of decomposition and runtime and context annotations. It then composes the different probabilities through suitable rewriting by mimicking the tree structure of the goal model. GODA is implemented as an extension to TAOM4E [36]: a TROPOS-based requirements elicitation and modeling tool implemented on top of the Eclipse framework.

In this paper, we also report on the empirical evaluation we carried out. First, we evaluate GODA on Mobee: a real-life software system that allows people to share live and updated information about public transportation via mobile devices. Mobee has been running for over a year and has already more than three thousand users. Our results in Mobee show that GODA is capable of performing dependability analysis efficiently, allowing it to be used under limited computational resources. The second part of the empirical study presents a time-space scalability analysis of the parametric verification. We artificially created goal models up to two thousand leaf-tasks, simulation results show a verification time below 35ms, and a use of less than 240 kbytes of memory in the worst-case.

The rest of the paper is organized as follows. Section 2 recalls the necessary background. Section 3 presents the conceptual model behind GODA, the proposed dependability analysis, and sketches our implementation as extension to TAOM4E. Section 4 describes the evaluation we carried out. Section 5 surveys related approaches and Section 6 concludes the paper.

2. Background

2.1. Goal Modeling and Context

Goal modeling provides a means to analyze the many requirements of the different stakeholders of a software system [14, 48, 10]. As defined by TROPOS [10], goals are owned by actors and actors may inter-depend on each other to reach their goals. Goals are ultimately

fulfilled by *leaf-tasks*, which denote processes to be executed by actors. Goals and tasks are decomposed and organized in a tree-structured model. A goal can be decomposed in subgoals or refined by a *means-end* task. Means-end relationships link a goal to a (means-end) task whose fulfillment is a sufficient and necessary condition to the fulfillment of the goal. A non-leaf task can be decomposed into other subtasks. An *AND-decomposition* requires that all sub-nodes to be fulfilled, while an *OR-decomposition* requires that at least one sub-node be fulfilled. Accordingly to TROPOS, only one type of decomposition per node is allowed. The alternative paths (OR-decompositions) in the goal tree can be evaluated with respect to qualitative objectives called *soft-goals*. Soft-goals are goals without a clear-cut criteria for their fulfillment. *Contribution links* identify the positive or negative impact of alternatives on soft-goals.

Figure 1 presents the different concepts related to goal modelling used by GODA. A single system actor (Mobee Mobile) has a root goal G_1, which is AND-decomposed into G_3 and G_4, meaning both subgoals must be achieved to fulfill G_1. Goal G_3 is AND-decomposed into G_8 and G_9, which are linked to tasks T_1 through means-end decomposition links. Other goals in the model are refined in a similar way. Tasks are also refined through AND/OR-decompositions, except leaf-tasks such as $T_{1.21}$ and $T_{1.22}$. Despite the support for multiple actors in TROPOS, in this work we consider a single system actor, i.e., all goals, tasks and relationships belonging to a system actor model.

An OR-decomposition represents alternative ways of fulfilling actor's goals [49]. For instance, task $T_{1.1}$: *Fetch geolocation* (a sub-tree of means-end task *T1: Track line location*) has two alternative tasks to track geo-location: *Fetch GPS* and *Fetch triangulation*. Each task contributes to the soft-goal *Geolocation accuracy* through positive or negative contribution links. Additionally, the fulfillment of a goal, the alternatives to do it, and the quality of each alternative can all be context-dependent [1]. In a Contextual Goal Model (CGM), context is defined as a formula of world predicates, henceforth defined as context facts. For example, if the context formed by fact *GPS Available* does not hold, the only way of fetching the geo-location would be through triangulation. Or, if *Battery* $\leq$ *15* holds, only manual information should be collected and goal G_4 is disabled, meaning that G_3 alone satisfies G_1 in that context.

2.2. Probabilistic Model Checking

Our proposal advocates that dependability requirements models be probabilistic, and that sound approaches and new tools be developed to guide self-adaptive capabilities and guarantee the fulfillment of goals. This is because goals are ultimately fulfilled by executable processes, which can fail due to different reasons. As a result, the dependability of a goal execution relies ultimately on the reliability of its leaf-tasks. For example, in order to realize goal G_1 : $Transportinfoisshared$ of Figure 1, if any of the leaf-tasks fails (e.g., $T_{1.22}Postdata$), the realization of G_1 can be sensibly compromised.

In contrast to model checking techniques for which the absolute correctness of a system is verified, probabilistic model checking (PMC) is about computing the probabilities with which the properties of interest are verified on the system. To this end, conventional transition systems are enriched with probabilities [6]. In our work, such probabilities represent estimates of successful task executions, i.e. the reliability of the different tasks. Accordingly, probabilistic model checking allows for quantitative statements about the behavior of a system, expressed as probabilities or expectations, in addition to the qualitative statements made by conventional model checking [26].

Transition systems are often augmented with Discrete-Time Markov Chains (DTMC) and Markov Decision Processes (MDP) [26]. In addition to specifying the probability and/or timing of transition occurrences, one can also set *rewards* (or equivalently, costs) to specify additional properties of interest [25] (e.g., expected power consumption or number of lost messages).

The PMC technique used in this paper is supported by the PRISM probabilistic model checker [29], which supports discrete- and continous-time markov chains as well as MDP. This choice comes from the richness of its environment —and of the probabilistic models it supports— and from its maturity, witnessed by the many different successful uses and case studies [27]. Both qualitative and quantitative analysis are supported. As we further explain in Section 3.2, this work adopts DTMCs as our modeling structures. As already said, goals are ultimately realized by leaf-tasks, and the overall system behavior is the realization of these tasks. Since the behavior of a leaf-task is modeled as a finite transition system, a DTMC is a natural modeling choice since it inherits the structures of such transition systems.

Modules are the main structures in a PRISM model [28]: local **variables** describe its (finite) states, while **commands** its behavior, that is, state transitions constrained by **guards**. These guards can predicate on any variable in the module. Finally, **actions** are used for naming commands and for synchronization. For example, a PRISM DTMC module takes the syntax in List-

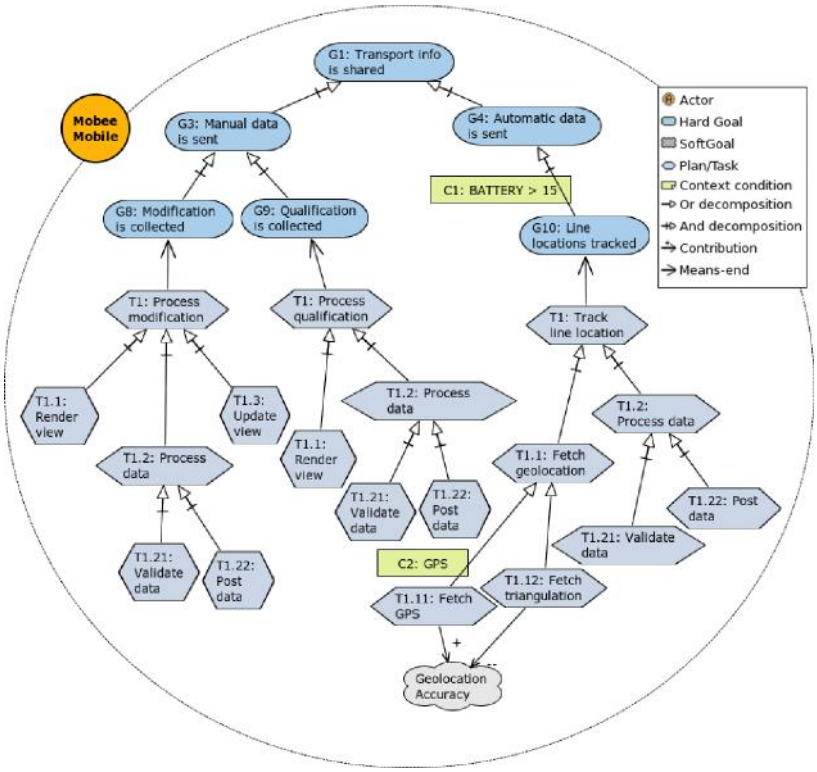


Figure 1: Excerpt of the Contextual Goal Model for the Mobee Mobile System

ing 1.

```
const double <probability>; //model constant
module <module_name>
    state : [0..n] init 0; //local module
        variable

    [<action>] <state guard> ->
        <probability>:(<state update>);//command
endmodule
```

Listing 1: PRISM DTMC syntax.

Once predicate *guard* is satisfied, the module makes a transition with a certain *probability*, where $0 \leq probability \leq 1$, to state *update*. The *probability* may be omitted, in which case it is assumed to be 1. Label *action* is used to tag a command that has to be synchronized with other commands in the same or in different modules following a process algebra style, as long as guard conditions are enabled. When there is no *action* label, the command is run asynchronously[1].

To reason about the properties of a system modeled as a DTMC, one can use Probabilistic Computation Tree Logic (PCTL) [24], which extends the original time-bounded or unbounded temporal logics for property specification in Computation Tree Logic (CTL) [12]. The specification of dependability properties in PCTL has been extensively explored in previous works [25, 41, 26]. In particular, the work by Grunske [18] shows that the reachability property expressed by the Probabilistic Existence PCTL formula $P =? [F(\varphi)]$ has been widely used to specify dependability attributes. Such a Probabilistic Existence property computes the probability with which a system will *eventually reach* a state that satisfies φ. The satisfaction of this formula also guarantees that a final and successful system state is reached regardless of the time needed to reach it. If a time bound is a concern, the Probabilistic Existence formula is expressed as $P =? [F^{[t_1,t_2]}(\varphi)]$, and it computes the probability with which φ will eventually become *true* within the time interval $[t_1, t_2]$. In our proposal, we verify the dependability properties that pertain the successful fulfillment of system goals, irrespective of the time. Therefore, the temporal logic property of our interest is the unbounded Probabilistic Existence.

Conventional PMC associates single values to the probabilities with which state transitions can be taken. This means that every time one of these values changes, the model must be recreated and reanalyzed —maybe,

[1]We refer the careful reader to [28] for further details on the PRISM language.

partially [15]. In contrast, parametric model checking calls for transition probabilities specified as functions over a set of parameters. Such a model checking technique enables a more flexible analysis, where verification produces *parametric formulas* and not single values: the probabilistic satisfiability of a PCTL property is an algebraic formula that relates the probability of satisfying the formula to used parameters. For example, these formulas can be computed off-line, while values (obtained by runtime system monitoring) are assigned to parameters at runtime.

To this end, PARAM [22] is a parametric model checking tool that extends the PRISM[2] modeling language with the possibility of specifying parameters as either float or integer values [23] via

$$param\ int|float < parameter >;$$

These parameters can then be used to specify the probability distributions in the model.

For example, PARAM would only require to change line 1 of Listing 1 into *param double < probability >*. The parametric formula is then built automatically by PARAM[3]. This way one can also solve the formula for a particular configuration of the parameters, or identify the optimal values of the parameters for satisfying it [22].

3. GODA

Figure 2 illustrates the analysis process behind GODA (Goal-Oriented Dependability Analysis): a first *Goal Model* is produced through conventional goal modeling, then it is augmented with contextual and runtime information and becomes a *Contextual and Runtime Goal Model* (CRGM). Once the CRGM of the system is complete, it is automatically translated 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. For example, context restrictions may end up in no alternatives to fulfill a certain goal, and this may require changes to the goal model. It could also be that the design-time analysis reveals violations in the dependability constraints, and then a new iteration must be carried out.

3.1. Contextual and Runtime Goal Models

A *Contextual and Runtime Goal Model* (CRGM) refines the concept of *Contextual Goal Model* and of *Runtime Goal Model*. In this case, goals are to be fulfilled by the system by taking into account (1) contextual specifications of the CGM [1] and (2) the cooperation of instantiated goals and tasks at runtime, following RGM runtime rules, extended from Dalpiaz et al. [13]. Table 1 synthesizes a textual description of each RGM rule we apply in GODA and its corresponding semantics.

Expression	**Meaning**
AND (n1;n2)	Sequential fulfillment of n1 and n2.
AND (n1#n2)	Parallel fulfillment of n1 and n2.
OR (n1;n2)	Sequential fulfillment of either n1 or n2 or both.
OR (n1#n2)	Parallel fulfillment of either n1 or n2 or both.
n+k	n must be fulfilled k times, with $k > 0$.
n#k	Parallel fulfillment of k instances of n, with $k > 0$.
n@k	Maximum $k-1$ fulfillment retries of k, with $k > 0$.
opt(n)	Optional fulfillment of n.
try(n)?n1:n2	If n is fulfilled, n1 must be fulfilled; otherwise, n2.
n1|n2	Alternative fulfillment of either n1 or n2, not both.
skip	No action. Useful for conditional ternary expressions.

Table 1: RGM rules, where n, $n1$ and $n2$ represent goals or tasks in a goal model (extended from Dalpiaz et al. [13]).

Suppose a node n, which could be either a goal or a task in a CRGM, is decomposed into $n1$ and $n2$. The AND/OR-decomposition in our CRGM is fulfilled in accordance with the sequential and parallel runtime rules associated with the parent node n: AND $(n1; n2)$ requires the fulfillment of $n1$ prior to $n2$, while an AND $(n1\#n2)$ does not impose any order on the fulfillment of $n1$ and $n2$.

