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

Translating Extended Goal Models into Goal Management Controllers in PRISM

Manoel Vieira Coelho Neto

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Genaína Nunes Rodrigues

Coorientador
Dr. Thomas Vogel

Brasília
2024

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

Tradução de Modelos de Objetivos Estendidos em Controladores de Gerenciamento de Objetivos em PRISM

Manoel Vieira Coelho Neto

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Genaína Nunes Rodrigues

Coorientador
Dr. Thomas Vogel

Brasília
2024

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Prof.ª Dr.ª Claudia Nalon

Banca examinadora composta por:

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientadora) — CIC/UnB
Dr. Thomas Vogel (Coorientador) — Humboldt-Universität zu Berlin
Prof. Dr. Vander Ramos Alves — CIC/UnB
Prof. Dr. Simos Gerasimou — University of York

CIP — Catalogação Internacional na Publicação

Coelho Neto, Manoel Vieira.

Translating Extended Goal Models into Goal Management Controllers in PRISM / Manoel Vieira Coelho Neto. Brasília : UnB, 2024.
418 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2024.

1. Self-adaptive systems, 2. model-driven engineering, 3. probabilistic model checking, 4. goal-oriented requirements engineering traceability, 5. controller synthesis

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

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

Translating Extended Goal Models into Goal Management Controllers in PRISM

Manoel Vieira Coelho Neto

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Dr. Thomas Vogel (Coorientador)
Humboldt-Universität zu Berlin

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof. Dr. Simos Gerasimou
University of York

Prof.ª Dr.ª Claudia Nalon
Coordenadora do Mestrado em Informática

Brasília, 19 de dezembro de 2024

Abstract

Self-adaptive systems (SAS) frequently conflate goal evaluation and strategy selection within a single architectural layer, which limits their runtime adaptation capacity. The MORPH architecture addresses this issue by structuring adaptation into three controller layers: Goal Management, Strategy Management, and Strategy Enactment. Within this context, the EDGE (ExtenDed Goal modEling) framework advances the Goal Management layer by introducing a new modelling notation, guided by five desiderata that support dynamic goal reasoning, plan synthesis, and decision memory. EDGE further enables the generation of Markov Decision Process (MDP) models in the PRISM language, representing goal controllers capable of monitoring achievements and computing adaptation plans under uncertainty. However, aligning high-level goal models with formally verifiable goal management controllers remains a core challenge. As systems evolve, the lack of automated, semantics-preserving transformations often leads to divergence between requirements and executable logic, undermining assurance and maintainability. To address this gap, we propose a tool that automatically synthesizes traceable and verifiable EDGE goal controllers in PRISM from annotated EDGE models. In line with the EDGE desiderata, the tool supports automated, modular, and scalable translation from extended i* models into PRISM code, reducing manual effort and reinforcing the connection between goal-level intent and runtime adaptation strategies, thus advancing the practical adoption of model-driven engineering in self-adaptive systems.

Keywords: Self-adaptive systems, goal modeling, i* models, EDGE framework, model-driven engineering, PRISM, Markov decision processes, controller synthesis

Resumo

Os sistemas autoadaptativos (SAS) frequentemente combinam a avaliação de objetivos e a seleção de estratégias em uma única camada arquitetural, o que limita sua capacidade de adaptação em tempo de execução. A arquitetura MORPH resolve essa questão estruturando a adaptação em três camadas de controle: Gerenciamento de Objetivos, Gerenciamento de Estratégias e Implementação de Estratégias. Nesse contexto, a estrutura EDGE (ExtenDed Goal modEling) aprimora a camada de Gerenciamento de Objetivos ao introduzir uma nova notação de modelagem, orientada por cinco desideratos que oferecem suporte ao raciocínio dinâmico de objetivos, síntese de planos e memória de decisões. O EDGE também permite a geração de modelos de Processo de Decisão de Markov (MDP) na linguagem PRISM, representando controladores de objetivos capazes de monitorar conquistas e calcular planos de adaptação em condições de incerteza. No entanto, alinhar modelos de objetivos de alto nível com controladores de gerenciamento de objetivos formalmente verificáveis continua sendo um desafio central. À medida que os sistemas evoluem, a falta de transformações automatizadas que preservem a semântica muitas vezes leva à divergência entre os requisitos e a lógica executável, prejudicando as garantias e a manutenção do sistema. Para resolver essa lacuna, propomos uma ferramenta que sintetiza automaticamente controladores de objetivos EDGE rastreáveis e verificáveis em PRISM a partir de modelos EDGE anotados. Alinhada com os desideratos do EDGE, a ferramenta suporta a tradução automatizada, modular e escalável de modelos i* estendidos para código PRISM, reduzindo o esforço manual e reforçando a conexão entre a intenção a nível de objetivos e a adaptação em tempo de execução.

Palavras-chave: Sistemas auto-adaptativos, engenharia dirigida por modelos, verificação de modelos probabilísticos, engenharia de requisitos orientada a objetivos, rastreabilidade, síntese de controladores

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Self-adaptive systems (SAS)—software systems that autonomously adjust their configurations and behaviors in response to environmental changes—are increasingly vital in domains such as healthcare robotics, IoT, and autonomous vehicles [5]. Adaptation enables these systems to maintain dependable operation under uncertainty and disruption [6]. However, designing SAS that can make effective decisions at runtime remains a key challenge—particularly in modeling system goals and selecting adaptation strategies.

To address these challenges, research in requirements engineering for self-adaptive systems has increasingly adopted goal-based modeling [7], particularly Goal-Oriented Requirements Engineering (GORE). This approach enables systems to represent objectives explicitly, relate them to environmental conditions, and assess their satisfaction during execution under uncertainty.

These developments in goal-oriented modeling establish the theoretical foundation for this work, motivating a closer examination of the current state of goal-based approaches, their open issues, and opportunities for improvement.

1.1 Context and Motivation

Dalpiaz et al. [1] demonstrate that extending a Design Goal Model (DGM) into a Runtime Goal Model (RGM) provides a structured way to evaluate the satisfaction of system goals during execution. This extension also enables the system to explore alternative configurations (or adaptations) that help it recover from failures and restore high-level functionality. As part of this approach, they propose that each goal instance at runtime should follow a specific life cycle, as illustrated in Figure 1.1.

The life-cycle in Figure 1.1 makes clear that runtime goal evaluation depends on explicit representations of uncertainty and behavioral alternatives. Approaches such as GODA [8] show that dependability questions, whether a goal will eventually be achieved,

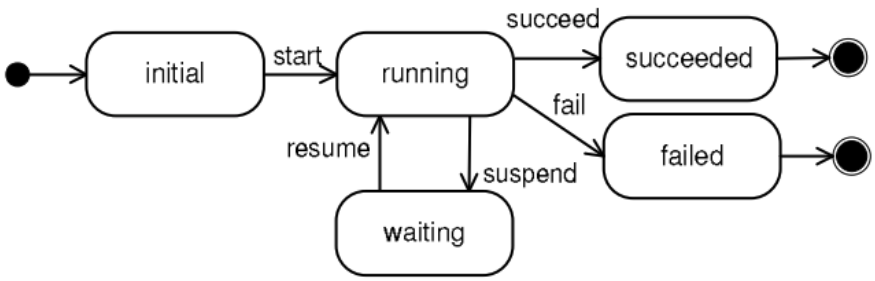


Figure 1.1: Goal life cycle proposed by Dalpiaz et al. [1], supporting runtime evaluation of goal satisfaction.

under what conditions, and with what likelihood, cannot be answered from the graphical model alone. They require an executable formal model that respects goal semantics and supports quantitative analysis. This thesis adopts this principle as a core motivation: enriched goal models should be accompanied by a rigorous probabilistic interpretation. The formal grounding for this, including the PRISM language and its probabilistic model-checking capabilities, is presented in Section 2.2.

Concurrently, the community also explored architectural frameworks capable of supporting goal-driven adaptation. Sykes et al. [2] were among the first to propose a three-layer architecture that separates adaptation concerns, distinguishing goal management from low-level system reconfiguration. Building on this foundation, Braberman et al. [3] refined the model into the MORPH architecture, which explicitly elevates the *Goal Management Layer* as a modular component responsible for interpreting runtime requirements and coordinating adaptive behavior.

Self-adaptive systems require a principled separation between high-level reasoning about goals and the lower-level mechanisms that enact adaptation. The MORPH architecture embodies this separation by distinguishing reasoning from execution and emphasizing traceable feedback loops. This work adopts MORPH's separation of concerns as a guiding principle for translation: runtime goal semantics must remain explicit and analyzable while supporting executable control. A detailed description of the MORPH architecture and its layers is provided in Section 2.3.2.

While MORPH [3] provides the architectural foundation for separating goal management from strategy synthesis and enactment, it does not specify the formalism or mechanism by which the Goal Management Layer reasons about and implements goal adaptation. Addressing this scope, the EDGE framework [9] complements MORPH's separation of concerns by defining a goal modeling language and a set of desiderata specifically for the Goal Management Layer. EDGE translates enriched goal models into Markov Decision Processes (MDPs) and uses probabilistic model checking to derive controller policies

under uncertainty. This approach keeps goal reasoning modular, traceable, and formally verifiable—qualities consistent with the design principles established in MORPH.

Beyond its formal analysis capabilities, EDGE incorporates concepts from cognitive science, particularly the dual-system model of reasoning, which distinguishes between goal-directed planning and habitual execution [10, 11]. This duality is pertinent in self-adaptive systems, where deliberate goal selection must coexist with responsive behaviors to meet latency and reliability constraints. Goal-Oriented Requirements Engineering (GORE) provides the theoretical basis for this perspective, capturing both strategic variability and systematic decomposition of behaviors. For example, Figure 1.2 contrasts AND- and OR-refinements for the same task, illustrating how goal decompositions can represent different operational semantics within the same model.

(a) AND refinement

(b) OR refinement

Figure 1.2: Goal refinements with conjunctive vs. alternative semantics, motivating distinct controller behaviors.

Building on this modeling foundation, EDGE introduces a framework in which each goal functions as an enriched runtime entity, aware of past executions and capable of making probabilistically grounded decisions. However, the automated transformation of such enriched goal models into verifiable controller specifications remains an open challenge.

1.2 Problem Statement

A persistent gap remains between the structural expressiveness of enriched goal models and the ability to derive executable, verifiable controllers from them. While the EDGE framework equips goals with runtime semantics—sequencing, alternatives, retries, and context-aware activation—its transformation into PRISM currently relies on manual and error-prone modeling work. Even small changes to refinements or execution conditions

require developers to adjust multiple fragments of a PRISM specification, demanding expertise that spans both goal-model semantics and low-level formal encoding.

This dual burden impedes scalability and correctness. As models grow or evolve, the risk of semantic drift between the intended behavior and its formal realization increases. Without automated, semantics-preserving transformation, developers lack reliable means to enact or verify the behavior described in their enriched goal models. Addressing this gap requires a principled understanding of what must be preserved, extended, or validated throughout the translation chain—from graphical specifications to executable PRISM controllers.

Research Question 1

RQ1: *What features must a goal modeling tool provide to support consistent traceability from graphical models to formal controller specifications?*

RQ1 (context). This question examines the need for traceability between design-level models and their formal execution artifacts. As stakeholder goals evolve into runtime controllers, the transformation must maintain semantic fidelity. We investigate which representational features—such as runtime execution metadata, activation guards, and dependency declarations—support this consistency.

Research Question 2

RQ2: *What structural extensions to the EDGE notation are necessary to ensure complete and lossless transformation into probabilistic models for runtime reasoning?*

RQ2 (context). This question focuses on the expressive limits of EDGE when used as a basis for automated synthesis. We identify which additional constructs and annotations are required to map enriched goal models to PRISM modules in a complete and lossless manner, capturing retries, guards, degradation paths, and other runtime semantics while preserving model clarity.