An OR-decomposition introduces a variation point [49]. As a result, goals/tasks in an OR-decomposition are mutually independent in their fulfillment. That is, the OR-decomposition of n into $n1$ and $n2$ requires that: (1) either $n1$ or $n2$ be fulfilled and (2) the fulfillment of $n2$ is independent of the fulfillment of $n1$. As such, an OR $(n1; n2)$ means the sequential fulfillment of either $n1$ or $n2$ or both, while an OR $(n1\#n2)$ does not impose any order on the fulfillment of either $n1$ or $n2$ or both.

The other rules work as follows. If n (either a goal or a task) must be repeated k times ($n+k$ rule), k instances of n must be fulfilled. If a node is *opt*-annotated, it is

[2]Currently, PRISM model checker also supports parametric model checking [29], which is an implementation of the same foundation over which PARAM is implemented [21, 20].

[3]We refer the careful reader to [23] for further explanation on PARAM.

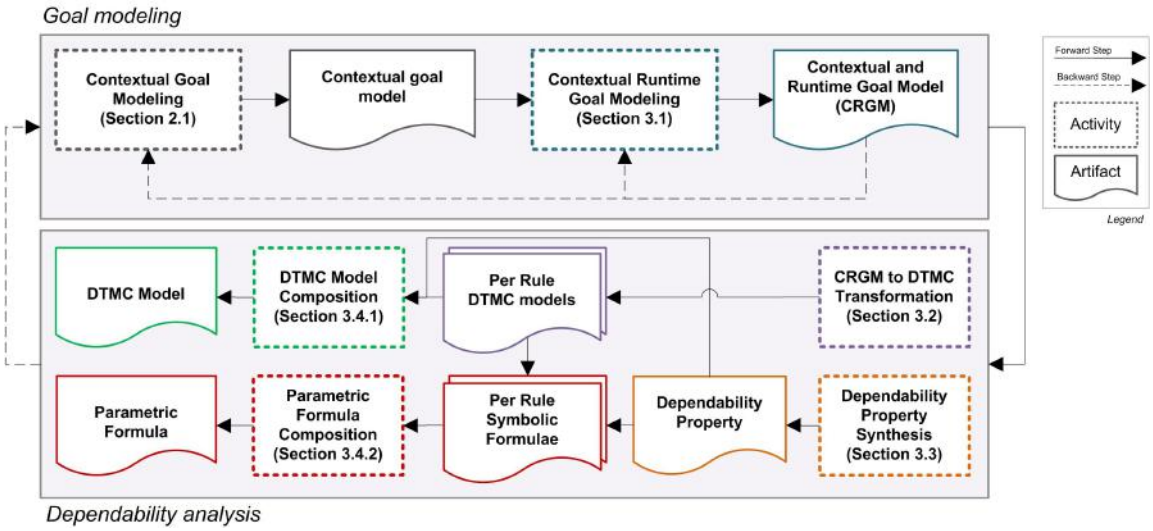


Figure 2: GODA process.

optionally fulfilled, otherwise skipped. The ternary *try* $(try(n)?\, n1 : n2)$ is interpreted as the fulfillment of n followed by $n1$ (if n succeeds) or $n2$ (should n fail). Finally, alternative rule $(n1|n2)$ is interpreted as an *exclusive or* execution of $n1$ or $n2$.

Definition 3.1. *A CRGM is a tuple (M, rt_annot, ctx_annot, ID) where:*

- *M is a design-time goal model [13] defined as a tuple (N, R), where N is a set of goals and tasks in the model, and R the corresponding set of relationship links between the elements in N.*
- *rt_annot is a runtime annotation function that returns the runtime annotation associated with a node $n \in N$ such that $rt\_annot(n)$ can be a single runtime rule or a composition of such rules.*
- *ctx_annot is a context annotation function that returns the context formulae associated with a node $n \in N$.*
- *ID is an index function that maps all $n \in N$ to an identifier $ID(n) = prefix(n) + counter(n)$, where $prefix(n)$ returns 'G' or 'T' for goals and tasks, respectively, + operator is a simple concatenation function and $counter(n)$ returns:*
  - *an integer to uniquely identify goals in ascending order;*
  - *a constant 1 for means-end tasks;*
  - *an integer vector corresponding to the branch and depth level of the task, incremented in ascending order;*

Figure 3 presents the CRGM that corresponds to the goal model of Figure 1, where each decomposed node is augmented with runtime annotations placed underneath the name of the node in square brackets. Root goal G_1 is achieved by the parallel fulfillment (denoted as $[G_3\#G_4]$) of subgoals G_3 and G_4, while subgoal G_3 is the realization of sub-goals G_8 and G_9, which are fulfilled by the realization of their corresponding means-end tasks. Taking the realization of task T_1 *Process modification*, the sequential successful execution of leaf-tasks $T_{1.1}$, $T_{1.2}$ and $T_{1.3}$ fulfills T_1, while $T_{1.2}$ is realized by the sequential realization of $T_{1.21}$ and $T_{1.22}$. Note that subtask $T_{1.22}$ can be executed at most twice (denoted as $[T_{1.22}@2]$) if the first execution fails (and thus the task is not realized). On the right-hand side branch, subgoal G_4 is realized by subgoal G_{10}, but only if context condition C_1 holds. Otherwise, G_4 is disabled and G_1 depends on G_3 only. Task T_1 is realized by the sequential and successful execution of $T_{1.1}$ and $T_{1.2}$. In particular, T_1 is fulfilled if at least one execution —out of maximum three attempts— of task $T_{1.1}$ is successful and then also task $T_{1.2}$ is realized, since the activation of $T_{1.2}$ depends on the success of $T_{1.1}$ (denoted as $[try(T_{1.1}@3)?\, T_{1.2} : skip]$). Task $T_{1.1}$ can be satisfied by the fulfillment of either $T_{1.11}$ or $T_{1.12}$. Additionally, $T_{1.11}$ is only available if context C_2 is satisfied.

Finally, to exemplify *ID* index function let us consider task $T_{1.2} : Processdata$ on the rightmost side

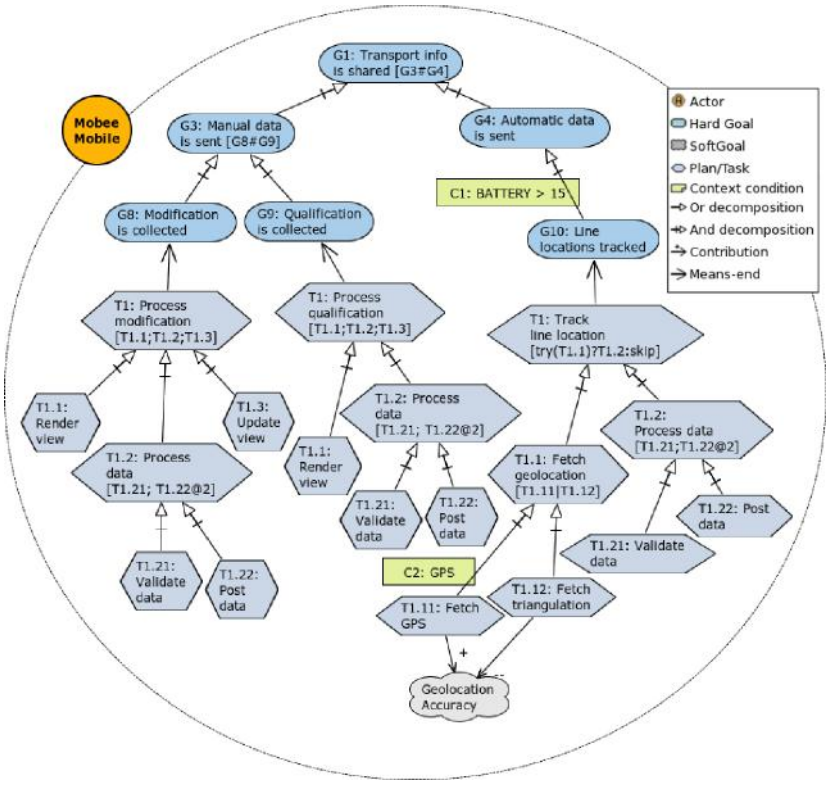


Figure 3: Corresponding CRGM for the goal model from Figure 1.

of Figure 3. Its T prefix identifies that *Process data* is a task. Function *counter* returns a decimal number (1.2), where 1 inherits from its parent means-end task T_1, while the remaining part (.2) represents its location vector w.r.t. to its parent node T_1. The content of the location vector (2) represents the second AND-decomposition branch of T_1 and the position in the vector represents its depth-level. Accordingly, the child nodes of a task inherit their parent's index and increment their location vector following their branch and depth level. For example, task $T_{1.2}$ has two children: *Validate data* and *Post data*. As a result, *Validate data*, on the first AND-decomposition branch, is indexed as $T_{1.21}$, while *Post data* is indexed as task $T_{1.22}$ since it is on the second AND-decomposition branch of $T_{1.2}$. Note that both child nodes of $T_{1.2}$ have their location vector incremented following their depth level in the tree structure.

3.2. From CRGM to DTMC

This section describes the transformation of a CRGM into a corresponding DTMC, rendered in the PRISM language.

The state diagram of Figure 4 describes the states of CRGM nodes. Differently from [13], state *Waiting* is not explicitly represented since we abstract it away in state *Running*. Additionally, state *Skipped* has been added to represent nodes that have not been either selected as an option (or alternative) or enacted due to contextual restrictions.

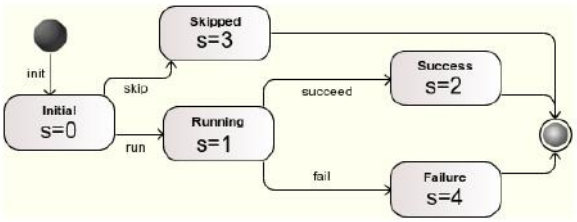


Figure 4: Transition states of CRGM nodes.

For each state in Figure 4, there is a corresponding value for the state variable s in the PRISM model. State **Initial** ($s = 0$) corresponds to the initial/ready state of a node. From this state, a node can enter state **Running** ($s = 1$), if it is selected, or the final state **Skipped** ($s = 3$), if it does not participate in the fulfillment of the

parent goal. The result of the fulfillment can be either **Success** ($s = 2$) or **Failure** ($s = 4$), that is, the fulfillment has deviated from its expected behavior as a consequence of natural phenomena, a human-made fault, a malicious fault, or an interaction fault [5].

The DTMC that corresponds to the state diagram of Figure 4 is presented in Listing 2 as a PRISM module. In particular, *dNode* represents the probability of the successful fulfillment of the node, that is, the probability of moving from **Running** (s=1) to **Success** state (s=2). Consequently, $1 - dNode$ represents the probability from **Running** (s=1) to **Failure** state (s=4). Due to optionality of node fulfillment, transition to **Skipped** state (s=3) means the node is opted-out. For simplicity, skipped states have their *action* labelled in the DTMC as *success*.

Note that the way in which *dNode* is initialized depends on the type of the node. If the node is a non-leaf task, *dNode* is given a value by means of the probabilistic existence property (Equation 1). For leaf-tasks, considering they ultimately realize goals in a concrete level, their *dNode* can be obtained in various ways, for example, by monitoring the system for Mean-times to failure (MTTF). A sound approach for such computation at runtime may follow from recent work in the field by Su et al. [43].

```
1  double dNode; //dependability of a node
2  module N1
3    s :[0..4] init 0;
4
5    [init] s = 0 -> (s'=1);//init to running
6    [] s =  1 -> dNode : (s'=2) + (1 - dNode) : (s
         '=4);//running to final or failure states
7    [success] s = 2 -> (s'=2);//final state success
8    [success] s = 3 -> (s'=3);//final state skipped
9    [fail] s = 4 -> (s'=4);//final state failure
10 endmodule
```

Listing 2: DTMC for the state diagram of nodes.

The DTMC for a goal G_i is obtained by composing the DTMC models of its subtrees. Accordingly, the DTMC models of nodes that fulfill all chains of subgoals up to G_i will be composed. To provide a precise specification of the transformation of a CRGM into a DTMC, we propose a generic inference rule in the form of structured operational semantics:

$$\frac{C_1 \vdash n_1 \rightarrow d_1 \; < C_2 \vdash n_2 \rightarrow d_2, C_3 \vdash n_3 \rightarrow d_3 >}{C_1 <, C_2, C_3 > \vdash *(n_1 <, n_2, n_3 >) \rightarrow \circledast(d_1 <, d_2, d_3 >)}$$

where n_1, n_2, and n_3 are are either CRGM goals or tasks, d_1, d_2, and d_3 are DTMCs, and C_1, C_2, and C_3 are context facts. $*$ identifies any if the runtime rule operators (sequential, interleaving[4], alternative, optional, conditional, cardinality), described bellow. Angle brackets denote an optional list of parameters to accommodate the different arities of the operators. Symbol $\vdash$ indicates that the context on the left is used in the transformation of the node on the right into a corresponding DTMC. If the context is omitted, it is assumed to be empty. The DTMC representation of a composed node $*(n_1 <, n_2, n_3 >)$ then corresponds to defining $\circledast(d_1 <, d_2, d_3 >)$ in terms of the primitives provided by PRISM, as explained in Subsections 3.2.1–3.2.5.

The DTMC obtained by transformation from a CRGM must preserve the temporal order among goals and tasks and respect the further behavioral constraints imposed by runtime rules. To convey this, we have defined an abstract fulfillment time of the CRGM nodes formalized as follows:

Definition 3.2. *The fulfillment time of a CRGM node is a tuple $\tau = (f, p)$, where p represents a sequential execution branch, named time path, and f, named time frame, represents the sequential execution position within a given time path, with $f, p \in \mathbb{N}$.*

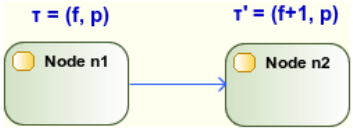
The fulfillment time of a CRGM starts at $\tau = (0, 0)$, and follows a depth-first traversal of the goal tree, from left to right, in accordance with the runtime rules of each CRGM node (goals and tasks). Nodes can be fulfilled either sequentially or in parallel with respect to each other. The initial state of each CRGM node is mapped to a specific τ, which defines the time before the fulfillment of the node. Sequential nodes n_i and n_j are in subsequent time frames, that is, $\tau_i = (f, p)$, and $\tau_j = (f', p)$, where $f' > f$. In contrast, parallel nodes share the same time frame and path, that is, $\tau_i = \tau_j = (f, p)$.

The effect of each rule on the τ of the involved nodes after their fulfillment —that is, in their final states— is also described in the corresponding annotations to the activity diagrams. It is also described in the DTMC expression $\circledast(d_1 <, d_2, d_3 >)$ as a suffix in the [*action*] construct of specific transitions, as follows: the first suffix term represents the time frame f and the second, separated by an underscore, the time path p of that node. The suffix is used for the synchronization of final and initial states of subsequent nodes and initial states of parallel nodes.

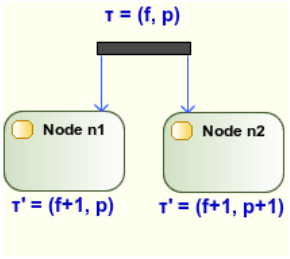
[4] A widely adopted paradigm for parallel systems is that of *interleaving*, that is, the nondeterministic choice between activities of the simultaneously acting processes [6]. Henceforth, the term *interleaving* is used to identify the nondeterministic fulfillment of nodes under parallel runtime rule.

3.2.1. *Sequential and Parallel Nodes*

Figures 5a and 5b illustrate the sequential and parallel fulfillment of nodes n_1 and n_2, where semicolon (;) denotes sequential and hash (#) denotes parallel, respectively. The bold line in Figure 5b refers to the parallel fulfillment of nodes n_1 and n_2.



(a) Sequential nodes $n_1; n_2$.



(b) Parallel nodes $n_1 \# n_2$.

Figure 5: Sequential and parallel nodes

Sequential nodes $(n_1; n_2)$ have subsequent time frames, meaning that the fulfillment of n_2 comes after the fulfillment of n_1. To model this, we have synchronized the initial state of d_2 (the DTMC that corresponds to n_2) with the final state of d_1, as shown in Listing 3. The synchronization occurs between lines 11/13 (final state of first module) and lines 21/23 (initial state of second module). Lines 27/29 say that the time frame is increased and the time path remains the same.

In contrast, parallel nodes $(n_1 \# n_2)$ have the same τ, i.e., they have the same time frame and path, meaning that they start their fulfillment in parallel—achieved through the synchronization of their initial state. Interleaving occurs in the running state. Accordingly, in Listing 4, the interleaving occurs at lines 9/25 (running state), while the synchronization in the initial state occurs between lines 5/7 and lines 21/23. Lines 11/13 and lines 27/29 (final state) are synchronized with sequential nodes (if any) after n_1 and n_2, respectively.

In both cases, the context of each node is addressed by using formulas (e.g, $C1$ and $!C1$ in n_1 and similarly for n_2) that are then employed as guard conditions in the corresponding node modules, for example, lines 5,7,21,23 in Listing 3 and Listing 4. If the context formula is not satisfied, a deterministic transition to the skipped final state automatically excludes the related nodes from the analysis; otherwise, a transition evolves the node to the running state. This treatment of context is also applied to the remaining templates.

```
1  double dNode;
2  module N1
3    sN1 :[0..4] init 0;
4    //init to running
5    [initN1<f>_<p>] C1 & sN1 = 0 -> (sN1'=1);
6    //init to skipped
7    [initN1<f>_<p>] !C1 & sN1 = 0 -> (sN1'=3);
8    //running to final state
9    [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1 -
         dNodeN1) : (sN1'=4);
10   //final state success
11   [success<f>_<p>] sN1 = 2 -> (sN1'=2);
12   //final state skipped
13   [success<f>_<p>] sN1 = 3 -> (sN1'=3);
14   //final state failure
15   [failN1] sN1   = 4 -> (sN1'=4);
16 endmodule
17
18 module N2
19   sN2 :[0..4] init 0;
20   //init to running
21   [success<f>_<p>] C2 & sN2 = 0 -> (sN2'=1);
22   //init to skipped
23   [success<f>_<p>] !C2 & sN2 = 0 -> (sN2'=3);
24   //running to final state
25   [] sN2 =  1 -> dNodeN2 : (sN2'=2) + (1 -
         dNodeN2) : (sN2'=4);
26   //final state success
27   [success<f+1>_<p>] sN2 = 2 -> (sN2'=2);
28   //final state skipped
29   [success<f+1>_<p>] sN2 = 3 -> (sN2'=3);
30   //final state failure
31   [failN2] sN2 = 4 -> (sN2'=4);
32 endmodule
```

Listing 3: DTMC template for the sequential composition of nodes n_1 and n_2.

```
1  double dNode;
2  module N1
3    sN1 :[0..4] init 0;
4    //init to running
5    [init<f>_<p>] C1 & sN1 = 0 -> (sN1'=1);
6    //init to skipped
7    [init<f>_<p>] !C1 & sN1 = 0 -> (sN1'=3);
8    //running to final state
9    [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1 -
         dNodeN1) : (sN1'=4);
10   //final state success
11   [success<f>_<p>] sN1 = 2 -> (sN1'=2);
12   //final state skipped
13   [success<f>_<p>] sN1 = 3 -> (sN1'=3);
14   //final state failure
15   [failN1] sN1 = 4 -> (sN1'=4);
16 endmodule
17
18 module N2
19   sN2 :[0..4] init 0;
20   //init to running
21   [init<f>_<p>] C2 & sN2 = 0 -> (sN2'=1);
22   //init to skipped
23   [init<f>_<p>] !C2 & sN2 = 0 -> (sN2'=3);
24   //running to final state
25   [] sN2 =  1 -> dNodeN2 : (sN2'=2) + (1 -
         dNodeN2) : (sN2'=4);
26   //final state success
27   [success<f+1>_<p+1>] sN2 = 2 -> (sN2'=2);
28   //final state skipped
29   [success<f+1>_<p+1>] sN2 = 3 -> (sN2'=3);
30   //final state failure
31   [failN2]  sN2 = 4 -> (sN2'=4);
```

```
endmodule
```

Listing 4: DTMC template for the parallel composition of nodes n_1 and n_2.

3.2.2. Optional Node

A node in an AND-decomposition may be tagged as optional, meaning that its fulfillment is not a necessary condition for the fulfillment of the element it refines. The other non-optional elements in an AND-decomposition must be fulfilled. Figure 6 illustrates an optional node denoted as $opt(n)$.

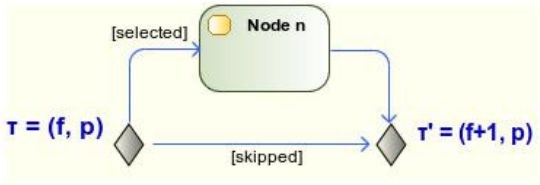


Figure 6: Optional node $opt(n)$.

Optional nodes do not cause any increment to τ. To synchronize an optional node n_1 with a subsequent node n_2, the initial state of n_2 is synchronized with both the *skipped* and *succeeded* states of n_1. Correspondingly, the optionality in the DTMC model of Listing 5 comes from an additional constant OPT_N_ID, whose value determines the actual state after the initial one: *running* if $OPT\_N\_ID = 1$, or *skipped* if $OPT\_N\_ID = 0$. This is implemented in the PRISM model (Line 7 of Listing 5) by means of a deterministic transition to either state running or state skipped. Therefore, the selection depends on an external input provided at design-time by the user or at runtime by the component responsible for dependability analysis.

```
1  const int OPT_N1;
2  double dNode;
3
4  module N1
5    sN1 :[0..4] init 0;
6    //init to running or to skipped
7    [initN1<f><p>] C1 & sN1 = 0 -> (OPT_N1) : (sN1
         '=1) + (1-OPT_N1) : (sN1'=3);
8    //init to skipped
9    [initN1<f><p>] !C1 & sN1 = 0 -> (sN1'=3);
10   //running to final state
11   [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1-dNodeN1)
          : (sN1'=4);
12   //final state success
13   [success<f+1>_<p>] sN1 = 2 -> (sN1'=2);
14   //final state skipped
15   [success<f+1>_<p>] sN1 = 3 -> (sN1'=3);
16   //final state failure
17   [failN1] sN1 = 4 -> (sN1'=4);
18 endmodule
```

Listing 5: DTMC template for an optional node n_1 with additional selector OPT_N1.

3.2.3. Alternative Nodes

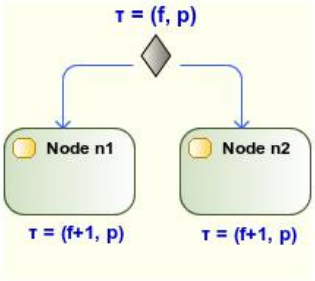


Figure 7: Alternative nodes $n1|n2$.

Alternative nodes are mutually exclusive (XOR-decomposition) and only one of them is executed. Figure 7 proposes the graphical representation of two alternative nodes ($n_1|n_2$). These nodes share both the time frame and path as only one node is selected. The increment of τ is defined by the preceding sequential or parallel rule. Like optional nodes, the selection among the alternative nodes depends on an external input. The corresponding DTMC model is then obtained by applying the pattern for optional nodes (Section 3.2.2) to each alternative node and also by requiring that the selectors be alternatively enabled, as shown in Listing 6. The alternative chosen to fulfill the parent node also comes from additional constants OPT_N_ID, which must be selected alternatively.

```
1
2  const int OPT_N1;
3  // alternatively selected with OPT_N1
4  const int OPT_N2;
5  double dNode;
6
7  module N1
8    sN1 :[0..4] init 0;
9    //init to running or to skipped
10   [init<f><p>] C1 & sN1 = 0 -> (OPT_N1) : (sN1
         '=1) + (1-OPT_N1) : (sN1'=3);
11   //init to skipped
12   [init<f><p>] !C1 & sN1 = 0 -> (sN1'=3);
13   //running to final state
14   [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1-dNodeN1)
          : (sN1'=4);
15   //final state success
16   [success<f+1>_<p>] sN1 = 2 -> (sN1'=2);
17   //final state skipped
18   [success<f+1>_<p>] sN1 = 3 -> (sN1'=3);
19   //final state failure
20   [failN1] sN1 = 4 -> (sN1'=4);
21 endmodule
22
23 module N2
24   sN2 :[0..4] init 0;
25   //init to running or to skipped
26   [init<f><p>] C2 & sN2 = 0 -> (OPT_N2) : (sN2
         '=1) + (1-OPT_N2) : (sN2'=3);
27   //init to skipped
28   [init<f><p>] !C2 & sN2 = 0 -> (sN2'=3);
29   //running to final state
```

```
  [] sN2 =  1 -> dNodeN2 : (sN2'=2) + (1-dNodeN2)
       : (sN2'=4);
  //final state success
  [success<f+1>_<p>] sN2 = 2 -> (sN2'=2);
  //final state skipped
  [success<f+1>_<p>] sN2 = 3 -> (sN2'=3);
  //final state failure
  [failN2] sN2 = 4 -> (sN2'=4);
endmodule
```

Listing 6: DTMC template for alternative nodes n_1 and n_2 with *OPT_N_*1 and *OPT_N_*2 used as selectors.

3.2.4. *Conditional Nodes*

The fulfillment of some nodes may depend on the fulfillment of others. In this case, an OR-decomposition is annotated with a ternary rule $try(n_1)\ ?\ n_2 : n_3$, where the fulfillment of n_2 depends on the success of n_1, whereas the fulfillment of n_3 depends on its failure. $Skip$ can be used instead of n_2 or n_3 to say that nothing must happen. For example, $try(n_1)\ ?\ skip : n_2$, states that n_2 is only executed if n_1 fails, and, if it succeeds, nothing happens. Note that the aforementioned solution can also be used to render a fault-tolerant fulfillment of n_1. Figure 8 illustrates a conditional decomposition.

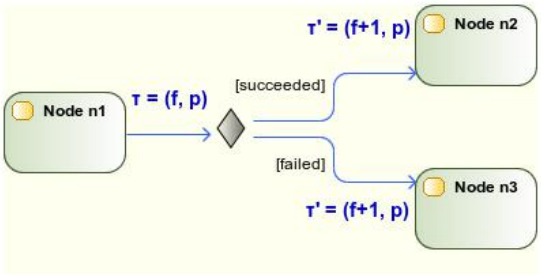


Figure 8: Conditional decomposition $try(n_1)\ ?\ n_2 : n_3$.