Research Question 3

RQ3: *How to evaluate the semantics, liveness, or safety of goal-directed behaviors in synthesized controllers?*

RQ3 (context). This question concerns the validation of synthesized controllers. Once transformed into PRISM models, enriched goal specifications must be evaluated to check

for satisfaction, avoidance of unsafe states, liveness of alternative paths, and semantics of the synthesized behavior. This evaluation is essential for determining controller reliability.

Together, these research questions delineate the scope and direction of the dissertation. They articulate the requirements for bridging enriched goal models and formal controllers, shaping the extensions proposed to the EDGE notation and the transformations needed for automated synthesis. The methodological realization of these ideas—including the full translation workflow and its semantic foundations—is presented in Chapter 3.

1.3 Proposed Solution

To address this gap, this dissertation proposes a translation methodology that systematically maps enriched goal models—annotated according to EDGE semantics—into verifiable PRISM specifications. The methodology defines a set of transformation rules that preserve goal semantics and structural relationships while enabling the formal analysis of adaptive behavior. To demonstrate its feasibility and validate its precision, the methodology has been implemented in a modular translation tool that automates the synthesis process.

The implementation encodes runtime semantics—such as retry strategies, maintain conditions, and goal activation constraints—into PRISM modules consistent with the layered execution principles of the MORPH architecture [3]. The generated PRISM models represent executable controllers that can be formally verified and analyzed while preserving the structural and semantic intent of the original goal model.

This translation systematically maps each goal, task, and refinement relation into corresponding PRISM modules and transitions that reproduce the execution semantics of the goal model, including AND/OR decompositions, retry behavior, and context-sensitive activation. The resulting models support both simulation and PCTL-based validation, enabling quantitative verification of adaptation behavior against defined system properties.

Semantic Preservation Criterion

We say that the transformation preserves the logic of the original goal model if:

1. Each goal or task is translated into a module that mirrors its activation, dependency, and achievement conditions;
2. The temporal and causal relationships between nodes (e.g., AND/OR refinement) are encoded through synchronized transitions and compound guards;
3. Contextual preconditions and retries are respected via explicit guards and counters;
4. The goal life cycle is captured by mutually exclusive pursuit/achievement states;
5. The resulting PRISM model can validate expected execution traces through property checking.

1.4 Methodology

This dissertation adopts an empirical research approach to evaluate a translation methodology that derives PRISM controller models from EDGE-style goal models [9, 12]. The method proceeds in three stages: (i) we identify the Contextual Runtime Goal Model (CRGM) elements required to satisfy EDGE's desiderata, further presented in Section 2.3.3; (ii) we define model-driven mappings from those elements to PRISM constructs using template-based code generation patterns [13]; and (iii) we provide a modular translator as a reference implementation of the method, used to generate analyzable controller models for the case studies.

1.4.1 Evaluation

To validate our approach, we verify the behaviors of two case studies: the Tele-Assistance System (TAS) [14] and the Lab Sample Logistics (LSL) system [15]. These cases differ in scale and operational context, providing complementary settings for applying the translation pipeline and analyzing the generated PRISM models.

For each case study, the methodology is applied end-to-end—from enriched goal model to synthesized PRISM specification—and the resulting models are evaluated according to structural, semantic, and performance metrics defined in the evaluation plan. These data provide quantitative and traceable evidence of the methodology's effectiveness.

This empirical design validates the translation methodology, demonstrating its capacity to produce verifiable controller models. The detailed evaluation framework and data collection process are described in Chapter 6.

1.5 Contributions and Outline

This dissertation contributes to the advancement of model-driven approaches for self-adaptive systems by introducing a systematic methodology that bridges enriched goal modeling and formal controller synthesis. Its relevance lies in addressing one of the major practical limitations of the EDGE framework [9]: the absence of a formalized, semantics-preserving translation process into verifiable PRISM specifications. The work advances the state of the art in goal-based adaptation by defining the modeling elements, transformation rules, and evaluation strategy required to automate this process and assess its correctness in realistic scenarios.

The specific contributions are summarized as follows:

1. **A systematic mapping from EDGE-compliant goal model constructs to a verifiable PRISM representation.** This work defines a precise correspondence between the syntactic elements of enriched goal models—including goals, tasks, maintain conditions, context variables, resources, and goal runtime annotations—and their formal counterparts in PRISM DTMCs. The mapping establishes how elements are translated, how structural relations and elements are preserved, and how complex goal behaviors are encoded into the PRISM model.

2. **A unified semantic structure (IR) that captures the execution meaning of enriched goal models.** The work formalizes the goal semantics of an EDGE goal model and provides a single representational structure capable of expressing all of them consistently. This semantic foundation ensures that the operational interpretation of goal models is preserved during translation, allowing the resulting controller to reflect the intended runtime behavior of the original specification.

3. **A semantics-preserving translation pipeline with explicit generation rules.** The dissertation defines a model transformation process whose generation rules ensure that the semantic structure is preserved in the resulting PRISM goal controller, by specifying:

    - the construction rules that preserve goal life-cycles, refinement relations, and behavioral constraints (e.g., interleaving, sequencing, choice, degradation) in the translated goal controller;

- the conditions under which context, maintain, and resource-related semantics are propagated into guards, updates, and variables structures so that the controller enforces the same adaptation logic as the original goal model.

1.6 Document Structure

The remainder of this document is organized to guide the reader through the conceptual, methodological, and empirical foundations of the work.

- Chapter 2 presents a review of the relevant literature, situating this research within prior developments—from Contextual Goal Models to the EDGE framework—and outlining how these foundations motivate the need for, and contributions of, the proposed translation methodology.

- Chapter 3 describes the translation methodology proposed, detailing the main artifacts introduced throughout the work and the end-to-end transformation workflow that takes goal models as input and produces PRISM specifications as output.

- Chapter 4 introduces the tool implementation developed as part of this thesis, showing how the methodology has been operationalized into an automated translator capable of generating EDGE-compliant PRISM goal controllers.

- Chapter 5 illustrates the methodology in practice by applying it to two established self-adaptive system exemplars frequently discussed in the literature, providing concrete grounding for the approach.

- Chapter 6 reports the empirical results obtained from evaluating the methodology and its implementation. This includes a structured GQM-based validation as well as an analysis of performance metrics related to translation and model checking.

- Chapter 7 concludes the thesis by reflecting on the overall contributions, summarizing the answers to the research questions, and outlining potential directions for extending and refining the EDGE goal-controller synthesis pipeline.

Chapter 2

Literature review

Traditional requirements engineering often focused on defining *what* a system should do, but remained agnostic to *why* stakeholders held those requirements, leading to brittle solutions that failed to adapt to evolving intentions. Goal-oriented Requirements Engineering (GORE) remedies this by making stakeholder objectives—*goals*—the central artifacts of requirements analysis [16]. Goals are then systematically decomposed into finer objectives (AND/OR refinements), making both *why* and *how* explicit across abstraction levels. These simple, but powerful capabilities make GORE a natural foundation for engineering self-adaptive systems that must continuously align goals with runtime decisions.

Goal modeling in self-adaptive systems has deep roots in classical GORE, yet the last decade has witnessed a resurgence of work on executable, context-aware, and formally verifiable goal models, including runtime goal life-cycle semantics [1], contextual activation constructs [17], and probabilistic controller synthesis [8]. To systematically analyze this corpus, we organize existing approaches along four critical dimensions, each illuminating a different aspect of how goal-based reasoning has evolved toward formal, runtime-adaptive modeling:

i goal modeling expressiveness;

ii formal representation for verification;

iii architectural integration; and

iv controller synthesis from contextual goal models.

Together, these dimensions outline the field's progression from descriptive goal specification to executable and verifiable models—setting the stage for the EDGE framework and the translation methodology proposed in this work.

2.1 Contextual and Runtime Goal Models

Ali et al. [17] propose a framework that enriches traditional goal models with explicit context annotations. Their Contextual Goal Model (CGM) encodes, for each goal, the precise environmental conditions under which that goal should be activated or suppressed. This enables self-adaptive software to dynamically choose among alternative goal refinements—AND/OR decompositions—based not only on structural dependencies but also on the current context, thus ensuring that the system continues pursuing its primary objectives even as its environment evolves.

In parallel, Dalpiaz et al. [1] extend the goal modeling paradigm toward runtime reasoning. They distinguish two complementary views: the *Design Goal Model* (DGM), which captures stakeholders' requirements and supports exploration of architectural alternatives at design time, and the *Runtime Goal Model* (RGM), which enriches the DGM with behavioral constraints and stateful goal instances—each following a defined life cycle—so that the system can monitor goal activation, success, and failure during execution and adapt its behavior accordingly.

In Runtime Goal Models (RGMs) [1], the traditional goal hierarchy is augmented with explicit *runtime annotations*—such as sequencing, cardinality, and lifecycle constraints on goal and task instances. These annotations transform a static design-time blueprint into an executable model, endowing the system with:

- **Precise runtime behavior:** exact ordering and repetition rules for goal activation and task invocation;
- **Instance-level reasoning:** each goal instance maintains its own state (pending, running, succeeded, failed);
- **Adaptive support:** dynamic enabling or disabling of goals in response to context changes;
- **Fault diagnosis:** rapid identification of which goal or task instance violated its constraint;
- **Traceability:** a concrete audit trail linking runtime events back to specific model elements.

Runtime annotations precisely define the behavioral, temporal, state or the constraints of the goal execution, specifying allowable sequences, concurrency, alternatives, and conditional execution among goals and tasks. The table 2.1 shows the operators introduced by the RGM work.

Table 2.1: Runtime Annotation Operators

Operator/Function	Meaning	Category
`skip`	Do nothing	Behavioral
`E1; E2`	Sequential occurrence: E1 followed by E2	Temporal
`E1 | E2`	Alternation (exclusive choice): either E1 or E2	Behavioral
`opt(E)`	Optional: E is optional	Behavioral
`E+`	One or more sequential occurrences of E	Temporal
G^{succ}	Goal instance starts and terminates successfully	State
G^{fail}	Goal instance starts and terminates with failure	State
`try(G)?E1:E2`	Conditional execution: E1 if G succeeds; E2 if G fails	Conditional
`E1 # E2`	Interleaved occurrence of E1 and E2	Temporal
`E#`	One or more concurrent (interleaved) instances of E	Temporal
`requires(G1,G2)`	G1 can occur only if G2 has occurred previously	Temporal constraint
`prevents(G1,G2)`	Occurrence of G1 prevents subsequent occurrence of G2	Temporal constraint

These enriched goal models allow designers to specify systems not merely in terms of what they should achieve but also how they should dynamically behave and adapt. By introducing context-sensitive activation and runtime goal lifecycle reasoning, Ali et al. [17] and Dalpiaz et al. [1] independently advanced the expressiveness of goal modeling toward self-adaptive systems capable of reasoning about operational context and goal satisfaction in real time.

The GODA framework [8] unifies these complementary perspectives through the Contextual Runtime Goal Model (CRGM), which integrates contextual activation from CGMs with lifecycle semantics from RGMs. GODA leverages these enriched models to automatically derive formal PRISM specifications, enabling quantitative verification and contextual runtime analysis. The influence of both CGM and RGM approaches extends to later frameworks such as MUTROSE [18] and MODALAS [19], which continue to refine runtime-oriented methodologies for goal-based adaptation.

As goal-based modeling matured, researchers increasingly sought formal verification methods to ensure that adaptive decisions remained dependable under uncertainty. The use of PRISM in GODA exemplifies this evolution, marking a shift from descriptive goal modeling to verifiable, probabilistic representations that support both analysis and automated controller synthesis.