A conditional decomposition increments the time frame as the conditional nodes come after the success or failure of n_1. If n_1 succeeds, it synchronizes its final successful state with the initial state of the subsequent node, that is, n_2. Otherwise, the final $failed$ state of n_1 synchronizes with the initial state of the exceptional node (n_3). In case n_2 succeeds its fulfillment, it synchronizes with the *skipped* state of node n_3. The analogous synchronization procedure applies between nodes n_1 and n_3 in case the second argument of the ternary statement, n_2, is not present. Thus to represent conditional executions in DTMC models, shared actions synchronize the initial state of conditional nodes to the success and failure of the ancestor ones (Listing 7).

```
double dNode;

module N1
  sN1 :[0..4] init 0;
  //init to running
  [initN1<f><p>] C1 & sN1 = 0 -> (sN1'=1);
  //init to skipped
  [initN1<f><p>] !C1 & sN1 = 0 -> (sN1'=3);
  //running to final state
  [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1 -
       dNodeN1) : (sN1'=4);
  //final state success
  [success<f><p>] sN1 = 2 -> (sN1'=2);
  //final state skipped
  [success<f><p>] sN1 = 3 -> (sN1'=3);
  //final state failure
  [failN1] sN1 = 4 -> (sN1'=4);
endmodule

module N2
  sN2 :[0..4] init 0;
  //init to running
  [failN1]  C2 & sN2 = 0 -> (sN2'=1);
  //init to skipped
  [failN1]  !C2 & sN2 = 0 -> (sN2'=3);
  //not used, skip running
  [success<f><p>] sN2 = 0 -> (sN2'=3);
  //running to final state
  [] sN2 =  1 -> dNodeN2 : (sN2'=2) + (1 -
       dNodeN2) : (sN2'=4);
  //final state success
  [success<f+1><p>] sN2 = 2 -> (sN2'=2);
  //final state skipped
  [success<f+1><p>] sN2 = 3 -> (sN2'=3);
  //final state failure
  [failN2] sN2 = 4 -> (sN2'=4);
endmodule
```

Listing 7: DTMC template for conditional nodes n_1 and n_2 such that $try(n_1)\ ?\ skip : n_2$.

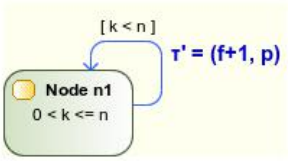
3.2.5. *Multiple Instances*

Instead of considering a single node instance, one can think of multiple instances of the same node n executed *sequentially* ($n + k$, Figures 9a) or *in parallel* ($n\#k$, Figure 9b). In both cases, an upper bound k is used to limit the number of instances that should be executed.

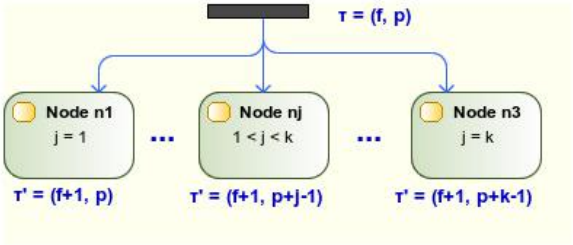
In the corresponding DTMC models, the modules are repeated $k - 1$ times —through renaming— in addition to the original one. In case of sequential composition, the state variable and the initial and final transition actions are renamed at each repetition, forming a sequential fulfillment chain of the same node, where $\tau_i = (f, p) \Rightarrow \tau_{i+1} = (f + 1, p)$, as shown in Listing 8.

In parallel compositions, only the state variable and the final state action are renamed, resulting in the synchronization of the initial transitions of the renamed modules, and the increment of the time path of subsequent nodes, which causes the run and final actions to remain unsynchronized. Thus, the renaming of modules forms an interleaving of the different instances of the same node at their running states, while the increment of $\tau_i = (f, p) \Rightarrow \tau_{i+1} = (f + 1, p + 1)$ leaves each final state unsynchronized w.r.t. other parallel nodes in the same decomposition (Listing 9).

```
double dNode;
module N1
```



(a) Sequential fulfillment of k instances of n ($n + k$).



(b) Parallel fulfillment of k instances of n ($n\#k$).

Figure 9: Multiple instances of n.

```
  sN1 :[0..4] init 0;
  //init to running
  [success<f>_<p>] C1 & sN1 = 0 -> (sN1'=1);
  //init to skipped
  [success<f>_<p>] !C1 & sN1 = 0 -> (sN1'=3);
  //running to final state
  [] sN1 =  1 -> dNodeN1 : (sN1'=2) + (1 -
      dNodeN1) : (sN1'=4);
  //final state success
  [success<f+1>_<p>] sN1 = 2 -> (sN1'=2);
  //final state skipped
  [success<f+1>_<p>] sN1 = 3 -> (sN1'=3);
  //final state failure
  [failN1] sN1 = 4 -> (sN1'=4);
endmodule
module N2 = N1 [ sN1=sN2, failN1=failN2, success<
    f+1>_<p>=success<f+2>_<p>, success<f>_<p>=
    success<f+1>_<p> ] endmodule
...
module N<K> = N<K-1> [ sN<K-1>=sN<K>, failN<K-1>=
    failN<K>, success<f+K-1>_<p>=success<f+K>_<p
    >, success<f+K-2>_<p>=success<f+K-1>_<p> ]
    endmodule
```

Listing 8: DTMC template for node $n1$ with $k =< K >$ sequential instances.

```
double dNode;
module N1
  sN1 :[0..4] init 0;
  //init to running
  [success<f>_<p>] C1 & sN1 = 0 -> (sN1'=1);
  //init to skipped
  [success<f>_<p>] !C1 & sN1 = 0 -> (sN1'=3);
  //running to final state
  [] sN1 = 1 -> dNodeN1 : (sN1'=2) + (1 - dNodeN1
      ) : (sN1'=4);
  //final state success
  [success<f+1>_<p>] sN1 = 2 -> (sN1'=2);
  //final state skipped
  [success<f+1>_<p>] sN1 = 3 -> (sN1'=3);
  //final state failure
  [failN1] sN1 = 4 -> (sN1'=4);
endmodule
```

```
module N2 = N1 [ sN1=sN2, failN1=failN2, success<
    f+1>_<p>=success<f+1>_<p+1>] endmodule
...
module N<K> = N<K-1> [ sN<K>=sN<K-1>, failN<K>=
    failN<K-1>, success<f+1>_<p+K-2>=success<f+1>
    _<p+K-1>] endmodule
```

Listing 9: DTMC template for node $n1$ with $k =< K >$ parallel instances.

In addition to sequential and parallel fulfillment, we also support the specification of retries for node fulfillment ($n@k$). This is a special case of sequential fulfillment. In this particular case, the *run* action is performed multiple times until either the maximum number of retries or the final success state is reached. If the former applies, the fail action takes the node to the final failure state. Listing 10 presents the corresponding DTMC template.

```
double dNode;
const double maxRetriesN1=<K>;

module N1
  sN1 :[0..4] init 0;
  triesN1 : [0..maxRetriesN1] init 0;
  //init to running
  [success<f>_<p>] C1 &  sN1 = 0 -> (sN1'=1);
  //init to skipped
  [success<f>_<p>] !C1 &  sN1 = 0 -> (sN1'=3);
  //try
  [] sN1 = 1 & triesN1 < maxRetriesN1 -> dNodeN1
      : (sN1'=2) + (1 - dNodeN1) : (triesN1'=
      triesN1+1);
  //no more retries
  [] sN1 = 1 & triesN1 = maxRetriesN1 -> (sN1'=4)
      ;
  //final state success
  [success<f+1>_<p>] sN1 = 2 -> (sN1'=2);
  //final state skipped
  [success<f+1>_<p>] sN1 = 3 -> (sN1'=3);
  //final state failure
  [failN1] sN1 = 4 -> (sN1'=4);
endmodule
```

Listing 10: DTMC template for node $n1$ with retry and $k =< K >$.

3.3. *Synthesis of Dependability Properties*

The specification of dependability properties in GODA is based on the idea of *probabilistic existence property* [19], that is, the probabilistic successful fulfillment of system goals, irrespective of the time. Such properties compute the probability that a system will *eventually* reach a state that satisfies a goal of interest. Therefore, the probability of fulfilling a goal G_i is defined as:

$$P_{G_i} =? \ [F \ (\phi)] \tag{1}$$

where the proposition ϕ represents the success of G_i and is recursively formed by composing the propositions of the nodes underlying G_i. For each runtime rule contained in the runtime annotation of a node n, a corresponding part of ϕ is synthesized as follows:

- Sequential or parallel fulfillment of nodes n_i and n_j:

$$\phi = succeeded_i \wedge succeeded_j$$

The case of k sequential or parallel nodes n_i is a mere extension of the aforementioned proposition.

- Optional fulfillment of nodes n_i:

$$\phi = \bigvee_{i=1}^{k} (succeeded_i \vee skipped_i),$$

where $0 < i \leq k$.

- Alternative fulfillment of nodes n_i and n_j:

$$\phi = (succeeded_i \wedge skipped_j) \vee (skipped_i \wedge succeeded_j),$$

where exactly one node should be successfully fulfilled.

- Conditional fulfillment of nodes n_i, n_j and n_k:

$$\phi = (succeeded_i \wedge succeeded_j) \vee (failed_i \wedge succeeded_k),$$

where n_j should succeed if n_i succeeds and n_k should succeed if n_i fails.

- Retry of k node instances of n_i:

$$\phi = succeeded_i$$

Note that property ϕ in this last case is the same as the one for n_i. This is due to the fact that the retries will happen on the same node while it tries to fulfill. If one of the attempts succeeds, it leads to the node's success state; it leads to the failure state if all attempts fail.

In particular, when a CRGM node n_i is constrained by a context formula C_i, its dependability property ϕ' is defined as follows:

$$\phi' = (!\, C_i \wedge skipped_i) \vee \phi \quad (2)$$

where $!\, C_i$ means that the context formula constraining n_i is not satisfied and, therefore, the fulfillment of n_i is skipped. Otherwise, C_i is satisfied and the fulfillment of n_i depends on the proposition ϕ corresponding to the runtime annotation of n_i.

Following Section 3.4 presents how the dependability property takes part into the design-time as well as runtime verification process in GODA.

3.4. Model Composition and Verification

The following sections explain the composition of a DTMC model for design-time verification via PRISM (Section 3.4.1) and parametric formulas for runtime verification (Section 3.4.2).

3.4.1. DTMC Model Composition

Since CRGM goals are ultimately realized by leaf-tasks, the overall system behavior is the composition of these tasks. Additionally, the fulfillment of CRGM goals is mapped as ϕ propositions, previously presented in Section 3.3. As such, goals states (following Figure 4) are reduced to fulfilled (if ϕ holds) or not fulfilled (otherwise). Accordingly, the design-time model checking for a goal G_i is first obtained by composing the DTMC model for all leaf-tasks that satisfy all chains of subgoals up to G_i. Then, the ϕ propositions for G_i are recursively formed by composing the propositions of the underlying subgoals of G_i.

The DTMC representation $\circledast(d_1 <, d_2, d_3 >)$ of the composed CRGM node $*(n_1 <, n_2, n_3 >)$ (following Section 3.2) is composed by the corresponding DTMCs for all leaf-tasks underlying n_1, and, if applicable, n_2, and n_3, respectively. That is, d_1, and, if applicable, d_2, and d_3 correspond to the compositions that started with leaf-tasks (bottom-up recursion of the CRGM tree). As a result, the overall CRGM DTMC may have as many PRISM modules as there are leaf-tasks underlying n_1, and, if applicable, n_2, and n_3.

Take, for example, the CRGM in Figure 10. The transformation of such CRGM renders the PRISM's DTMC in Listing 11. The CRGM DTMC is composed by four PRISM modules, one for each leaf-task. The bottom-up recursion of the CRGM tree can be noticed, for example, in *module* $G3\_T1$. Such module models the behavior of task T_1, which realizes the leftmost alternative realization of goal G3, considering G_1's alternative rule $[G3|G4]$. This alternate is expressed into XOR_G3 and XOR_G4 constants declared in lines 03/04 and used in line 9 of Listing 11. In case goal G_3 is chosen ($XOR\_G3 = 1$), a transition to running state will happen. Multiple fulfillment of G_3 as three instances of T1 ($[T1 + 3]$) in lines 19 and 20 of Listing 11.

```
1  dtmc
2
3  const int XOR_G3;
4  const int XOR_G4;
5  const double dTaskG3_T1;
6  module G3_T1
7    sG3_T1 :[0..4] init 0;
8    //init to running or skip
9    [success0_0]   sG3_T1 = 0 -> XOR_G3 : (sG3_T1
         '=1) + (1 - XOR_G3) : (sG3_T1'=3);
10   //running to final state
11   [] sG3_T1 =  1 -> dTaskG3_T1 : (sG3_T1'=2) + (1
         - dTaskG3_T1) : (sG3_T1'=4);
12   //final state success
13   [success1_0] sG3_T1 = 2 -> (sG3_T1'=2);
14   //final state skipped
15   [success1_0] sG3_T1 = 3 -> (sG3_T1'=3);
16   //final state failure
17   [failG3_T1] sG3_T1 = 4 -> (sG3_T1'=4);
18 endmodule
19 module G3_T1_S2 = G3_T1 [ sG3_T1=sG3_T1_S2,
```

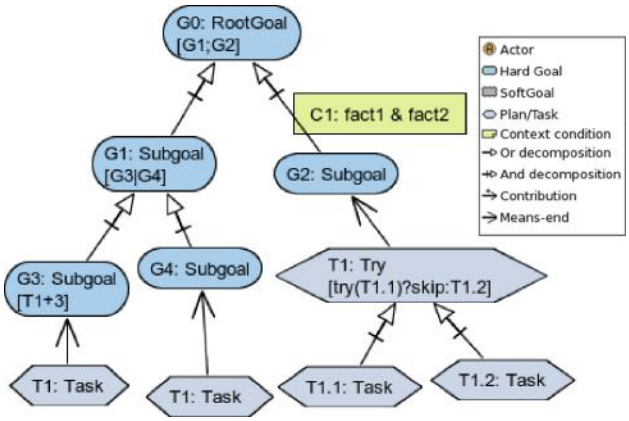


Figure 10: A CRGM example.

```
64   //init to running
65   [failG2_T1_1] (G1) &  sG2_T1_2 = 0 -> (sG2_T1_2
         '=1);
66   //try succeeded, skip running
67   [success5_0] (G1) &  sG2_T1_2 = 0 -> (sG2_T1_2
         '=3);
68   //running to final state
69   [] sG2_T1_2 =  1 -> dTaskG2_T1_2 : (sG2_T1_2
         '=2) + (1 - dTaskG2_T1_2) : (sG2_T1_2'=4);
70   //final state success
71   [success6_0] sG2_T1_2 = 2 -> (sG2_T1_2'=2);
72   //final state skipped
73   [success6_0] sG2_T1_2 = 3 -> (sG2_T1_2'=3);
74   //final state failure
75   [failG2_T1_2] sG2_T1_2 = 4 -> (sG2_T1_2'=4);
76 endmodule
77
78 formula G2 = (sG2_T1_1=2) | (sG2_T1_1=4 &
       sG2_T1_2=2) | (sG2_T1_1=3 & sG2_T1_2=3 & !(
       fact1 & fact2));
79 formula G0 = G1 & G2;
```

Listing 11: PRISM model generated for the CRGM in Figure 10.

```
     success1_0=success2_0, success0_0=success1_0
     ] endmodule
20 module G3_T1_S3 = G3_T1_S2 [ sG3_T1_S2=sG3_T1_S3,
      success2_0=success3_0, success1_0=success2_0
     ] endmodule
21
22 formula G3 = sG3_T1_S3=2 | sG3_T1_S3=3;
23
24 const double dTaskG4_T1;
25 module G4_T1
26   sG4_T1 :[0..4] init 0;
27   //init to running or skip
28   [success3_0]  sG4_T1 = 0 -> XOR_G4 : (sG4_T1
         '=1) + (1 - XOR_G4) : (sG4_T1'=3);
29   [] sG4_T1 =  1 -> dTaskG4_T1 : (sG4_T1'=2) + (1
          - dTaskG4_T1) : (sG4_T1'=4);
30   //final state success
31   [success4_0] sG4_T1 = 2 -> (sG4_T1'=2);
32   //final state skipped
33   [success4_0] sG4_T1 = 3 -> (sG4_T1'=3);
34   //final state failure
35   [failG4_T1] sG4_T1 = 4 -> (sG4_T1'=4);
36 endmodule
37
38 formula G4 = sG4_T1=2 | sG4_T1=3;
39 formula G1 = G3 | G4;
40
41 const bool fact1;
42 const bool fact2;
43
44 const double dTaskG2_T1_1;
45 module G2_T1_1
46   sG2_T1_1 :[0..4] init 0;
47   //init to running
48   [success4_0] (G1) & (fact2 & fact1) & sG2_T1_1
         = 0 -> (sG2_T1_1'=1);
49   //not selected, skipped
50   [success4_0] (G1) & !(fact2 & fact1) & sG2_T1_1
         = 0 -> (sG2_T1_1'=3);
51   //running to final state
52   [] sG2_T1_1 =  1 -> dTaskG2_T1_1 : (sG2_T1_1
         '=2) + (1 - dTaskG2_T1_1) : (sG2_T1_1'=4);
53   //final state success
54   [success5_0] sG2_T1_1 = 2 -> (sG2_T1_1'=2);
55   //final state skipped
56   [success5_0] sG2_T1_1 = 3 -> (sG2_T1_1'=3);
57   //final state failure
58   [failG2_T1_1] sG2_T1_1 = 4 -> (sG2_T1_1'=4);
59 endmodule
60
61 const double dTaskG2_T1_2;
62 module G2_T1_2
63   sG2_T1_2 :[0..4] init 0;
```

As for the probability existence property of a goal G_i, propositions ϕ_{G_i} are recursively composed according to nodes rules underlying G_i, following Section 3.3. In our approach, such propositions are represented in PRISM as a set of PRISM *formula* constructs. For example, the fulfillment of root goal G_0 in Figure 10 is a sequential fulfillment of goals G_1 and G_2 (G_1;G_2), whose proposition is the PRISM formula G_0 in line 79 of Listing 11. In turn, the fulfillment of G_1 is defined as the alternative fulfillment of G_3 and G_4 (line 39 of Listing 11). The propositions for G_3 and G_4 are defined in a similar way. Differently, the fulfillment of G_2 requires the realization of task T_1. The proposition for G_2 is in line 78 of Listing 11.

Finally, dependability of goal G_0 can be simply obtained by running the following PCTL formula:

$$P =? [F(G_0)]$$

where G_0 is the PRISM formula in line 79 of Listing 11.

3.4.2. *Parametric Formula Composition*

At design-time, we can then exploit PRISM to carry out the verification of the DTMC corresponding to the CRGM. This is usually useful to identify the most critical goals/tasks in the model, that is, those with a higher impact on its dependability through sensitivity analysis at early stages of a software system development.

However, performing such analysis at runtime would be too slow and would not scale properly —as observed in [40] and also witnessed by the experiments further presented in Section 4. While verification speed can be mitigated by the fact that probability formulas can be computed off-line, scalability of the verification process itself remains an issue.

To sort this out, we exploit both parametric model checking and the particular tree structure of goal models in a compositional way: probability formulas of smaller DTMCs corresponding to smaller goal models are computed and then composed to obtain the probability formula of a DTMC for its corresponding larger goal model. In particular, we use parametric PMC, PARAM more precisely, to analyze the DTMCs corresponding to behavioral models according to the transformation explained in Section 3.2. These models are pretty small and the resulting formulas can be computed —and recomputed—quite easily. Table 2 presents them, together with their corresponding runtime rules. Note that, on the first column, n, $n1$ and $n2$ represent subtrees, whereas, on the second column, N, $N1$ and $N2$ represent their respective symbolic parametric formula to estimate the dependability of the respective sub-trees. These formulas are generated once in PARAM and they represent the formulas for the diverse behavior models.

Runtime Rule	**Symbolic formula**
Sequential AND (n1;n2)	$N1 * N2$
Sequential OR (n1;n2)	$MAX(N1, N2)$
Parallel AND (n1#n2)	$N1 * N2$
Parallel OR (n1#n2)	$MAX(N1, N2)$
Optional (n)	$X_n * N - X_n + 1$
Alternative n1|n2	$X_{n1} * N1 - X_{n1} * X_{n2} * N2 + X_{n2} * N2$
Try (n)?n1:n2	$N * N1 - N * N2 + N2$
Cardinality (n+k)	$(N)^k$
Cardinality (n#k)	$(N)^k$
Retry (n@k)	$1 - (1 - N)^{k+1}$

Table 2: Symbolic parametric formulas for their corresponding CRGM runtime rules.

Sequential and *Parallel* compositions are rendered into the same formula. This comes from the fact that the order with which the dependability of parallel tasks is computed should not matter for computing the formula. Therefore, given the quantitative nature of the dependability estimate, operators ; and # are commutative. Note also that the symbolic formulas for *Optional* and *Alternative* compositions yield a variable X that can be either 1 (true) or 0 (false) to render the actual choice. Finally, the n-ary MAX operator for OR-decomposition computes the highest dependability value among its operands.

The formulas of Table 2 define the dependability of a CRGM model recursively on the structure of this model. Therefore, these formulas define building blocks so that atomic formulas are then composed for obtaining the parametric formula for the goal model by rewriting the formula associated with this model in terms of the formulas associated with its sub-trees. The rewriting terminates when it reaches leaf tasks, in which case the formula cannot be further rewritten.

Algorithm 1 presents the compositional approach to build the parametric formulas. Such an algorithm follows a recursive depth-first strategy to visit both the tree structure of the goal model and the syntactic tree built by parsing the runtime annotations in the nodes. The algorithm starts from *node*, which is the root node of the model for which the parametric formula is to be built, and $perRuleForms$, which is a map structure that contains the correspondences presented in Table 2. Starting from *node*, Line 1 stores its children nodes in a list named $decNodes$. Line 2 fetches $decType$, that is, the decomposition type—either AND or OR—of *node*. Line 3 uses *rtAnnot* to store the fetched runtime annotation of *node*. Line 4 calls function $getForm$, which returns the parametric formula for runtime annotation *rtAnnot* and decomposition type $decType$ of that *node* in $nodeForm$.

A syntactic tree composed of runtime rules operators and operands is built using a grammar parser over *rtAnnot*. Each rule operator in the tree is parsed into its corresponding symbolic formula in $perRuleForms$. Each operand, i.e., symbol, is parsed as follows: goals or tasks into their ID, nested operators into the result of a call to $getForm$ to fetch their own symbolic formulae (refer to the definition of *rtAnnot* in Sec. 3.1). The right formula is obtained by applying the correspondences of Table 2.

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 5 to 9). Finally, Line 10 returns the symbolic parametric formula for the CRGM model whose root is *node*.

Figure 11 exemplifies how the algorithm works for the creation of the formula associated with a model having root goal G_1 realized by task T_1, which in turn is realized by the annotation $Try(T_{1.1}\%2)?\,T_{1.2} : T_{1.3}$, that is, if one of the two execution retries of $T_{1.1}$ succeeds, then execute $T_{1.2}$, otherwise execute $T_{1.3}$. Note the dual recursion (performed by $composeNodeForm$ and $getForm$, respectively): one over the goal tree (Figure 11a) and the other one over the syntactic tree representing the runtime annotation for T_1, where a retry ($T_{1.1}$@2) is nested into a *try* (Figure 11b).

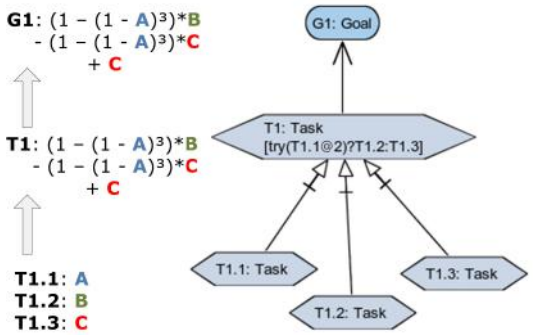
This algorithm is a key enabler towards an affordable time-space runtime analysis. It allows one to perform the model checking of independent and smaller

Algorithm 1: composeNodeForm(Node node, Map perRuleForms)

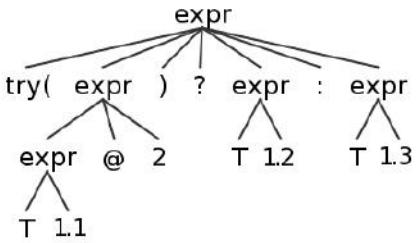
Input: A node, either a root or a local goal
Result: Symbolic Parametric formula of the node

```
1  List [] decNodes ← getDecomposition(node);
2  DecType decType ← getDecType(node);
3  String rtAnnot ← getRuntimeRule(node);
4  String nodeForm ← getForm(decType, rtAnnot,
   perRuleForms);
5  foreach subNode in decNodes do
6      String subNodeId ← getId(subNode);
7      String subNodeForm ←
       composeNodeForm(subNode, perRuleForms);
8      replaceSubForm(nodeForm,subNodeForm,subNodeId);

9      replaceSubForm(nodeForm, subForm, subNodeId);
10 end
11 return nodeForm;
```



(a) CRGM tree.



(b) Syntactic tree of the runtime annotation of T_1.

Figure 11: Example application of Algortihm 1.

parts of the system—that is, the behavior of each behavior model— and compose these pre-computed results, thus avoiding combinatorial state explosion problems. It requires significantly smaller formulas than the parametric model checking of the whole CRGM through a single DTMC, and does it in significantly shorter time. Otherwise, long delays and memory consumption could make the approach prohibitive in many cases. The scalability of the approach is analyzed in Section 4.

The computation of obtained parametric formulas quantifies the dependability of the given combinations of goals/tasks in the CRGM. Accordingly, one can use such formulas to evaluate the dependability of the variants specified in the CRGM at runtime.

Since the focus of this work is on the runtime analysis, we reasonably assume the existence of other layers/components in charge of detecting and monitoring context information, which is then used to feed the parametric formulas evaluated at runtime to estimate the dependability of the system. We also assume the monitorability of the execution of individual leaf-tasks. As such, the dependability of leaf-tasks can be estimated based on logged data, following, for example, a mean-time between failures (MTBF) approach. As soon as this information is available, GODA is ready to carry out the analysis.

3.5. Tool support

The manual translation of Context and Runtime Goal Models into DTMC models could be a tedious, costly, and error-prone activity. The absence of significant and automated tool support would definitively hamper the usability of the proposed solution. The purpose of the GODA framework is thus to support the formal verification by hiding the probabilistic modeling to reduce the overhead and errors caused by the manual generation of the verification model. GODA automates the generation of the DTMC models rendered into PRISM/PARAM.