2.2 Formal Representation and Probabilistic Evaluation

Ensuring that autonomous systems reliably satisfy all specified properties throughout their execution is a fundamental challenge. Informal or semi-formal approaches often lack the rigor needed to guarantee correctness, leaving critical system behaviors potentially unchecked or insufficiently validated. According to Araujo et al. [20], formal verification has emerged as the predominant approach to validate autonomous and robotic systems, as it systematically ensures that all specified properties are consistently met during execution.

We next summarize probabilistic model checking and the PRISM tool, which together provide the formal substrate for the quantitative analyses conducted in this dissertation (Sections 2.2–2.2.2).

2.2.1 Probabilistic Model Checking

Model checking originated as a formal verification method to ensure that systems conformed to qualitative logical specifications. Over time, this approach evolved to incorporate quantitative reasoning, giving rise to *Probabilistic Model Checking* (PMC), which enables the verification of systems exhibiting uncertainty and stochastic behavior [21]. PMC determines whether a model satisfies quantitative properties expressed in probabilistic terms, making it particularly suited for safety-critical domains such as robotics, communication networks, and autonomous systems.

Among the stochastic models used in PMC, Markovian frameworks are especially relevant for representing uncertainty through probabilistic transitions. Two primary models are the *Discrete-Time Markov Chain* (DTMC), which defines transition probabilities $\pi(s, s')$ between states such that $\sum_{s'} \pi(s, s') = 1$ [12], and the *Markov Decision Process* (MDP), which extends DTMCs by introducing nondeterministic choices that support strategic decision-making under uncertainty.

Integrating MDPs into PMC frameworks has become essential for robotics and autonomous systems. This integration provides end-to-end validation—from high-level strategy synthesis to runtime behavioral assurance—by systematically quantifying risk, optimizing decisions, and verifying safety properties under uncertainty [9]. As a result, MDP-based probabilistic verification forms the analytical backbone for modern adaptive systems.

The following subsection introduces PRISM, a leading tool for probabilistic model checking capable of analyzing DTMCs and MDPs. It highlights PRISM's temporal logic

support and its application across domains ranging from distributed algorithms and wireless networks to goal-based system verification [8, 12].

2.2.2 PRISM

Among existing tools for probabilistic verification, PRISM is the most widely used, offering support for *Discrete-Time Markov Chains* (DTMCs), *Continuous-Time Markov Chains* (CTMCs), and *Markov Decision Processes* (MDPs) [12]. PRISM specifies and verifies quantitative properties using probabilistic temporal logics, particularly *Probabilistic Computation Tree Logic* (PCTL) for DTMCs and MDPs, and *Continuous Stochastic Logic* (CSL) for CTMCs.

PRISM provides a modular modeling language in which systems are described through *modules* containing local state variables and probabilistic commands. Modules interact either through shared variables or via synchronized action labels, enabling the modeling of interdependent behaviors in distributed or concurrent settings [12]. These constructs are central to the formal artifacts generated by this dissertation.

Listing 2.1 presents a simple DTMC model illustrating the core elements of PRISM: modules, state variables, synchronization labels, probabilistic transitions, and updates. The example models a light–door interaction in which opening the door depends on the light state, and closing the door synchronizes with the light module.

```
dtmc

// Light and Door System
module light
  // Light state: 0 = off, 1 = on
  lstate : [0..1] init 0;

  // Toggle the light independently
  [toggle_light] lstate=0 -> 1.0 : (lstate'=1);
  [toggle_light] lstate=1 -> 1.0 : (lstate'=0);

  // Synchronize: Closing the door turns off the light
  [close_door] lstate=1 -> 1.0 : (lstate'=0);
  [close_door] lstate=0 -> 1.0 : (lstate'=0);
endmodule

module door
  // Door state: 0 = closed, 1 = open
  dstate : [0..1] init 0;
```

```
20
21    // Open door only if light is on
22    [open_door] dstate=0 & lstate=1 -> 1.0 : (dstate'=1);
23    // Closing door triggers synchronization with light module
24    [close_door] dstate=1 -> 1.0 : (dstate'=0);
25  endmodule
```

Listing 2.1: DTMC model in PRISM representing a light-door interaction with inter-module synchronization.

Commands in PRISM follow the general form:

$$\texttt{[action]}\ \langle \text{guard} \rangle \ \rightarrow\ p_1 : \langle \text{update}_1 \rangle + \cdots + p_n : \langle \text{update}_n \rangle\ ;$$

where **actions** label transitions for synchronization, **guards** determine when a transition is enabled, **probabilities** (or rates) quantify choice, and **updates** specify the resulting state.

PRISM has been used in goal-oriented verification and synthesis tasks relevant to this work. Sekizawa et al. [22] apply PRISM to analyze autonomous robotic vehicles under faults, and the GODA framework [8] synthesizes parameterized PRISM models for goal achievability and plan generation. These precedents demonstrate PRISM's suitability as a backend for transforming enriched goal models into verifiable controller specifications.

The following subsection reviews architectural approaches for goal-based adaptation, motivating how expressive goal models integrate with probabilistic controllers during execution.

2.3 Architectural Support for Goal-Based Adaptation on Autonomous Systems

Autonomous systems are highly reliable systems where human intervention is either restricted or undesirable [2]. To operate effectively under such conditions, these systems must be capable of reacting to environmental changes, pursuing alternative goals, and executing fallback actions when errors occur during runtime. Designing systems with these capabilities poses significant challenges. However, prior research has shown that adopting an architectural approach to enable self-adaptability often results in more granular and generalizable frameworks, applicable across multiple domains. Such architectural designs provide essential abstraction mechanisms for defining runtime adaptation in large and complex systems [2, 3].

The following subsections examine three key architectures that have shaped the evolution of adaptation reasoning in self-adaptive systems. We begin with the Three-Layer Architecture, which established the foundational separation of concerns for adaptive control. This model was later refined into the MORPH Reference Architecture, introducing a more formalized and modular approach to goal and strategy management. Finally, we discuss how MORPH's principles directly informed the design of the EDGE framework, which advances these ideas by providing a formal and verifiable implementation of the Goal Management Layer.

2.3.1 Three-layer Architecture

To achieve greater adaptability and modularity in autonomous systems, Sykes et al. [2] introduced a three-layer architectural model that separates adaptation concerns across abstraction levels. At its core lies the *Goal Management Layer*, which interprets high-level objectives and drives runtime reasoning, while the underlying *Change Management* and *Component Layers* coordinate strategy execution and system reconfiguration. This separation established a foundational principle for self-adaptive architectures—distinguishing between goal reasoning and operational control—that directly influenced subsequent frameworks such as MORPH and EDGE.

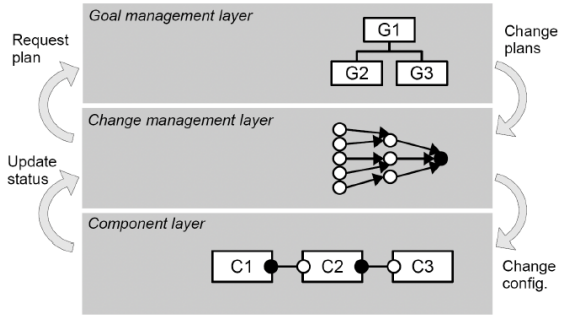


Figure 2.1: Three-layer model [2].

Building on this architectural separation, later research explored the underlying mechanisms that enable coordination across layers in practice. Two main theoretical foundations emerged: **dynamic configuration**, which allows structural and behavioral modifications of the system during execution, and **discrete control theory**, which provides formal mechanisms to plan and regulate these adaptations. Together, these approaches ensure that self-adaptive systems can reconfigure themselves while preserving stability and dependability under changing runtime conditions [3].

These advances reinforced the importance of distinguishing between *decision-making* about *what* to adapt—handled by the Goal Management Layer—and the mechanisms that

determine *how* adaptations are enacted at lower levels. Many early approaches blurred this separation, leading to rigid architectures that coupled reasoning and execution logic. By emphasizing clear feedback loops between layers, Sykes et al. [2] laid the groundwork for scalable and modular adaptation frameworks. These principles were later consolidated in the **MORPH architecture** [3], which formalized this division of responsibilities into a reusable and analyzable reference model for self-adaptive systems, as discussed in the next subsection.

2.3.2 MORPH Reference Architecture

The MORPH reference architecture (see Figure 2.2), proposed by Braberman et al. [3], provides a structured foundation for engineering self-adaptive systems by formalizing the conceptual separation of concerns introduced by Sykes et al., organizing adaptive reasoning into three interconnected layers—*Goal Management*, *Strategy Management*, and *Strategy Enactment*. Each layer assumes a distinct role within the adaptation loop, from high-level decision-making to runtime execution, while communicating through well-defined feedback channels. This hierarchy promotes modularity, traceability, and reuse, ensuring that adaptation decisions remain coherent across abstraction levels.

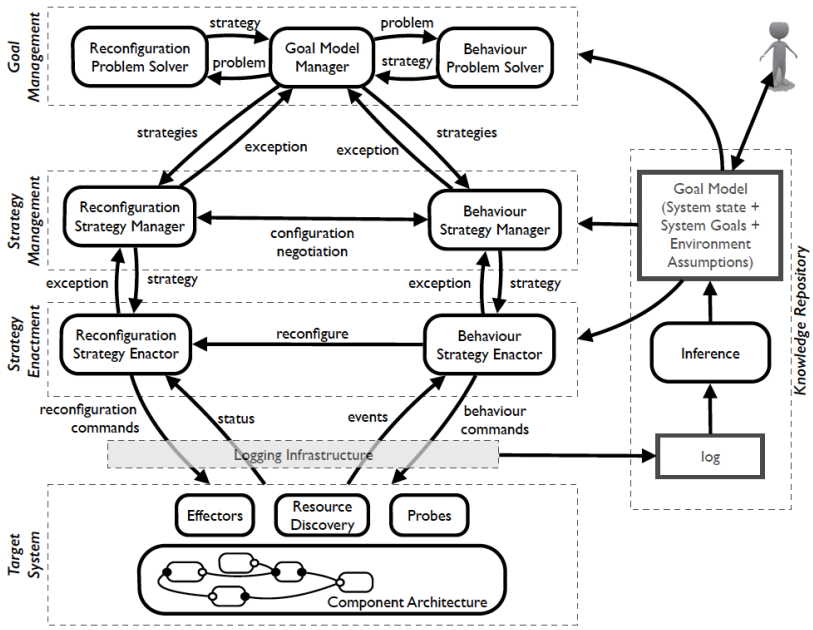


Figure 2.2: The MORPH reference architecture [3].

At the top of this hierarchy, the **Goal Management Layer** interprets system objectives, decomposes them into concrete adaptation problems, and orchestrates reasoning

about both behavioral and reconfiguration goals. This layer represents the cognitive center of MORPH—it abstracts the system's intent into analyzable models and determines how strategies should evolve when the system faces uncertainty or failures. Beneath it, the **Strategy Management Layer** selects among pre-computed behavior and reconfiguration strategies to balance performance, reliability, and response time, while the **Strategy Enactment Layer** executes these strategies on the managed system, monitoring outcomes and propagating exceptions back to higher layers.

These layers implement a feedback hierarchy where adaptation flows downward—from goal reasoning to operational control—while monitoring and failure information propagate upward.

This feedback design allows high-level goals to remain stable even as the system continually reconfigures itself to meet runtime constraints. MORPH's architectural principles thus institutionalize a key idea in self-adaptive systems: separating the question of *what* to adapt (reasoning) from *how* to adapt (execution).