The modeling and transformation environments provided by GODA extend an existing open source tool for TROPOS called TAOM4E [36]. TAOM4E provides a graphical environment for goal modeling based on the Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF). It also supports model-driven code generation. The CRGM-to-DTMC code generator has been implemented as an Eclipse plug-in in Java and integrated in TAOM4E[5].

Figure 12 presents a high-level view of the GODA framework. It starts from a CRGM file in the TAOM4E format and generates the PRISM/PARAM verification models for the proposed dependability analysis. Two dedicated plug-ins generate the PRISM and PARAM models, respectively.

[5]The GODA framework is available at `https://github.com/lesunb/CRGMToPRISM`.

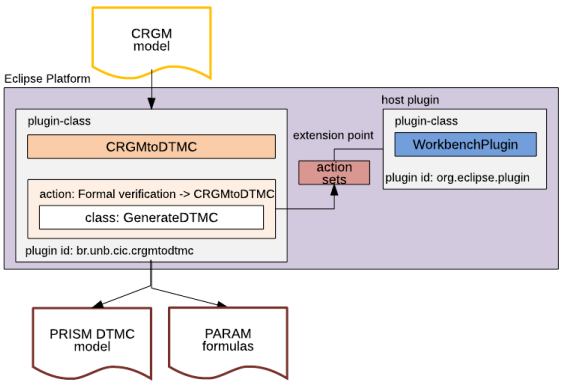


Figure 12: High-level view of the GODA framework.

4. Evaluation

This section describes the evaluation of GODA and its supporting framework on Mobee, a real-life software system that allows people to share live and updated information about public transport via mobile devices. Mobee has been running for more than a year and has already more than three thousand users. Additionally, we evaluate GODA on its time-space scalability to support design time decisions and runtime dependability analysis by considering a comprehensive set of behavior models that can be embedded in a CRGM.

4.1. Mobee

Mobee has been conceived as a platform for sharing public transport information among users. The basic idea is to empower users with the access to information regarding how to reach a destination from a given location, either in advance or live through a mobile application. Users are not only the receivers of information, but also the providers: they can feed the system with new bus lines and bus stops that have not been mapped before or they can send suggestions about modifications of those assets. Also, the automatic monitoring of the location of a user's device enables the tracking of the transport in use. The project started in early 2013 and has been deployed to production and available for use since November of the same year as a web application. Mobile versions for both Android and iOS systems have been released in early 2014 and they have achieved three thousand users each [34].

Mobee, as many applications running on a mobile device, is designed to be sensitive to its dynamic context including battery, availability of GPS signal, and other spatio-temporal aspects. Some of these elements impact the dependability of the application and on its ability to meet user requirements. Figure 13 presents the overall system architecture and Figure 14 presents one of the user interfaces used for displaying the itinerary of a bus line over a city map.

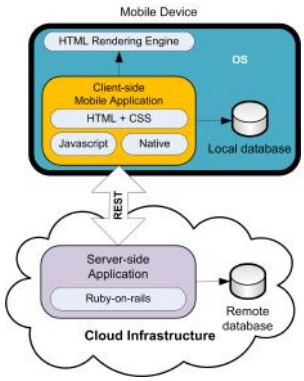


Figure 13: Mobee Architecture.

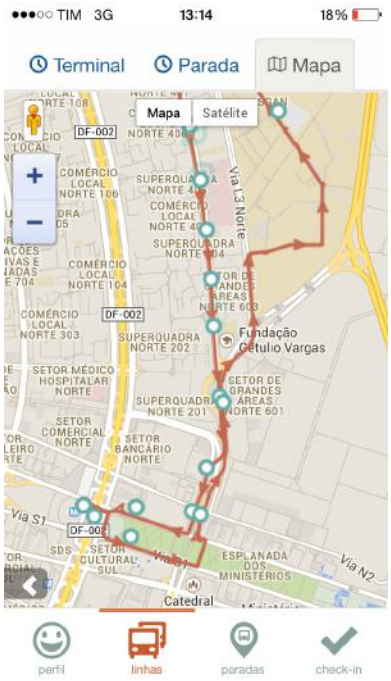


Figure 14: Mobee - Example GUI.

The context conditions related to Mobee are listed in Table 3. Each context condition has a unique identifier (CID), a description, and the corresponding Boolean expression (context facts). Column *Affected Element* indicates the goals or tasks activated/restricted by the context condition.

CID	Description	Boolean expression	Affected element
C1	Minimum battery level	$BATTERY >= 15$	G_{10}
C2	GPS	GPS	$T_{1.11}$ (G_{10})
C3	Minimum storage level	$STORAGE > 10$	$T_{1.33}$ (G_6)

Table 3: Contexts conditions affecting the behavior of Mobee.

The system has been modeled as the CRGM presented in Figure 15. This model includes all the main goals of the current system and the tasks responsible for fulfilling them. Despite the reverse engineering approach used to elicit goals and tasks, the use of GODA at design-time has been useful to analyze the dependability of the system given a certain context of operation and the dependability of leaf-tasks. GODA can then be useful to clarify and document dependability aspects that are usually left implicit.

4.2. Goal Question Metric

The design of our empirical study follows the Goal Question Metric (GQM) method, which provides a systematic structure to guide the evaluation process [9].

Our first evaluation goal G1 aims to perform the dependability assessment of Mobee while varying through a controlled experiment the contexts that affect its operation, that is, connectivity, precision of geolocation as well as battery and storage capabilities, as presented in Table 3. Such an evaluation required the definition of the following questions and metrics:

Q1.1: Is GODA a feasible approach to provide valuable strategies to conduct the dependability assessment of Mobee while varying its contexts of operation?

- M1.1: Estimated sensitivity analysis of Mobee dependability.

Q1.2: Is GODA a feasible approach for the dependability assessment of Mobee at *runtime* while varying its contexts of operation?

- M1.2: Time needed for evaluating the dependability of Mobee at runtime.

Since the CRGM defined for Mobee does not allow for the experimentation of all behavior models, our second evaluation goal G2 aims to provide a more comprehensive time-space scalability analysis of GODA. As such, we defined the following questions and metrics:

Q2.1: What is the space complexity of the automated generation of DTMC models?

- M2.1a: The size of the file of the DTMC model.
- M2.1b: The number of states in the model.
- M2.1c: The number of transitions in the model.

Q2.2: What is the time complexity to create the verification structure?

- M2.2a: The time consumed to model check the generated DTMC.
- M2.2b: The time consumed to build the parametric formula.

Q2.3: What is the time-space complexity of the runtime verification?

- M2.3a: The size of the generated CRGM parametric formula.
- M2.3b: The time consumed to evaluate the parametric formula.

Conducted experiments used version 4.0.3-linux64 of PRISM and version 2-2-64α of PARAM on a personal computer with dual-core/4-threads Intel i5 2.40GHz processor, 8GB DDR3-800 1066 MHz memory, and Linux Ubuntu 14.04 as operating system. The evaluation of the parametric formulas as part of the runtime analysis is based on a 2GB memory and Dual-core 1.7 GHz Krait 300 processor running Android 4.4.

4.3. Feasibility Assessment

At design-time, we used PRISM to reason about the dependability of the system. As no measurements that refer to individual leaf-tasks exist at this time, we have conducted experiments with dependability values varying within a reasonable range, and considered different aspects related to Mobee contexts of operation to evaluate alternative design decisions.

At runtime, our dependability analysis is based on PARAM and the use of a parametric formula that states the probability of fulfilling system goals given the monitored values of the reliability of leaf-tasks. This way, one can evaluate any variation of the system and validate stated requirements. It could also help decide the alternative solutions that should be used to fulfill the different system goals.

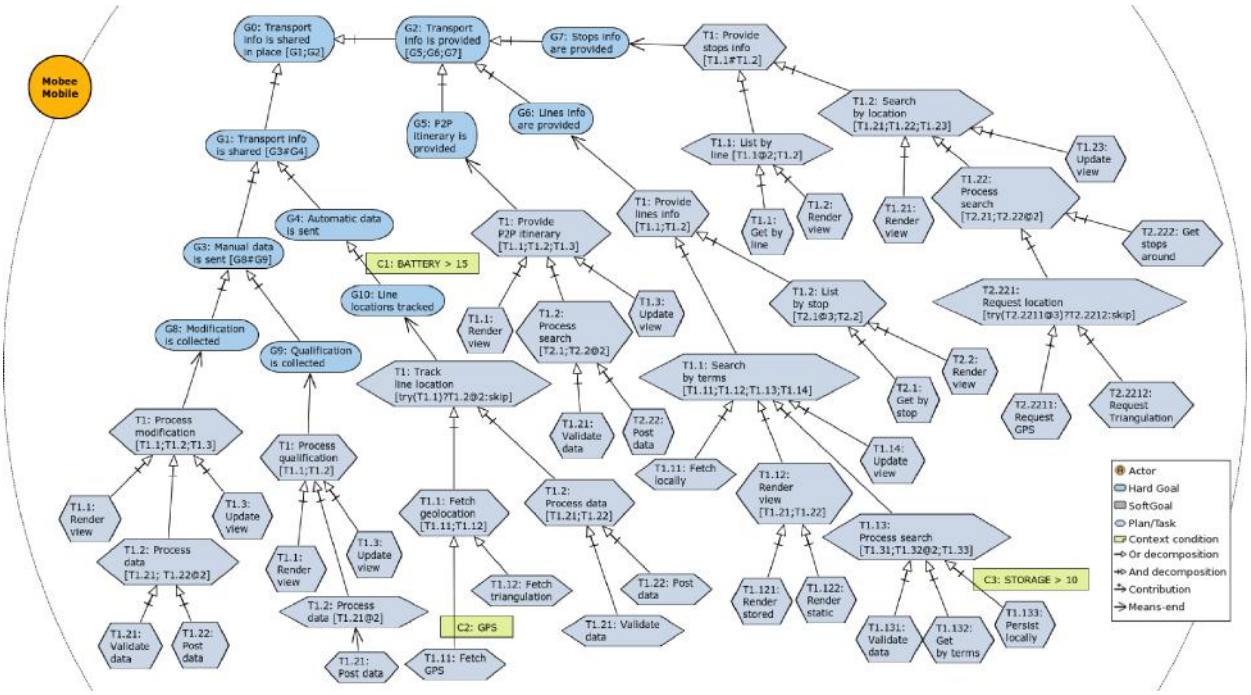


Figure 15: The complete CRGM of Mobee Mobile system

4.3.1. Design-time Contextual dependability Analysis

As for question **Q1.1**, PRISM was used for running the experiments in which the values of the constants in the model vary within set ranges. The important aspects related to the dependability of Mobee were analyzed and produced valuable results for designing the system (metric M1.1).

Connectivity: The service provided by mobile data providers is still subject to signal quality degradation and even unavailability. By varying the dependability of connection-dependent leaf-tasks, their impact on the global dependability was estimated for different numbers of retries. The results guided the selection of the appropriate number of retries. Figure 16 presents the sensitivity analysis for a connectivity-dependent leaf-task, and we can clearly see the positive effect of a single retry. With more than two retries —with an average dependability of the leaf-task as low as 87.5%— there is no additional significant gain in the global dependability of Mobee. This justifies our choice of maximum two retries for connection-dependent tasks.

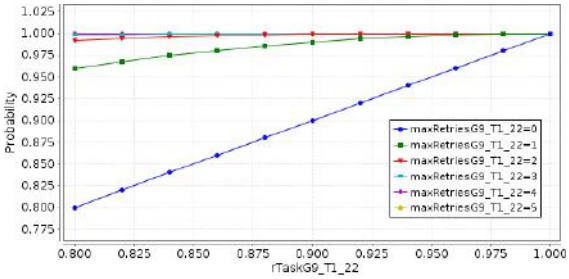


Figure 16: Dependability with different numbers of retries while varying task $T_{1.22}$ (goal G_9).

Precision of geolocation: To provide accurate information about the different buses, only GPS-based solutions are accepted as a means to fulfill the goal that says that 'line location is traced'. As GPS signal may become unavailable, this goal could be deactivated instead of resulting in a failure. A retry rule is also used to allow for multiple attempts in case of failures in retrieving accurate geolocation from GPS. Figure 17 presents the results for task 'request GPS location'. The chart shows that three retries can be considered an appropriate choice.

Battery and storage: The system has been designed to persist locally the data of searched bus lines to reduce the amount of exchanged data. Given the vast number of bus lines in a big city, this task is only activated when the storage available on the mobile device is greater than 10% of its capacity before starting retrieving data from the server. This is clearly to avoid failures caused by the unavailability of storage on the device. Moreover, a battery level below 15% imposes a restriction on goal 'line

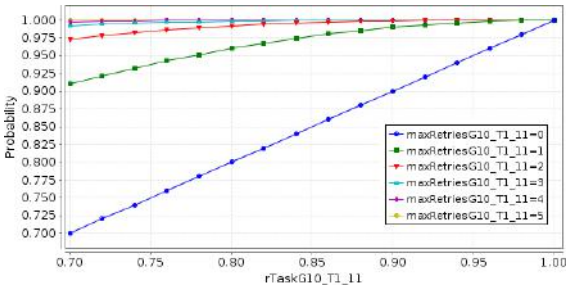


Figure 17: Dependability with different numbers of retries while varying task $T_{1.11}$ (goal G_{10}).