Table 2.3 summarizes the main responsibilities of each layer and how MORPH structures failure handling across its adaptation cycle. The design ensures that recovery and decision logic are isolated at appropriate abstraction levels, supporting modular reassessment of strategies without disrupting lower-level components.

Layer	**Design Rationale**	**Failure Handling**
Goal Management	Enables computationally intensive reasoning and planning to occur offline, allowing fast decision-making at runtime.	Relies on upward feedback from lower layers to trigger strategic reassessment.
Strategy Management	Coordinates behavior and reconfiguration strategies under time constraints using precomputed alternatives.	Re-evaluates or negotiates new strategies when notified of execution failures.
Strategy Enactment	Ensures low-latency application of selected strategies with minimal computational overhead.	Monitors execution outcomes and reports unexpected conditions upward for correction.

Table 2.3: Design rationale and failure handling across MORPH layers.

However, while MORPH defines these interfaces conceptually, it leaves the internal realization of goal reasoning mechanisms open—precisely the focus of EDGE. In par-

ticular, MORPH does not prescribe how the Goal Management Layer should represent, analyze, or verify adaptation logic. The EDGE framework [9] builds directly on this gap, formalizing the Goal Management process through a goal modeling language and a verification-oriented workflow. The following subsection introduces EDGE, outlining how it operationalizes MORPH's conceptual principles into a formally analyzable goal reasoning mechanism.

2.3.3 EDGE

As outlined in Section 1.2, the EDGE (ExtenDed Goal modElling) framework [9] fills MORPH's gap at the Goal Management Layer by providing a goal modeling language and a verification-oriented workflow for runtime reasoning. In this section, we focus on the aspects of EDGE that matter for this dissertation: its modeling constructs, its desiderata for autonomous decision-making, and their formal realization in PRISM.

Figure 2.3 illustrates an EDGE goal model. It highlights three extensions central to runtime reasoning: explicit goal variants (e.g., at G3/G4), inter-goal dependency links (e.g., G2→G5), and quantitative properties (e.g., utility, cost). These elements are precisely the anchors our method maps into verifiable PRISM structures.

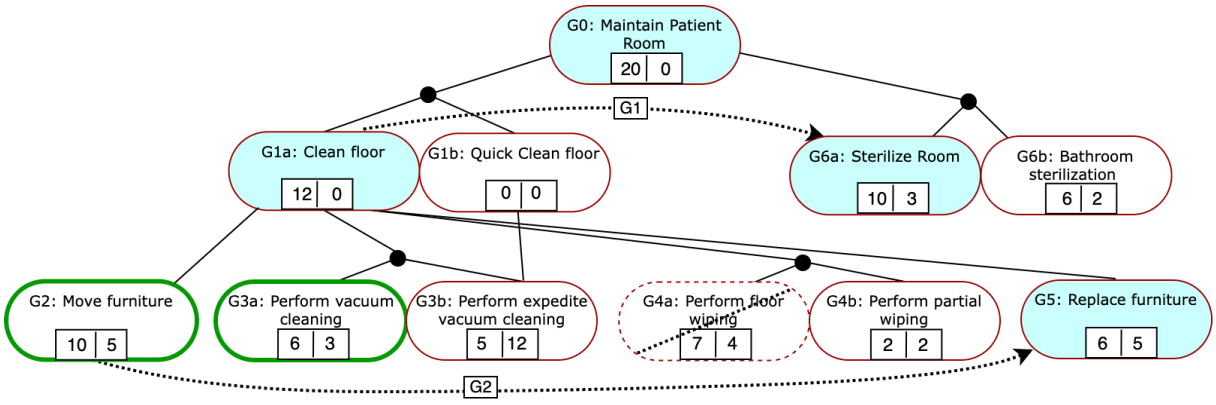


Figure 2.3: Example goal model in EDGE notation showing goal variants, inter-goal dependencies, and quantitative properties (utility and cost).

We analyze EDGE from two complementary angles that matter for this dissertation: (1) its *desiderata* for runtime goal management (what a model must express to support adaptive decision-making), and (2) its *formal realization* in PRISM (how those concepts become analyzable controllers).

EDGE Desiderata The following table summarizes EDGE's modeling requirements for autonomous decision-making—what information a goal model must carry to support dependable adaptation under uncertainty.

Desideratum	Description	Human-Inspired Motivation
D1 – Non-Idempotent Goal Variants	Explicitly represent and dynamically select between multiple non-repeatable variants of a goal.	People switch variants as context, time, or failures evolve.
D2 – Goal Properties	Associate goals with quantitative/qualitative properties (utility, risk, cost, impact).	People trade off safety, cost, reward when choosing actions.
D3 – Goal Status Tracking	Continuously track runtime status (*achievable*, *active*, *achieved*, *failed*).	People monitor progress and adjust after success/-failure.
D4 – Goal Interdependencies	Model enabling, disabling, and sequencing effects across goals/-variants.	People reason about how one objective constrains another.
D5 – Automated Goal Selection	Support transformation into formal models suitable for controller synthesis (e.g., MDPs).	People revise choices under change; systems need a computable analog.

Table 2.4: EDGE Desiderata for Goal Models Supporting Autonomous Decision-Making

For this dissertation, D1–D4 delineate the semantic commitments that the translation must preserve, while D5 motivates an automatic translation from enriched goal models to PRISM.

EDGE Contributions for Goal Modeling The next table connects each desideratum to its concrete impact on model design—clarifying how EDGE's notation informs our translation templates and verification targets.

Desideratum	Impact on Goal Modeling
D1	Makes alternative achievement paths explicit and selectable at runtime.
D2	Enables property-aware trade-offs and prioritization of variants.
D3	Provides decision memory via status variables for adaptive revision.
D4	Ensures consistency by encoding feasibility and conflict constraints.
D5	Establishes a bridge to model checking and policy synthesis.

Table 2.5: Impact of EDGE desiderata on goal model design and translation

Together, these effects shape the *inputs* that our methodology transforms and the *properties* we later verify in PRISM.

EDGE PRISM Model The following examples illustrate how EDGE realizes these desiderata in PRISM, bridging modeling concepts (D1–D4) with executable semantics.

EDGE's encoding comprises three modules—*Goal Controller*, *Change Management*, and *Turn*—mirroring the upper layers of MORPH [2, 3]. Each goal is associated with `achieved`, `achievable`, and `pursued` state variables, enabling decision memory (D3) and variant selection (D1) under interdependency constraints (D4). The controller chooses whether—and which variant—to pursue based on current status:

```
[G2_skip]t & n=0 & (G2_achieved | !G2_achievable) -> 1:(G2_pursued'=0)&(n'=1);
[G2_pursue0]t & n=0 & !G2_achieved & G2_achievable -> 1:(G2_pursued'=0)&(n'=1);
[G2_pursue]t & n=0 & !G2_achieved & G2_achievable -> 1:(G2_pursued'=1)&(n'=1);
```

Listing 2.2: EDGE Goal Controller: decisions for goal G2 (illustrating D1, D3). [23]

Execution outcomes are captured probabilistically in *Change Management*, updating status and flagging failures that trigger plan revision—thus linking D1/D3 to verifiable dynamics:

```
[] !t & !fail & step=2 & G4_pursued=0 -> 1:(step'=3);
[] !t & !fail & step=2 & G4_pursued=1 -> p4_1:(G4a_achieved'=true)&(step'=3) +
    ↪ (1-p4_1):(G4a_achievable'=false)&(fail'=true)&(step'=3);
[] !t & !fail & step=2 & G4_pursued=2 -> p4_2:(G4b_achieved'=true)&(step'=3) +
    ↪ (1-p4_2):(G4b_achievable'=false)&(fail'=true)&(step'=3);
```

Listing 2.3: EDGE Change Management: probabilistic outcomes for goal G4 (illustrating D1–D4). [23]

However, despite this formal integration of modeling and verification, EDGE currently lacks automated support to perform the translation from enriched goal models to PRISM. The next section examines this limitation and motivates the semantics-preserving translation methodology proposed in this work.

2.4 Synthesis of Controllers from Contextual Goal Models

To contextualize EDGE's contribution, this section revisits prior efforts on translating contextual goal models into PRISM, focusing on GODA [8] as the main precursor to our work. Earlier in this review, we noted GODA's importance in formalizing contextual goal

dependability; here, we contrast its translation process with EDGE to highlight how the field evolved toward semantics-preserving controller synthesis.

The GODA framework translates Contextual Runtime Goal Models (CRGMs) into PRISM specifications. It encodes task compositions and goal as formulas using probabilistic operators and guarded commands, enabling quantitative reasoning about goal satisfaction under uncertainty. GODA supports multiple composition patterns—sequential, conditional, parallel, and fallback structures—organized as reusable templates for task orchestration. Table 2.1 summarizes these patterns and their notation.

Structure	**Notation and Semantics**
Sequential AND	`AND(n1; n2)`: Sequential fulfillment of n_1 then n_2
Parallel AND	`AND(n1#n2)`: Parallel fulfillment of n_1 and n_2
Sequential OR	`OR(n1; n2)`: Fulfillment of either or both n_1, n_2 sequentially
Parallel OR	`OR(n1#n2)`: Fulfillment of either or both n_1, n_2 in any order
Repetition	`n+k`: Fulfill n exactly k times, where $k > 0$
Retry	`n@k`: Retry fulfillment of n at most $k - 1$ times
Optional Fulfillment	`opt(n)`: n may be skipped optionally
Ternary Conditional	`try(n)?n1:n2`: If n succeeds, do n_1; else, do n_2
Alternative Choice	`n1|n2`: Exclusive fulfillment of either n_1 or n_2

Table 2.6: GODA control structures and their notation (adapted from [4]).

These constructs form the foundation of GODA's PRISM encoding. Figure 2.4 illustrates a sequential composition in which task `T2` begins only after `T1` completes successfully. The corresponding PRISM excerpt in Listing 2.4 demonstrates how GODA translates such structures into modules synchronized through shared actions and context variables. Together, these examples show how GODA formalizes runtime orchestration but stops short of full goal-state reasoning.

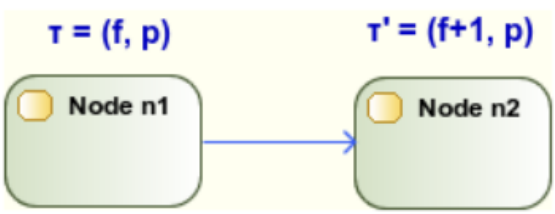


Figure 2.4: Sequential node composition in the GODA framework [4].

```
module N1
  sN1 : [0..4] init 0;
  [initN1] sN1=0 -> (sN1'=1);
  [] sN1=1 -> 0.8:(sN1'=2) + 0.2:(sN1'=4);
  [success] sN1=2 -> (sN1'=2);
  [failN1] sN1=4 -> (sN1'=4);
endmodule

module N2
  sN2 : [0..4] init 0;
  [success] sN1=2 & sN2=0 -> (sN2'=1);
  [] sN2=1 -> 0.9:(sN2'=2) + 0.1:(sN2'=4);
  [success] sN2=2 -> (sN2'=2);
  [failN2] sN2=4 -> (sN2'=4);
endmodule
```

Listing 2.4: PRISM model excerpt illustrating sequential node composition in GODA.

While GODA successfully formalizes CGRMs into DTMCs and MDPs PRISM models, its focus remains primarily on static task orchestration—determining the execution order of predefined tasks—rather than dynamic goal management. The framework does not explicitly track runtime goal states (such as whether a goal was previously pursued, succeeded, or failed), limiting its ability to support adaptive reasoning when contextual or operational conditions change.