C1	C2	C3	Evaluation Time (ms)
true	true	true	22.019291000
true	true	false	12.011298000
true	false	true	19.344944000
true	false	false	13.807750000
false	true	true	15.942896000
false	true	false	13.744944000
false	false	true	13.743191000
false	false	false	12.383381000

Table 4: Parametric formulas evaluation time in different contexts of Mobee's operation (M1.2)

locations traced' due to the high power consumption of the GPS module. The battery and storage analyses were performed based on domain knowledge. Note that, at design time, the evaluation of this goal carried out through PRISM is more a qualitative analysis (yes/no) rather than a quantitative dependability assessment per se.

4.3.2. *Runtime Contextual Dependability Analysis*

To assess the feasibility of using GODA as runtime solution for estimating the dependability of Mobee (question **Q1.2**), we created a computable parametric formula that represents the probability of fulfilling the root goal of the system for different contexts of operation. This formula includes parameters for the dependability of leaf-tasks and for the selection of optional and alternative tasks. We used the algorithm presented in Section 3.4.2 to generate the formula, optimize its size, and thus reduce the verification time.

Since the focus of this experiment is on the runtime analysis, we reasonably assume the existence of other layers/components responsible for detecting and monitoring context information. According to retrieved context information, the parametric formula is evaluated at runtime to estimate the dependability of the root goal of the system. We also assume the monitorability of the execution of individual leaf-tasks. As such, the reliability of leaf-tasks is estimated within a fixed time window by calculating its mean-time between failures from its previous failures registered in a history log. As soon as this information is available at runtime, GODA is ready to start the analysis phase. As for Mobee, the analyses were performed by evaluating the formulas on an Android phone. Such formulas required less than 1Kbytes of storage.

Table 4 presents the results for metric M1.2 obtained by measuring, on the Android version of Mobee, the time to evaluate the each of formulae corresponding to different contexts of operation, as in this work contexts are not parametrized, thus each context combination has a parametric formula. The magnitude of the results is compatible with the initial requirements set for Mobee as far as performance is concerned.

4.4. *Scalability Assessment*

The second goal of our evaluation was the assessment of the actual scalability of GODA with respect to all the behavior models defined for CRGMs. Since Mobee does not require all the behaviors, we decided to create generic CRGMs with a growing number of leaf-tasks for different behaviors. This is to obtain a clear understanding of the impact each behavior has on the metrics of interest.

As for question **Q2.1**, the size of the textual DTMC model in PRISM (M2.1a) grows linearly with respect to the number of leaf-tasks as each leaf-task is mapped onto a single PRISM module. This result is illustrated in Figure 18.

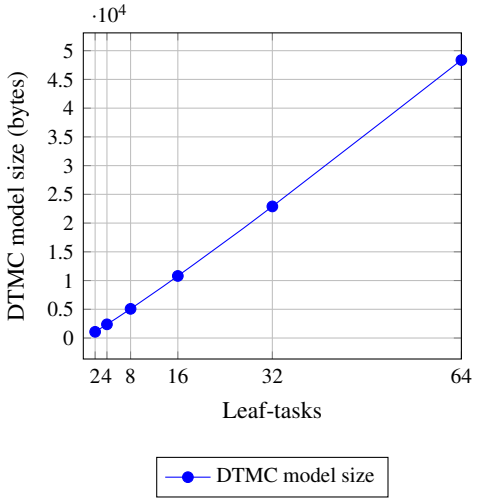


Figure 18: Size of the textual DTMC model with respect to the increasing number of leaf-tasks (M2.1a).

The plots of Figures 19a and 19b say that the number of states (M2.1b) and transitions (M2.1c) of the resulting DTMC grows exponentially with the number of leaf-tasks, with the exception of sequential leaf-tasks. In contrast to interleaved ones, sequential tasks only have a single order for state transitions to happen. Moreover, with sequential tasks we also have a single path from the initial state to the final, successful state, while the other behavior models add multiple possible paths causing the number of states and transitions to grow exponentially.

The time complexity **Q2.2**, that is, the time required by PRISM to model check the generated DTMC (M2.2a) is shown in Figure 19c. As can be seen, the verification time grows exponentially for all the rules.

At runtime GODA applies the solution presented in Section 3.4.2. The results for the generation time of the parametric formula (M2.2b), its size (M2.3a) and its evaluation time (M2.3b) are shown in Figures 20a, 20b, and 20c, respectively. Results show that GODA allows for orders of magnitude improvement compared to previous results with PRISM, in accordance with the arguments presented in Section 3.4.2, namely the isolated offline verification of behavior models to generate symbolic formulae used later by Algorithm 1 and the commutativity of sequential and interleaved behaviors. Accordingly, the results for sequential and interleaved results are merged in Figures 20a, 20b, and 20c.

In conclusion, obtained results show that while design-time verification in PRISM is an expensive process, our runtime verification seems to be very affordable: for goal models up to two thousand leaf-tasks, formula generation time, performed off line, remained under 4.5 seconds (M2.2b), requiring less than 240 kbytes (M2.3a) of storage in the worst-case scenario. Finally, the runtime verification time remained under 35ms (M2.3b). As a result, we can conclude that GODA is scalable for a size up to few thousands of leaf tasks at runtime. Such scalability highlights the efficiency of our approach to optimize runtime model checking in the context of goal modeling. As a future follow-up work to provide a more scalable approach for design time verification in PRISM, a step-wise approach could be performed by following the principles of partial order reduction. As such, an analysis based on the branches of the CRMG could be iteratively performed, as long as those branches do not share common (or mutually dependent) behaviors or information.

4.5. Threats to Validity

Construct validity: The mapping of leaf-tasks representing operations of the system on their concrete counterparts in the implemented system is a common and well-known problem of the requirements engineering community. We assume that leaf-tasks can be traced back to software operations and that the dependability of the former ones represents the probability of a successful execution of the later elements.

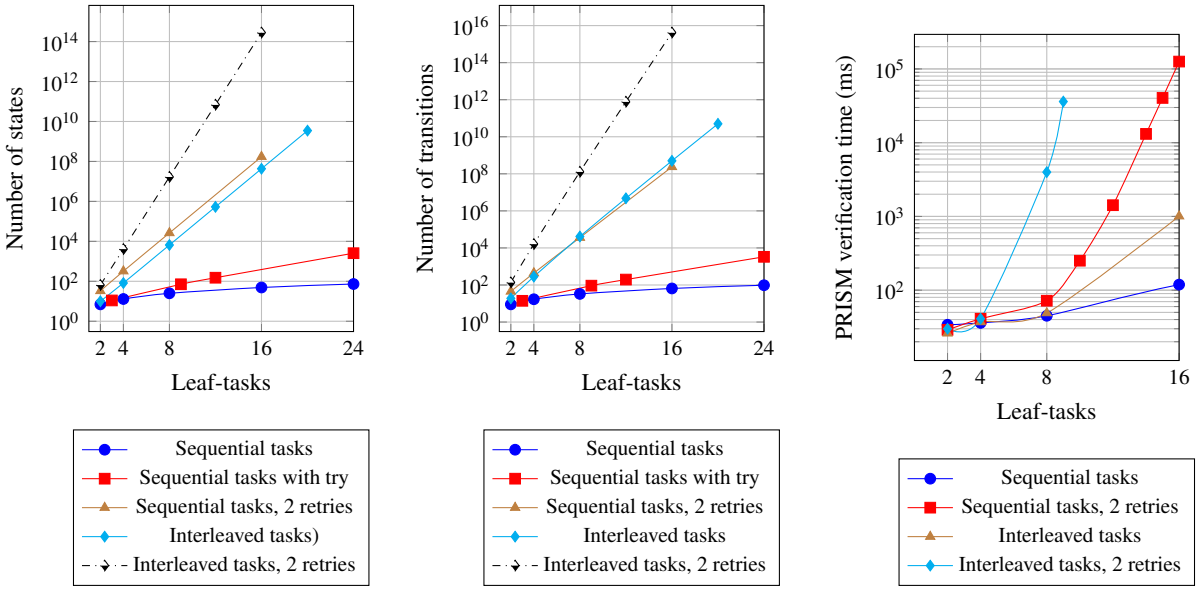
Internal validity: The suitability of GODA for reasoning about Mobee has been presented. The dependability analysis at design-time showed how fault-tolerant aspects can be better designed in Mobee; at runtime, it also showed the low overhead the analysis takes.

External validity: We performed the evaluation of our framework in the context of Mobee and further performed a scalability analysis with GODA through question metrics M2.1a to M2.3b. All the behaviors a CRGM can accommodate were considered, without taking into account a particular domain. It is most likely the performance results are compatible with many kinds of systems where the analysis overhead must be constrained to low bounds. However, to empirically validate such statement, assessment on the suitability of the GODA framework on other real-life systems has still to be performed.

5. Related Work

Goal models have been extensively used in requirements engineering to elicit, model, and analyze stakeholders' requirements [14, 46, 47, 48, 45, 10]. The majority of goal-oriented requirements engineering solutions share a common set of conceptual elements and differ in the syntax/notation used to render them. They also target specific categories of requirements: for example security [32, 37], trust [39] and risk [4]. The fact that GODA uses Tropos does not compromise its generality as the concepts and analyses proposed are built around the core concepts of GORE. The ability to reuse TAOM4E, a Troops-based open source tool, enabled us to enrich a tested and stable machinery and effectively implement our GODA framework as extension to a well-known solution.

Dalpiaz et al. [13] proposed a conceptual distinction between static Goal Model, called Design Goal Model (DGM) , and the Runtime Goal Model (RGM), which extends DGM with additional state, behavioral, and historical information about the fulfillment of goals. GODA has benefited from the RGM in the sense that it provides a high-level description of a system behavior. In GODA, RGM leaf-tasks are mapped onto a probabilistic verification model that preserves the behavior semantics inherited from the RGM. Our work also lever-

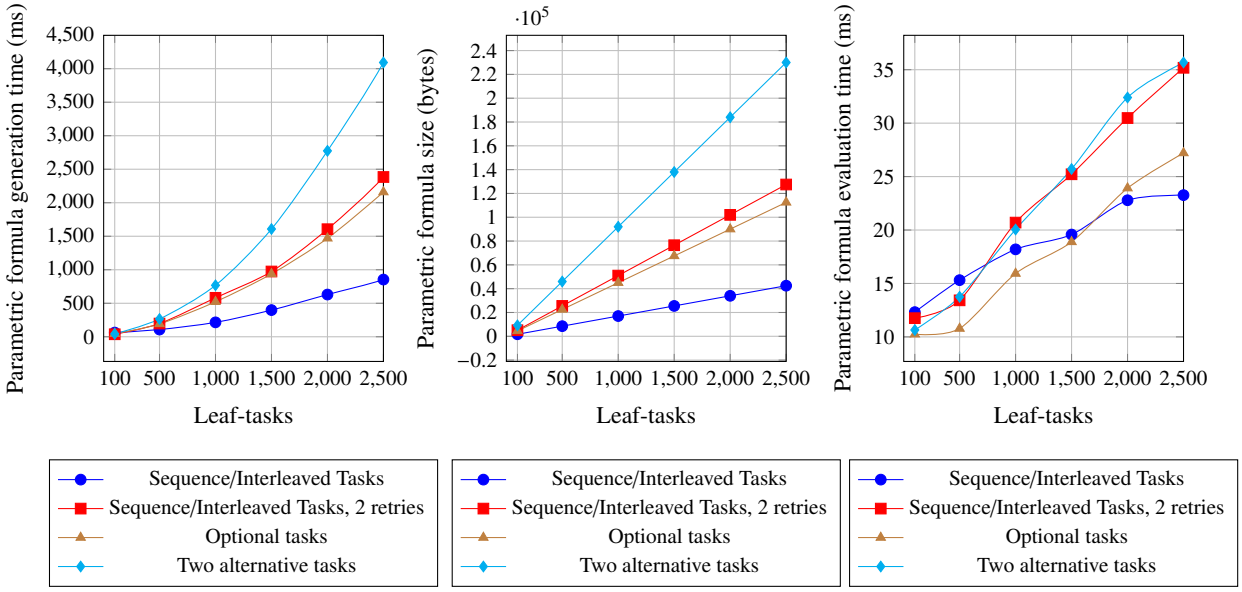


(a) Number of DTMC states (M2.1b).

(b) Number of DTMC transitions (M2.1c).

(c) Verification time (M2.2a).

Figure 19: Results for design-time verification metrics M2.1b, M2.1c and M2.2a.



(a) Generation time of the parametric formula (M2.2b).

(b) Size of the parametric formula (M2.3a).

(c) Evaluation time of parametric formula (M2.3b).

Figure 20: Results for runtime verification metrics M2.2b, M2.3a and M2.3b.

ages the RGM by taking into account contextual facts, which are paramount in a real runtime environment.

GODA also exploits formal methods to maximize rigor and enable automated reasoning. KAOS is a goal-oriented requirements engineering methodology that offers a formal language that combines semantic nets and linear-time temporal logic. Formal Tropos (FT) [16] is motivated by translating Tropos models to a formal representation to enable consistency verification in the different strategies to achieve goals. The work by Ali et al [2] only considers whether context specifications over goals are consistent and whether tasks and operations require that contradictory context changes be made. The formalization of the context and probability facets of goal achievement was limitedly researched. GODA considers these two facets for analyzing dependability requirements and extends the basic model checking proposed in KAOS and FT, for example, for the consistency of the model and for reasoning about the fulfillment of goals and soft goals [17].

GODA is meant to estimate the dependability of the different strategies for goal fulfillment as preliminary step for deciding the one to take. This enables informed decisions at both design- and run-time. Similarly, Souza et al. [42] proposed AwReqs as a class of meta-requirements in a goal model, that is, to specify the success/failure rate of other requirements in the model, including goals, tasks, domain assumptions, and even other AwReqs meta-requirements. Their objective is to enrich the original goal model with constraints on the performance of the system and criteria on its self-adaptation capabilities —as runtime AwReqs violations should be addressed by corrective actions. In contrast, GODA tackles the contextual and probabilistic facets of possible failures and provides reasoning capabilities that could precede the decision making done via AwReqs and adds information that could enrich it further (e.g. the probabilities of success before actual failures occur).

Mohammadi et al. [35] proposed an iterative and recursive approach to bridge the gap between the knowledge of the context and the system purposes. While our approach focuses on the quantitative aspects of dependability analysis, that approach can be quite useful in conjunction with ours so as to refine the goal model strategies in the presence of contexts that not only constrain the goal realization but also also compromises the system dependability.

Baresi et al. [7, 8] express and assess the degree of goal fulfillment by exploiting fuzziness with the idea of preventing violations and tolerating some deviations. They allow for the selection and tuning of adaptation strategies. While "crisp" goals are defined in LTL, "fuzzy" goals use a fuzzy temporal language they specified. In our work, we focus on the probabilistic nature of goal fulfillment as we advocate the success reachability of a system is a probabilistic measure. In addition, we explicitly considered context and runtime annotations in the goal-model so that adaptation strategies may be considered as variation points, specially when dependability is a major concern.

Cailliau and Lamsweerde [11] propose a probabilistic framework for goal specification and obstacle/risk assessment. They provide a specification language for goals and obstacles with a probabilistic layer. Their work can be compared to ours to the extent they envision the benefits of a sound quantitative goal-modeling analysis structure. While our work focuses on helping the system to self-adapt at runtime by analyzing the different alternatives, this is not a direction they target their contribution. As a result, their work does not take into account context modelling structure and neither consider such structure as part of the provided risk assessment. Finally, there is no evidence of automated analysis of the proposed methodology and no evidence of its suitability for runtime assessment. As the authors clearly state on their paper, the analysis by using markov-based model checking was an open issue not addressed in their work.

The work closest in nature to our approach is that by Letier et al. [31, 30]. They present techniques for specifying partial degrees of goal fulfillment and for quantifying the impact of alternative system designs on the degree of goal fulfillment. From the perspective of quantitative analysis, their approach is close to ours in the sense that they enrich goal refinement models with a probabilistic layer for reasoning about partial fulfillment. They also derived a model that can be used for the formal analysis and animation of KAOS operation models in LTSA [38]. In our work we leverage such an approach by taking into account runtime aspects that a GORE model can embrace, where dependability is our first-class citizen. Also, our work takes into account the complex implications that context may impose on the partial and global realization of goals. This is central to our contribution since the overall goal fulfillment may not only be affected but also be refrained from being achieved due to context restrictions.

6. Conclusions and Future Work

This paper proposes the Goal Oriented Dependability Analysis (GODA) framework to help experts assess dependability at both design- and run-time, and where

different contexts may take place. GODA allows for the estimation of the dependability of the different strategies to reach goals, and thus it can be seen as enabler for the self-adaptation of systems with respect to context changes. We implemented GODA by extending TAOM4E and made it available to be further explored by the GORE community. We evaluated our framework on Mobee, a real-life software system that allows people to share live and updated information about public transport via mobile devices. Our results show that GODA enables contextual and dependability-driven analyses at runtime even under limited computational resources. We also performed a comprehensive simulation to carry out a time-space scalability analysis, where results overcame our expectations for probabilistic runtime verification based on a parametric approach.

As future work, we envision the integration of GODA into a complete self-adaptation loop and its use on other real-life software systems that require self-adaptation strategies, potentially allowing for multi-agents modelling, specially towards dependability assurance at runtime. In addition, from the software design perspective, we plan to elicit the traceability of CRGM tasks on corresponding software components at both architectural and implementation levels. The major purpose behind this goal/task-component traceability is to be able to monitor the dependability of GODA tasks at runtime.

Probability values are not a limitation of our work, since they are parameters and can be fed into the model once these information become readily available. As previously stated, ways to obtain the probability values are various ones such as computing the system's MTTF or following a runtime computation approach by Su et al. [43]. Although the monitoring and update procedure of probability values at runtime is out of scope of this work, a very suitable approach that could be incorporated into GODA in the future is the proposal by Grunske [19]. Finally, we plan to extend our GODA framework to further improve the design-time evaluation by supporting a cost-based analysis to achieve a solution that balances dependability and use of system resources, since these are often conflicting requirements.

References

[1] Ali, R., Dalpiaz, F., Giorgini, P., 2010. A goal-based framework for contextual requirements modeling and analysis. Requirements Engineering 15 (4), 439–458.

[2] Ali, R., Dalpiaz, F., Giorgini, P., 2013. Reasoning with contextual requirements: Detecting inconsistency and conflicts. Information and Software Technology 55 (1), 35–57.

[3] Ali, R., Dalpiaz, F., Giorgini, P., 2014. Requirements-driven deployment. Software & Systems Modeling 13 (1), 433–456.

[4] Asnar, Y., Massacci, F., 2011. A method for security governance, risk, and compliance (GRC): A goal-process approach. In: Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures. Lecture Notes in Computer Science. Springer, pp. 152–184.

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

[6] Baier, C., Katoen, J.-P., 2008. Principles of Model Checking (Representation and Mind Series). The MIT Press.

[7] Baresi, L., Pasquale, L., 2010. Adaptive Goals for Self-Adaptive Service Compositions. In: 2010 IEEE International Conference on Web Services. IEEE, pp. 353–360.

[8] Baresi, L., Pasquale, L., Spoletini, P., 2010. Fuzzy Goals for Requirements-Driven Adaptation. In: 2010 18th IEEE International Requirements Engineering Conference. IEEE, pp. 125–134.

[9] Basili, V. R., Caldiera, G., Rombach, H. D., 1994. The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley.

[10] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J., 2004. Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8 (3), 203–236.

[11] Cailliau, A., Lamsweerde, A., 2013. Assessing requirements-related risks through probabilistic goals and obstacles. Requirements Engineering 18 (2), 129–146.

[12] Clarke, E. M., Emerson, E. A., Sistla, A. P., 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8 (2), 244–263.

[13] Dalpiaz, F., Borgida, A., Horkoff, J., Mylopoulos, J., May 2013. Runtime goal models: Keynote. In: IEEE Seventh International Conference on Research Challenges in Information Science (RCIS). pp. 1–11.

[14] Dardenne, A., van Lamsweerde, A., Fickas, S., 1993. Goal-directed requirements acquisition. Science of Computer Programming 20 (1), 3–50.

[15] Filieri, A., Ghezzi, C., Tamburrelli, G., May 2011. Run-time efficient probabilistic model checking. In: Software Engineering (ICSE), 2011 33rd International Conference on. pp. 341–350.

[16] Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P., May 2004. Specifying and analyzing early requirements in tropos. Requirements Engineering 9 (2), 132–150.

[17] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R., 2003. Reasoning with goal models. In: Spaccapietra, S., March, S., Kambayashi, Y. (Eds.), Conceptual Modeling, ER 2002. Vol. 2503 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 167–181.

[18] Grunske, L., 2008. Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering. ICSE '08. ACM, New York, NY, USA, pp. 31–40.

[19] Grunske, L., 2011. An effective sequential statistical test for probabilistic monitoring. Information and Software Technology 53 (3), 190 – 199.

[20] Hahn, E., Hermanns, H., Zhang, L., 2011. Probabilistic reachability for parametric markov models. International Journal on Software Tools for Technology Transfer, STTT 13 (1), 3–19.

[21] Hahn, E. M., Han, T., Zhang, L., 2011. Synthesis for pctl in parametric markov decision processes. In: Proc. 3rd NASA Formal Methods Symposium, NFM'11. Vol. 6617 of LNCS. Springer.

[22] Hahn, E. M., Hermanns, H., Wachter, B., Zhang, L., 2010.

Param: A model checker for parametric markov models. In: CAV. pp. 660–664.

[23] Hahn, E. M., Hermanns, H., Zhang, L., Wachter, B., 2015. PARAM language manual. http://depend.cs.uni-sb.de/tools/param/manual, [Online; accessed 25-august-2015].

[24] Hansson, H., Jonsson, B., 1994. A logic for reasoning about time and reliability. Formal Aspects of Computing 6 (5), 512–535.

[25] Kwiatkowska, M., Norman, G., Parker, D., 2009. Prism: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Performance Evaluation Review 36 (4), 40–45.

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

[27] Laboratory, O. U. C., 2015. PRISM case studies. http://www.prismmodelchecker.org/casestudies/, [Online; accessed 25-august-2015].

[28] Laboratory, O. U. C., 2015. PRISM language manual. http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction, [Online; accessed 25-august-2015].

[29] Laboratory, O. U. C., 2015. PRISM web site. http://www.prismmodelchecker.org/, [Online; accessed 25-august-2015].

[30] Letier, E., Kramer, J., Magee, J., Uchitel, S., 2008. Deriving event-based transition systems from goal-oriented requirements models. Automated Software Engg. 15 (2), 175–206.

[31] Letier, E., van Lamsweerde, A., 2004. Reasoning about partial goal satisfaction for requirements and design engineering. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering. SIGSOFT '04/FSE-12. ACM, New York, NY, USA, pp. 53–62.

[32] Massacci, F., Mylopoulos, J., Zannone, N., 2007. Computer-aided support for secure tropos. Autom. Softw. Eng. 14 (3), 341–364.

[33] Mendonça, D. F., Ali, R., Rodrigues, G. N., 2014. Modelling and analysing contextual failures for dependability requirements. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2014. ACM, New York, NY, USA, pp. 55–64.

[34] Mobee, 2015. Mobee: App for Android in Google Play. https://play.google.com/store/apps/details?id=app.mobee, [Online; accessed 3-august-2015].

[35] Mohammadi, N. G., Alebrahim, A., Weyer, T., Heisel, M., Pohl, K., 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, Ch. A Framework for Combining Problem Frames and Goal Models to Support Context Analysis during Requirements Engineering, pp. 272–288.

[36] Morandini, M., Nguyen, D. C., Perini, A., Siena, A., Susi, A., 2008. Tool-supported development with tropos: The conference management system case study. In: Proceedings of the 8th International Conference on Agent-oriented Software Engineering VIII. AOSE'07. Springer-Verlag, Berlin, Heidelberg, pp. 182–196.

[37] Mouratidis, H., Giorgini, P., 2007. Secure tropos: a security-oriented extension of the tropos methodology. International Journal of Software Engineering and Knowledge Engineering 17 (2), 285–309.

[38] of Computing Imperial College London, D., 2013. Labelled Transition Analyser Tool. http://www.doc.ic.ac.uk/ltsa/, [Online; accessed 25-august-2015].

[39] Paja, E., Chopra, A. K., Giorgini, P., 2013. Trust-based specification of sociotechnical systems. Data Knowl. Eng. 87, 339–353.

[40] Rodrigues, G. N., Alves, V., Nunes, V., Lanna, A., Cordy, M., Schobbens, P., Sharifloo, A. M., Legay, A., 2015. Modeling and verification for probabilistic properties in software product lines. In: 16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015. IEEE, pp. 173–180.

[41] Rodrigues, G. N., Alves, V., Silveira, R., Laranjeira, L. A., 2012. Dependability analysis in the ambient assisted living domain: An exploratory case study. Journal of Systems and Software 85 (1), 112 – 131, dynamic Analysis and Testing of Embedded Software.

[42] Silva Souza, V. E., Lapouchnian, A., Robinson, W. N., Mylopoulos, J., 2011. Awareness requirements for adaptive systems. In: Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11. p. 60.

[43] Su, G., Rosenblum, D. S., Tamburrelli, G., 2016. Reliability of run-time quality-of-service evaluation using parametric model checking. In: Proceedings of the 38th International Conference on Software Engineering. ICSE '16. ACM, pp. 73–84.

[44] Tang, S., Peng, X., Yu, Y., Zhao, W., 2009. Goal-directed modeling of self-adaptive software architecture. In: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (Eds.), Enterprise, Business-Process and Information Systems Modeling. Vol. 29 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pp. 313–325.

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

[46] van Lamsweerde, A., Willemet, L., December 1998. Inferring declarative requirements specifications from operational scenarios. IEEE Transactions on Software Engineering 24 (12), 1089–1114.

[47] Yu, E., January 1993. Modeling organizations for information systems requirements engineering. In: Requirements Engineering, 1993., Proceedings of IEEE International Symposium on. pp. 34–41.

[48] Yu, E. S.-K., 1996. Modelling strategic relationships for process reengineeringUMI Order No. GAXNN-02887 (Canadian dissertation).

[49] Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J., 2008. From goals to high-variability software design. In: An, A., Matwin, S., Raś, Z., Ślęzak, D. (Eds.), Foundations of Intelligent Systems. Vol. 4994 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–16.