When evaluated against the desiderata established by EDGE [23], these limitations become evident. GODA falls short on D1 (Non-idempotent Goal Variants), lacking mechanisms for runtime selection among alternative goals; on D4 (Goal Interdependencies), as dependencies between goals are not explicitly formalized; and on D5 (Automated Goal Selection), since its translation focuses on deterministic task sequences rather than adaptive policy derivation. In contrast, EDGE's modular MDP encoding tracks *achieved*, *achievable*, and *pursued* states—enabling policy-driven decision-making that revises plans based on success, failure, or contextual change.

Despite these advancements, EDGE currently provides no automated means of translating extended goal models into PRISM specifications. The transformation remains manual and error-prone, demanding both expertise in goal modeling and detailed knowledge of PRISM syntax. As model complexity grows—incorporating goal variants, contextual dependencies, and quantitative properties—the lack of automation hinders scalability, traceability, and consistent semantics across model iterations.

In summary, research has progressed from contextual goal reasoning (GODA) to probabilistic controller modeling (EDGE), yet a crucial gap persists: the absence of a systematic, semantics-preserving methodology to automate the translation from enriched goal models to verifiable controller specifications. This dissertation addresses that gap by defining and implementing such a translation methodology, ensuring faithful alignment between high-level design intent and executable verification models.

Chapter 3

Methodology

This chapter details the methodological foundation of the proposed translation process, outlining how enriched goal models are systematically transformed into verifiable PRISM specifications. The methodology is intentionally designed as a structured, multi-phase pipeline that links conceptual goal-oriented reasoning with formal, analyzable controller models. Its overarching aim is to preserve the semantics encoded in extended EDGE-compliant goal models while expressing their behavior in a form suitable for probabilistic verification. To achieve this, the chapter formalizes the modeling constructs and transformation rules that collectively support semantics-preserving synthesis under uncertainty.

The methodology unfolds through three tightly connected stages. First, the *Identification of Semantically Relevant Elements* establishes the conceptual modeling requirements of the EDGE framework, defining the goal, task, resource, and dependency constructs that constitute the modeling language. This phase formalizes the additional annotations—contextual guards, maintenance constraints, retry limits, runtime execution structures—that are necessary to capture the desiderata D1–D4 and ensure that the goal model is sufficiently expressive for automated reasoning. Second, the *Intermediate Representation (IR)* abstracts these enriched semantics into a normalized, language-agnostic `GoalTree`. The IR serves as the semantic anchor of the translation pipeline: it filters away graphical syntax and captures only the structural, behavioral, and quantitative information required for the controller synthesis. Finally, the *PRISM Controller Translation* phase defines the transformation rules that traverse the IR and generate the corresponding PRISM modules, preserving both the reasoning semantics of the goal hierarchy and the operational semantics of tasks and resources.

The resulting pipeline comprises three main PRISM components: a `ChangeManager` module that operationalizes task execution behavior, a `System` module that captures contextual and environmental variables, and a set of goal modules that encode the decision logic of each goal node. Each component is generated through parameterized templates

whose structure and semantics are described in detail throughout this chapter. The rationale behind this separation is twofold. First, it mirrors architectural principles from MORPH, maintaining a clear distinction between reasoning and enactment layers from the system. Second, it ensures modularity and traceability, allowing each part of the model—the extended notation, the IR, and the generated PRISM code—to reflect one another in a systematic manner.

The remainder of this chapter presents each methodological stage in depth. Section 3.1 outlines the empirical research design supporting the translation methodology. Section 3.2 introduces the conceptual structure of the four-phase transformation framework. The subsequent sections describe the syntactic and semantic foundations of the extended goal model (Phase 1), the construction of the Intermediate Representation that captures its semantics (Phase 2), and the template-based generation rules that synthesize the executable PRISM controller (Phase 3). These sections provide a comprehensive account of the methodological principles, modeling decisions, and transformation mechanisms that ground the semantics-preserving translation process developed in this dissertation.

3.1 Research Approach

This section operationalizes the research questions defined in Section 1.2 by adopting an *artifact-based methodological inquiry*. Rather than relying solely on theoretical analysis, the study builds and observes a working translation process that maps enriched goal models into verifiable PRISM specifications. This empirical orientation enables the investigation of how modeling constructs—such as interdependencies, contextual guards, and retry strategies—affect the formal structure and behavioral properties of the resulting controllers.

The inquiry proceeds through representative case studies that capture distinct operational characteristics of Self-Adaptive Systems (SAS). Each case provides a realistic setting for evaluating how systematically defined transformation rules behave under different goal hierarchies, contextual conditions, and sources of uncertainty. By applying the same translation rules across models with varying structure and semantics, the method allows systematic observation of cause–effect relationships between model design and verifiability outcomes.

Through this process, the research examines how the translation reacts to structural variation, how semantic commitments are preserved in the PRISM specification, and how PCTL properties reflect the behaviors encoded in the original goal model. These observations form the empirical foundation for assessing the feasibility and adequacy of

the proposed methodology. The following sections formalize the conceptual framework and describe the design of the translation pipeline that instantiates this approach.

3.2 Methodological Framework

The methodological framework defines the conceptual and procedural backbone of the translation process from enriched goal models to verifiable PRISM specifications. As introduced in Section 1.2, the research questions motivate a structured approach that transforms goal reasoning constructs into analyzable, executable controllers.

Figure 3.1 presents the overall process, which is organized into three sequential phases: (i) **Syntax & Semantic Mapping**, (ii) **Intermediate Representation Construction**, and (iii) **PRISM Controller Translation**. Each phase captures a distinct responsibility in the transformation pipeline, establishing a disciplined pathway from enriched modeling constructs to formal verification artifacts.

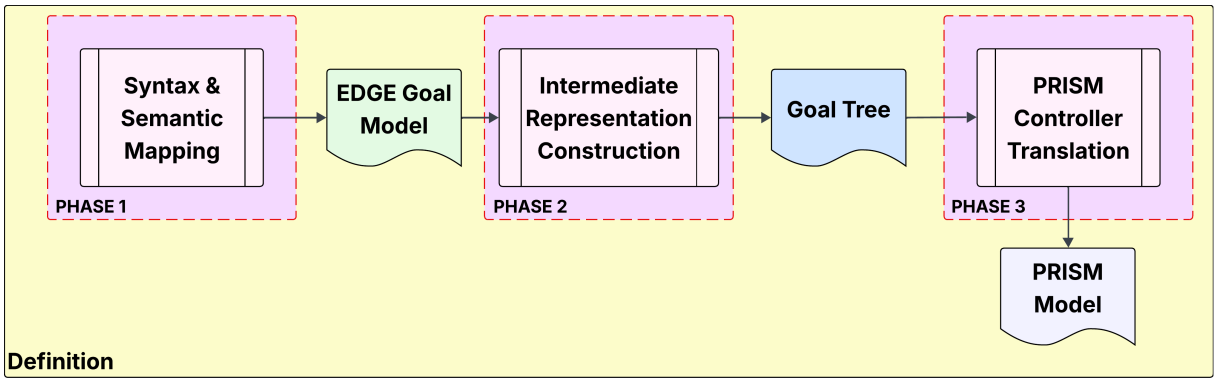


Figure 3.1: Translation framework methodology

The first phase, **Syntax & Semantic Mapping** (section 3.3), defines the syntactical and semantic foundations required for a goal model to conform to EDGE's desiderata. This involves identifying structural extensions and annotations such as goal variants, contextual guards, quantitative properties, inter-goal dependencies, and runtime annotations. These constructs specify both the structure of valid models and the semantics that must be preserved throughout the transformation. The outcome is an *extended goal model* that offers a complete and analyzable representation of the system's adaptive behavior.

The second phase, **Intermediate Representation Construction (IR)** (section 3.4), introduces an abstraction layer between the extended goal model and its PRISM encoding. The IR captures refinement structure, behavioral conditions, and execution policies in a unified, language-agnostic form. By isolating semantic concerns from notation, the IR supports modular and traceable manipulation of goal semantics and prepares the model for systematic code generation.

The third phase, **PRISM Controller Translation**, employs a template engine that applies predefined transformation rules to translate the IR into executable PRISM modules. Each template encodes the semantics of a specific element type—goal, task, or dependency—and generates synchronized transitions that reproduce AND/OR decompositions, contextual activation, retries, and failure handling. This phase renders the IR directly into formal specifications suitable for probabilistic verification.

The resulting PRISM architecture reflects the layered reasoning principles of the MORPH reference model [3], maintaining a clear separation between decision logic and execution control. Template-Based Code Generation (TBCG) [13] ensures consistency and reproducibility across generated models, since all transformation rules are explicitly encoded in the engine.

Each phase contributes directly to the research questions defined in Section 1.2: Syntax & Semantic Mapping operationalizes the traceability requirements (RQ1), the IR formalizes the structural extensions needed for complete transformation (RQ2), and the PRISM Controller Translation phase provides the executable artifacts that enable formal reasoning (RQ3). Together, these phases establish a repeatable and theoretically grounded process for translating enriched goal models into verifiable controllers under the EDGE framework.

In the following subsections, we detail each phase of the methodology. Phase 1 (section 3.3) identifies the semantically relevant elements and features required by EDGE—goals, tasks, resources, AND/OR links, contextual guards, dependencies, and runtime annotations—culminating in a concise metamodel and ERD that constrain valid models and trace to PRISM constructs. Phase 2 (section 3.4) defines a normalized Intermediate Representation based on a `GoalTree` schema that captures structure, quantitative properties, and execution semantics for template traversal. Finally, Phase 3 (section 3.5) specifies the template-based code generation rules that render the IR into PRISM modules, including a `ChangeManager`, a `System` module for resources and context, and one module per goal encoding pursue/achieve/skip synchronization.

3.3 Phase 1: Identification of Semantically Relevant Elements

To conform with the EDGE desiderata, certain extensions to goal models are required. In this subsection, we enumerate the elements and features that a goal modeling tool must support in order to be EDGE-compliant and to enable the transformation into PRISM models. We begin by describing the basic building blocks of the model (goals, tasks, and resources) along with their required features. We then examine the semantics of runtime

annotations associated with goals, and finally, we depict the relationships among these elements with an EMF diagram that formalizes the goal model.

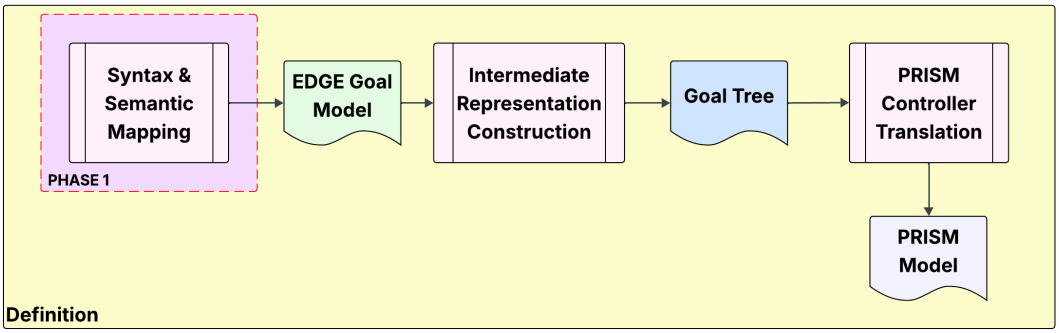


Figure 3.2: First stage of the methodology: define the goal model requirements

EDGE-compliant goal models rely on four fundamental building elements drawn from GORE—goals, tasks, resources, and AND/OR links—which together define the hierarchical structure of intent and behavior. These constructs provide the baseline needed to address desiderata D1–D4; however, meeting those desiderata in full demands extending each element with additional semantic features. The following subsections detail these extensions and juxtapose them with their generic PRISM counterparts to clarify how the enriched notation aligns with the target formalism.

3.3.1 Goals

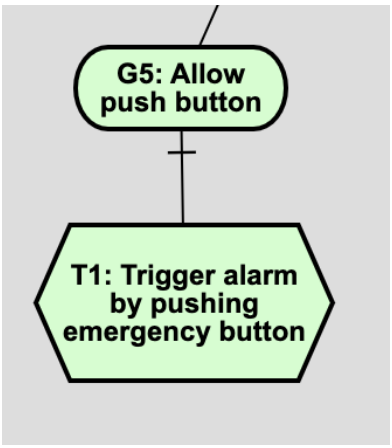
As the fundamental element of the model, goals lie at the core of both traditional GORE methodologies and the EDGE framework. Traditional GORE approaches are largely aligned with EDGE's principles, as EDGE builds directly upon the foundational concepts of goal-oriented modeling. However, in the original EDGE paper, the goal model is composed exclusively of goals enriched with additional properties, where the leaf goals in the hierarchy represent concrete actions to be executed in the target system. In our approach, we refine this structure by transforming these leaf goals into *tasks* (see section 3.3.4), thereby achieving a clearer separation of concerns between intentional objectives and operational activities.

Goals capture the system's intentional rationale—what must be achieved and under which conditions—serving as the core mechanism for linking high-level requirements to executable behaviors (see chapter 2). While this semantic foundation aligns with the EDGE desiderata D1–D4, fulfilling them requires explicit modeling extensions: identifiers for distinguishing goal variants (D1), quantitative attributes such as cost and utility (D2), support to goal lifecycles (D3) and constructs for expressing inter-goal dependencies (D4).

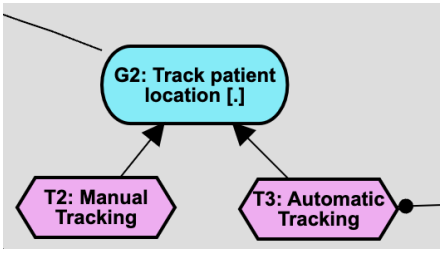
3.3.1.1 Goal types

In our proposal, goals are classified into two types. **Achieve goals** represent the default type, requiring a single decision that leads to a terminal system action. Their objective is to reach a defined achieved state, exploring alternative strategies as needed while respecting runtime constraints. In contrast, **Maintain goals** extend the notion of Achieve goals: they are governed by the same runtime constraints but must be pursued continuously to preserve a specific system state within a desired configuration until a given maintenance condition is met. Figures 3.3a and 3.3b illustrate examples of each goal type, depicting their respective visual representations.

Maintain goals are particularly relevant to desideratum D3 (Goal Status Tracking) because their continuous nature requires explicit monitoring of status changes over time. Unlike Achieve goals—which follow a linear lifecycle from activation to fulfillment—Maintain goals operate cyclically, transitioning between satisfied and violated states as system conditions evolve. This dynamic behavior demands continuous status tracking and reactivation mechanisms to ensure that the maintained condition remains valid throughout execution.



(a) Achieve goal in i* model for the TAS system Figure 5.1



(b) Maintain goal in i* model for the TAS system Figure 5.1

Figure 3.3: Achieve and maintain goals

3.3.1.2 Goal properties

To satisfy desideratum D2 (Goal Properties), the goal node is extended with quantitative attributes such as `utility` and `cost`. These properties capture the trade-offs between alternative goal variants, expressing how desirable or resource-intensive a particular option is under given runtime conditions. Their inclusion enables the system to reason quantitatively during decision-making, balancing benefit and effort when selecting among

competing goals. In the PRISM representation, these attributes are encoded as numerical rewards and penalties, allowing probabilistic model checking to evaluate expected outcomes, optimize strategy selection, and formally verify that runtime adaptations align with the modeled trade-offs.

Maintain goal In addition to the standard `context` property (see Section 3.3.5), maintain goals include a second guard, the `maintain` property, which follows the same Boolean syntax but serves a distinct semantic purpose. While the `context` guard determines when the goal may be activated, the `maintain` guard defines a persistent condition that must remain true for the goal to be considered achieved. In practice, this guard substitutes the standard semantics of the `achieved` state for maintain goals: the goal is regarded as continuously achieved only while its `maintain` condition holds. Whenever this condition evaluates to false at runtime, the controller reactivates the maintain goal, initiating corrective actions to restore the desired system state. This mechanism enables the model to represent ongoing objectives—such as maintaining safety constraints, resource levels, or performance thresholds—ensuring that the generated PRISM model faithfully captures both transient achievement and continuous maintenance behaviors.

3.3.1.3 Goal dependencies

To satisfy desideratum D4 (Goal Interdependencies), the model goal node is extended with a custom `dependsOn` property that explicitly encodes the relationships among goals within the hierarchy. This property establishes that the activation of one goal may depend on the state of another, introducing causal and temporal coupling between related objectives. Such dependencies are critical for representing realistic adaptation logic, where the pursuit of one goal is only meaningful once its prerequisite conditions are satisfied or where the failure of a supporting goal propagates upward to affect higher-level intentions.

Operationally, these dependency relationships constrain the decision-making process during runtime reasoning, ensuring that the system evaluates feasible goals based on their contextual readiness and dependency status. In the generated PRISM models, `dependsOn` links are implemented through synchronized transitions with guarded commands that enforce these logical constraints. This design enables the probabilistic model checker to evaluate dependency-driven dynamics—such as cascading failures or blocked goal activations—thereby supporting verification of both causal consistency and dependency resilience within the adaptive behavior.

Listing 3.1 illustrates the generic PRISM module template used as the foundation for transforming goal elements into formal specifications. This template defines the common control structure shared across all goal types, including variables for pursuit and achieve-

ment states (lines 2-3), and command patterns for activation, completion, and skipping behavior. The placeholders `<runtime_annotation>` and `<context>` are PRISM boolean sentences indicate where specific execution semantics and contextual constraints—defined by the goal's annotations—are later integrated during code generation.

Bellow we describe the structure and semantics of the goal module template presented in Listing 3.1. This explanation clarifies how each section of the template contributes to maintaining synchronization across the hierarchical pursuit–achievement chain that connects parent and child goals during runtime execution.

3.3.1.4 PRISM Goal Module

The module G_n, defined between lines 1 (`module Gn`) and 12 (`endmodule`), represents the behavior of a single goal node in the target PRISM model. It is organized into five logical sections:

1. **Variable definition (lines 2–3)** – Each goal defines two state variables, `Gn_pursued` and `Gn_achieved`, which represent its current lifecycle status. Although declared as integer ranges, they operate as Boolean flags indicating whether the goal is being pursued or has been achieved. These variables synchronize decision flow between parent and child goals by controlling when a goal can initiate or terminate its actions.

2. **Pursue transitions (lines 5–7)** – The first `[pursue_Gn]` label (line 5) activates the goal once its dependency and contextual conditions are satisfied (`<dependency_assertion>` and `<context>`). Once active, the subsequent `[pursue_Gi]` and `[pursue_Gj]` transitions (lines 6–7) propagate this activation downward, triggering the initial pursue transitions of their respective subgoals. This recursive activation continues through the goal hierarchy until a terminal *task* node is reached, at which point execution is delegated to the `ChangeManager` module responsible for operational enactment.

3. **Achievement synchronization (line 9)** – The `[achieved_Gn]` label ensures proper synchronization between a goal and its children. A parent goal is marked as achieved only when its subgoals have completed, with the logical operator (`&` for AND relations or `|` for OR relations) determining how the joint satisfaction condition is evaluated. This mechanism guarantees that causal and temporal dependencies are respected across goal layers.

4. **Skip transition (line 11)** – The `[skip_Gn]` transition resets the goal to a non-pursued state if none of its children are selected for pursuit after activation. This

enables the controller to revisit the goal in subsequent decision cycles, ensuring continuous adaptability and re-evaluation under changing conditions.

5. **Goal formulas (lines 13–15)** – Each goal declares a mandatory `Gn_achievable` formula capturing its achievability semantics. For **AND** goals, this value corresponds to the product of the achievability of all child goals. For **OR** goals, it is computed as the sum of the children's achievability values minus their product, while goals with a single child simply inherit that child's achievability. In addition, every **maintain** goal defines a `Gn_achieved_maintain` formula specifying the conditions under which the goal should be pursued or considered satisfied.

Through this recursive pursue–achieve interaction pattern, the template enforces both downward propagation of intentions and upward aggregation of results, preserving the semantics of goal refinement and runtime reasoning established in the original model.

```
1  module Gn
2    Gn_pursued : [0..1] init 0;
3    Gn_achieved : [0..1] init 0;
4
5    [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 & <dependency_assertion> & <context>
       ↪ -> (Gn_pursued'=1);
6    [pursue_Gi] Gn_pursued=1 & <runtime_constraint> -> true;
7    [pursue_Gj] Gn_pursued=1 & <runtime_constraint> -> true;
8
9    [achieved_Gn] Gn_pursued=1 & (Gi_pursued=1 <link_condition_token> Gj_pursued
       ↪ =1) -> (Gn_pursued'=0) & (Gn_achieved'=1);
10
11   [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0 -> (Gn_pursued'=0);
12 endmodule
13 formula Gn_achievable = <achievability_formula>
14 // optional formula for maintain goals
15 formula Gn_achieved_maintain = <maintainCondition>
```

Listing 3.1: Generic PRISM module template for a goal `Gn` with two subgoals `AND(Gi, Gj)`

In the subsequent subsections, this base structure is progressively specialized to reflect different runtime relations (e.g., sequential, parallel, or alternative decompositions). Synchronization among the `pursue` commands ensures that subgoals are activated only under the conditions prescribed by their parent goal's logic, preserving causal and temporal dependencies within the generated PRISM model.

3.3.2 AND/OR Links

The next foundational element required by an EDGE-compliant goal modeling tool is support for connecting reasoning elements (goals and tasks) through **AND** and **OR** links. These operators follow the traditional GORE semantics of goal decomposition, establishing hierarchical dependencies among model elements.

While **AND** links define conjunctive refinements that must all be satisfied to fulfill a parent goal, **OR** links introduce alternative refinements, enabling the model to represent multiple strategies for achieving the same objective. This disjunctive structure is essential for adaptability and directly fulfills desideratum D1 (Non-Idempotent Goal Variants), as it defines the decision points where the controller must select among goal variants according to their contextual and runtime conditions as well as quantitative properties.

From a transformation perspective, AND/OR junctions are encoded as synchronized transitions in the PRISM model, with guard conditions ensuring that AND refinements proceed only when all subgoals are achievable and OR refinements trigger a probabilistic or conditional choice among variants. They define the condition constraint for the goal achievement as previously explained in the section 3.3.1.4. This encoding preserves the decision semantics of the original goal model, allowing the probabilistic controller to evaluate and pursue strategies dynamically during execution.

In the following subsections, we examine how these syntactical elements combine to form higher-level semantic constructs—such as contextual dependencies, runtime annotations and quantitative properties—that rule goal activation and coordination of the goal management controller decisions.

3.3.3 Runtime annotations

The final element incorporated into the model is the synchronization logic defined by runtime annotations. These annotations capture the temporal and logical relationships among goals and tasks, specifying the order, concurrency, or conditional execution of elements at runtime. In our approach, each annotation is written directly alongside the goal name, enclosed in brackets—for example, `G1: Provide Sample [G2; G3]`, which denotes a sequential dependency between the subgoals `G2` and `G3`. This syntax associates the execution structure exclusively with the annotated goal, clarifying which node governs the sequencing or synchronization of its subgoals. By embedding the runtime operator in the goal label itself, the model maintains readability while ensuring that execution semantics remain unambiguous and goal-scoped.

During transformation, these annotations are translated into guarded commands within the PRISM model, where the guards of the `pursue` actions encode the corresponding tem-

poral and causal constraints. Table 3.1 summarizes the supported runtime annotations according to the relation type of each goal.

Structure	**Notation**	**Semantics**
Sequential AND	`AND(n1;n2)`	Sequential achievement of n_1 and n_2.
Interleaved AND	`AND(n1#n2)`	Parallel achievement of n_1 and n_2.
Alternative OR	`OR(n1|n2)`	Any child may be pursued, but only one at a time.
Choice OR	`OR(+)`	Exclusive choice between n_1 and n_2. The controller remembers the choice.
Degradation OR	`OR(n1->n2)`	If n_1 degrades, fallback to n_2, and so forth.
Degradation OR with retries	`OR(n1@k->n2)`	Retries n_1 k times before falling back to n_2.

Table 3.1: Supported goal composition structures and their semantics in the extended EDGE notation.

In the subsequent paragraphs, we detail the operational semantics of each runtime annotation through multiple complementary representations. For each relation type, we first describe its intended execution semantics within the goal hierarchy, followed by a corresponding diagram illustrating its structure in the extended goal model. Next, we provide a state transition diagram that visualizes the runtime behavior and causal dependencies among subgoals, and finally, a PRISM code excerpt that demonstrates how these semantics are encoded in the synthesized controller model. Throughout these explanations, n_i denotes a child node of goal G_n, where n_i may correspond to either a subgoal or a task element.

3.3.3.1 Sequential AND

A sequential AND composition specifies that a parent goal G_n activates its children in a fixed order $G_i \rightarrow G_j \rightarrow \cdots \rightarrow G_k$. Only the first child is enabled initially; each subsequent child becomes enabled *iff* its immediate predecessor has been achieved. Formally, the enablement condition for G_{r+1} depends on the achievement of G_r, and G_n is achieved only after the last child has completed. Operationally, this enforces causal ordering: the runtime tuple $\tau = (f, p)$ advances the frame when moving to the next child, e.g., $\tau' = (f + 1, p)$, reflecting that G_{r+1} cannot start before G_r ends.

Figures 3.4 and 3.5 illustrate these semantics. Figure 3.4 shows a goal model annotated with the sequential AND relation. Figure 3.5 depicts the corresponding state-transition view, where nodes $n_1, n_2, \ldots, n_k$ activate strictly in sequence as the frame component of τ increments.

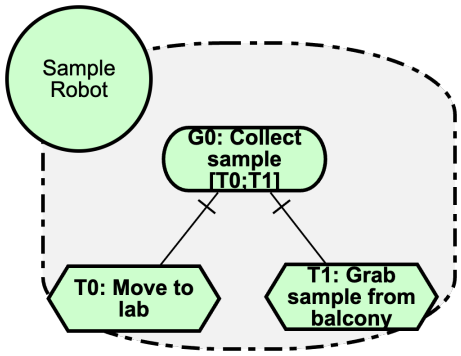


Figure 3.4: Goal model example with sequential AND decomposition over child goals.

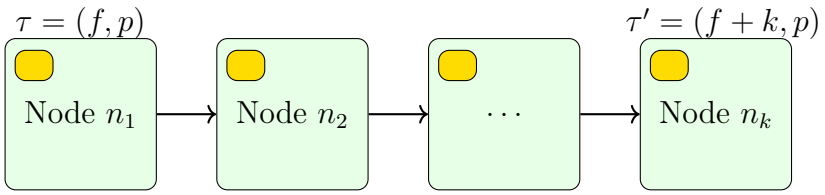


Figure 3.5: Sequential AND execution over nodes $n_1 \to n_2 \to \cdots \to n_k$ with updated frame in τ'.

In PRISM, runtime annotations modify the `[pursue_Gi]` and `[pursue_Gj]` labels in the base goal template (Listing 3.1) to encode the execution constraints imposed by the annotation type. For sequential AND compositions, these annotations enforce a strict ordering among child goals, ensuring that each subgoal is activated only after its predecessor has been successfully achieved.

Listing 3.2 illustrates this behavior. Lines 1–3 and 15 correspond to the base module structure, defining the goal's state variables and delimiters. Line 5 initializes the goal's pursuit once it becomes eligible, setting `Gn_pursued` to 1. Lines 7–9 specify the sequential activation logic for the child goals: `[pursue_Gi]` (line 6) enables the first subgoal when the parent goal is active and `Gi` has not yet been achieved. `[pursue_Gj]` (line 7) introduces an additional guard, requiring that `Gi` has already been achieved before `Gj` can start. `[pursue_Gk]` (line 8) extends this dependency chain, allowing pursuit only once all previous subgoals have completed.

The synchronization label `[achieved_Gn]` (line 10) defines the parent goal's completion condition: it is achieved only when all children have been pursued and completed according to the defined ordering.

This implementation operationalizes the semantics of the sequential AND runtime annotation, ensuring that the PRISM model preserves both the causal ordering and dependency semantics of the original goal structure.

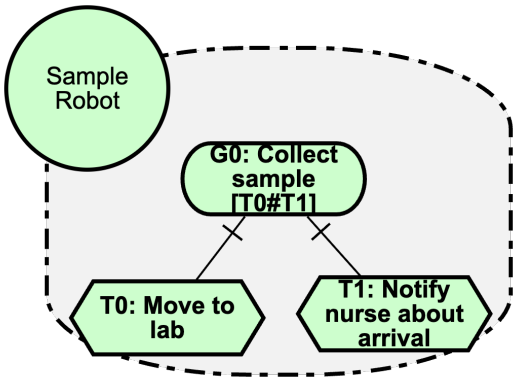
```
1  module Gn
2    Gn_pursued : [0..1] init 0;
3    Gn_achieved : [0..1] init 0;
4
5    [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 <dependency_assertion> & <context> ->
       ↪  (Gn_pursued'=1);
6    [pursue_Gi] Gn_pursued=1 & Gi_achieved=0 -> true;
7    [pursue_Gj] Gn_pursued=1 & Gi_achieved=1 & Gj_achieved=0 -> true;
8    [pursue_Gk] Gn_pursued=1 & Gi_achieved=1 & Gj_achieved=1 & Gk_achieved=0 ->
       ↪ true;
9
10   [achieved_Gn] Gn_pursued=1 & (Gi_pursued=1 & Gj_pursued=1 & Gk_pursued=1)
11       -> (Gn_pursued'=0) & (Gn_achieved'=1);
12
13   [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0 & Gk_pursued=0
14       -> (Gn_pursued'=0);
15 endmodule
```

Listing 3.2: PRISM module for a sequential AND goal G_n with children $G_i \rightarrow G_j \rightarrow G_k$

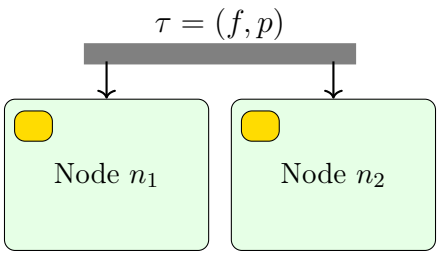
3.3.3.2 Interleaved AND

An interleaved AND (AND) composition defines a parent goal G_n whose child goals $G_i, G_j, \ldots, G_k$ must all be pursued and achieved, but without any imposed execution order. Unlike the sequential AND structure, interleaved AND allows concurrent or arbitrarily ordered pursuit of subgoals once the parent G_n is active. Each child n_i can begin independently as soon as the parent has entered its **pursued** state, and the parent goal transitions to the **achieved** state only after all its children have individually completed. This captures the semantics of parallel or unordered goal realization within a single execution frame $\tau = (f, p)$, with synchronization occurring only upon completion, when the system advances to $\tau' = (f + m, p)$, where m is the amount of steps necessary to reach the synchronization step.

Figures 3.6a and 3.6b illustrate these semantics. Figure 3.6a presents a goal model annotated with the interleaved AND relation, while Figure 3.6b depicts the corresponding execution structure, where nodes $n_1, n_2, \ldots, n_k$ can be pursued concurrently within the same frame and synchronize only upon collective achievement.



(a) Interleaved AND goal module with two task children



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

Runtime annotations define the execution constraints that govern how a goal selects and coordinates among its variants. In the case of an `Interleaved AND` composition, no sequential or conditional restrictions are imposed—since all child goals are meant to be pursued concurrently once the parent goal becomes active. Thus, the interleaved relation expresses full parallelism in pursuit behavior.

Listing 3.3 illustrates this implementation in PRISM. Lines 1–3 and 12 define the module boundaries and the standard state variables, while line 5 (`[pursue_Gn]`) initializes the pursuit of the parent goal under the satisfaction of its dependency and contextual conditions. Lines 6–7 encode the concurrent activation of the child subgoals `Gi` and `Gj`: both may be pursued as soon as the parent goal G_n is active, without dependency on one another. This reflects the semantics of interleaving—each subgoal executes independently within the same decision frame.

The synchronization condition is expressed in line 9, where the `[achieved_Gn]` label ensures that G_n transitions to the achieved state only when all child subgoals have individually reached completion (`Gi_achieved = 1` and `Gj_achieved = 1`). Finally, line 11 provides the `[skip_Gn]` transition, which resets the goal if no child subgoal was initiated, maintaining consistency in the control cycle.

This encoding captures the semantics of the interleaved AND annotation: all children of a goal may be pursued in parallel, but synchronization and completion occur only after every subgoal has been successfully achieved.

```
module Gn
  Gn_pursued : [0..1] init 0;
  Gn_achieved : [0..1] init 0;

  [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 &<dependency_assertion> & <context>
    ↪ -> (Gn_pursued'=1);
  [pursue_Gi] Gn_pursued=1 & Gn_achieved=0 -> true;
  [pursue_Gj] Gn_pursued=1 & Gn_achieved=0 -> true;

  [achieved_Gn] Gn_pursued=1 & (Gi_achieved=1 & Gj_achieved=1)
      -> (Gn_pursued'=0) & (Gn_achieved'=1);

  [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0
      -> (Gn_pursued'=0);
endmodule
```

Listing 3.3: PRISM module for an Sequential AND goal G_n with children G_i and G_j

3.3.3.3 Choice OR

An choice OR composition defines a parent goal G_n that may pursue exactly one among its children $G_i, G_j, \ldots, G_k$. It captures the semantics of a *non-idempotent variants*, as formalized in the EDGE desiderata [9]. This construct directly supports **Desideratum D1** by enabling explicit modeling of alternative, non-repeatable goal variants that represent commitment decisions made under uncertainty. Once a child goal is chosen, this decision remains fixed for the entire lifecycle of the parent goal, prohibiting any reevaluation or switching across execution frames.

This persistent-choice behavior contrasts with the dynamic `Any OR` composition, which allows re-selection at runtime. The Alternative OR therefore models situations where choices are irreversible or context-dependent scenarios that introduce structural dependencies between subgoals, addressing **Desideratum D4**.

Semantically, the Alternative OR enforces both determinism and exclusivity: only one child can be pursued, and the parent goal is achieved if and only if that chosen subgoal succeeds. It also relates to **Desideratum D3**, since the system must track the pursued and achieved states over time to prevent any re-selection once commitment has occurred.

Figures 3.7 and 3.8 illustrate these semantics. Figure 3.10 shows a goal model annotated with the alternative OR relation, depicting a single, committed choice between subgoals. Figure 3.8 highlights two execution frames, showing that once the controller selects a node—e.g., n_2 at frame $\tau = (f, p)$—that selection remains valid across subsequent

decision frames $\tau' = (f + n, p)$, preserving consistency and decision stability in line with D1 and D4.

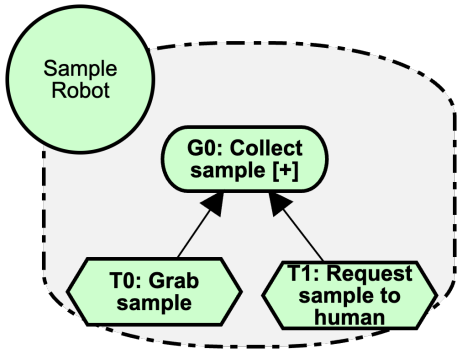


Figure 3.7: Any OR goal with two task children.

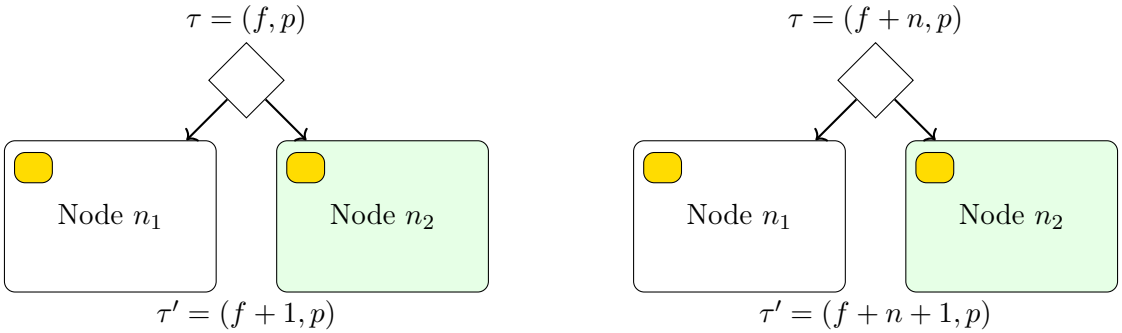


Figure 3.8: Two decision cycles for alternative nodes $n_1 \mid n_2$. In each frame, the controller always selects a different node

Listing 3.4 illustrates the PRISM encoding for the `Choice OR` composition, which represents non-idempotent goal variants. This structure enforces mutual exclusivity among subgoals, ensuring that only one alternative can be pursued throughout the parent goal's lifecycle. Once a choice is made, it is remembered and cannot be altered, reflecting the commitment semantics central to this runtime annotation.

Lines 1–3 and 13 define the module boundaries and the standard state variables. Line 4 introduces an additional variable, `Gn_chosen`, which records the identifier of the selected subgoal, preserving decision persistence across execution frames. The parent goal's pursuit is initialized on line 6 through the `[pursue_Gn]` command, which activates the decision process once the parent goal becomes eligible. Lines 7–8 specify the guarded transitions for `[pursue_Gi]` and `[pursue_Gj]`, where each guard ensures that only one subgoal can be pursued at a time—if one is chosen, the other becomes permanently excluded.

The synchronization logic is defined on line 10, where the `[achieved_Gn]` label marks the parent as achieved when the selected subgoal successfully completes (`Gi_achieved =`

1 or Gj_achieved = 1). Finally, the [skip_Gn] command (line 12) resets the goal to a non-pursued state if no child goal was selected, ensuring that the controller can reevaluate decisions in subsequent cycles.

```
1  module Gn
2    Gn_pursued : [0..1] init 0;
3    Gn_achieved : [0..1] init 0;
4    Gn_chosen : [0..2] init 0;
5
6    [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 &<dependency_assertion> & <context>
       -> (Gn_pursued'=1);
7    [pursue_Gi] Gn_pursued=1 & Gn_achieved=0 & Gn_chosen!=1 -> true;
8    [pursue_Gj] Gn_pursued=1 & Gn_achieved=0 & Gn_chosen!=0 -> true;
9
10   [achieved_Gn] Gn_pursued=1 & (Gi_achieved=1 | Gj_achieved=1) -> (Gn_pursued'
       =0) & (Gn_achieved'=1);
11
12   [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0 -> (Gn_pursued'=0);
13 endmodule
```

Listing 3.4: PRISM module for a Choice OR goal G_n where the controller remembers the chosen variant

This implementation operationalizes the semantics of the `Alternative OR` runtime annotation, maintaining exclusivity, decision persistence, and non-repetition of variants—core properties that satisfy the non-idempotent goal variant requirement defined in Desideratum D1.

3.3.3.4 Alternative OR

An `Alternative OR` composition defines a parent goal G_n that may pursue one of several alternative children $G_i, G_j, \ldots, G_k$, selecting among them dynamically at runtime. Unlike the `Choice OR`, which enforces a fixed and repeatable decision, the `Alternative OR` supports adaptive re-evaluation: the controller may choose a different subgoal in later decision frames based on changing environmental or contextual conditions. This construct captures a *reconfigurable disjunction* aligned with the EDGE desiderata [9].

Figures 3.10 and 3.9 illustrate these semantics. Figure 3.10 presents a goal model annotated with the `Any OR` relation, while Figure 3.9 depicts two decision frames in which the controller dynamically selects different subgoals based on context. In frame $\tau = (f, p)$, the system may pursue n_1; if conditions change in a later frame $\tau' = (f + n, p + n)$, it

may instead pursue n_2. This adaptive decision-making ensures responsiveness to changing environments while maintaining exclusivity in goal activation.

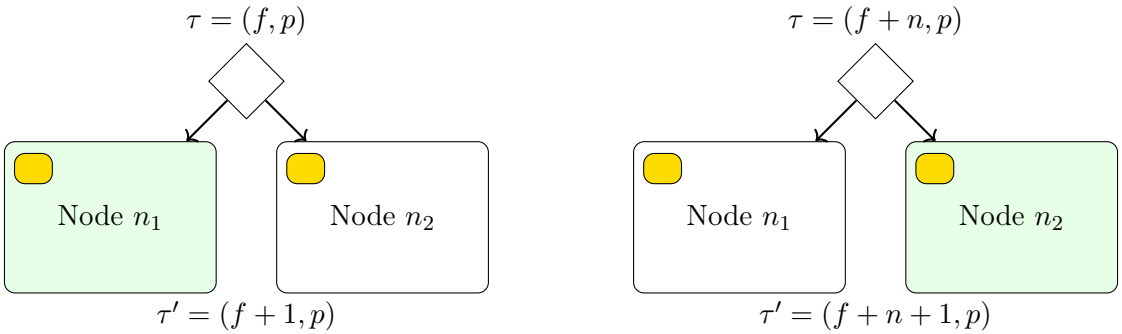


Figure 3.9: Two decision cycles for alternative nodes $n_1 \mid n_2$. In each frame, the controller selects one subnode based on runtime context.

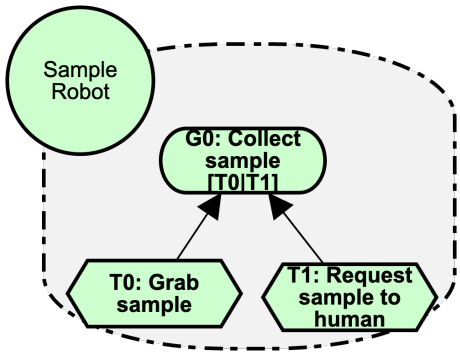


Figure 3.10: Alternative OR goal with two task children

Listing 3.5 illustrates the PRISM encoding of the `Any OR` composition, which models idempotent goal variants that can be reselected dynamically at runtime. This structure enables the controller to pursue one subgoal per decision frame, while allowing future re-evaluation based on changing contextual conditions.

Lines 1–3 and 14 define the module boundaries and initialize the standard state variables: `Gn_pursued` and `Gn_achieved`, which track whether the parent goal is currently being pursued or has already been completed. The `[pursue_Gn]` command in line 5 activates the parent goal when its enabling conditions are satisfied, initiating the decision process for selecting among its subgoals.

Lines 7 and 8 define the `[pursue_Gi]` and `[pursue_Gj]` commands, which govern subgoal activation. Each guard ensures that only one child can be pursued at a time: for example, `[pursue_Gi]` executes only if G_j is not currently being pursued, and vice versa. This conditional exclusivity enforces idempotent behavior—although subgoal selection may vary between decision frames, it remains exclusive within a single frame.

Line 9 defines the synchronization condition through the `[achieved_Gn]` command. Here, the parent goal G_n transitions to the achieved state once any child subgoal (G_i or G_j) completes, as expressed by the disjunction `(Gi_achieved = 1 | Gj_achieved = 1)`. This condition ensures that the parent reflects successful completion of its current alternative without requiring all subgoals to finish.

Finally, line 12 introduces the `[skip_Gn]` transition, which resets the parent goal when no subgoal is pursued, maintaining consistency across decision cycles. This guarantees that the controller can re-evaluate alternative choices in subsequent frames, enabling the adaptive, reconfigurable behavior characteristic of the `Any OR` construct.

This implementation thus captures the semantics of the `Any OR` relation—dynamic re-selection of goal variants, exclusivity within frames, and contextual adaptability—fulfilling the requirements of Desiderata D1 and D3.

```
1  module Gn
2    Gn_pursued : [0..1] init 0;
3    Gn_achieved : [0..1] init 0;
4
5    [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 -> (Gn_pursued'=1);
6    [pursue_Gi] Gn_pursued=1 & Gn_achieved=0 & Gj_pursued=0 -> true;
7    [pursue_Gj] Gn_pursued=1 & Gn_achieved=0 & Gi_pursued=0 -> true;
8
9    [achieved_Gn] Gn_pursued=1 & (Gi_achieved=1 | Gj_achieved=1)
10       -> (Gn_pursued'=0) & (Gn_achieved'=1);
11
12   [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0
13       -> (Gn_pursued'=0);
14 endmodule
```

Listing 3.5: PRISM module for an Alternative OR goal G_n with non-exclusive children G_i and G_j

3.3.3.5 Degradation OR

A Degradation OR composition defines a parent goal G_n that pursues its children $G_i, G_j, \ldots, G_k$ according to a predefined priority order, providing a deterministic fallback mechanism in case of failure. It is denoted as $\texttt{Gn}[n_i \rightarrow n_j \rightarrow \ldots \rightarrow n_k]$, where the controller always attempts the highest-priority child n_i first. If that subgoal fails—either implicitly or after exceeding its retry limit—the controller degrades its selection to the next available subgoal n_j, continuing this fallback sequence until one succeeds or all have failed. This semantics ensures robust execution continuity under fault or uncertainty.

Retry limits can be explicitly specified within the notation. For instance, `Gn[`n_1@3 $\rightarrow$ n_2@2 $\rightarrow$ n_3`]` declares that n_1 should be retried up to three times before degrading to n_2, which may be retried twice before defaulting to n_3. This controlled degradation behavior contributes to **Desideratum D1**, by supporting non-idempotent goal variants that reflect bounded persistence under failure, and to **Desideratum D3**, by requiring the system to monitor the pursuit and retry states of each subgoal. Moreover, the prioritized ordering and fallback dependencies among children operationalize **Desideratum D4**, which captures explicit inter-goal relationships and their influence on runtime decision flow.

Figures 3.11 and 3.12 illustrate these semantics. Figure 3.11 shows a goal model annotated with a Degradation OR decomposition, while Figure 3.12 depicts the corresponding controller behavior: the system repeatedly attempts the first subgoal n_1 for up to r retries before degrading to the next subgoal n_2, ensuring a controlled and memory-aware recovery strategy.

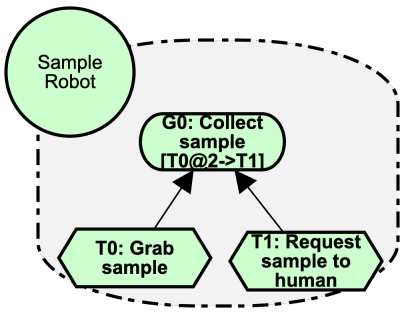


Figure 3.11: Degradation OR goal with two task children

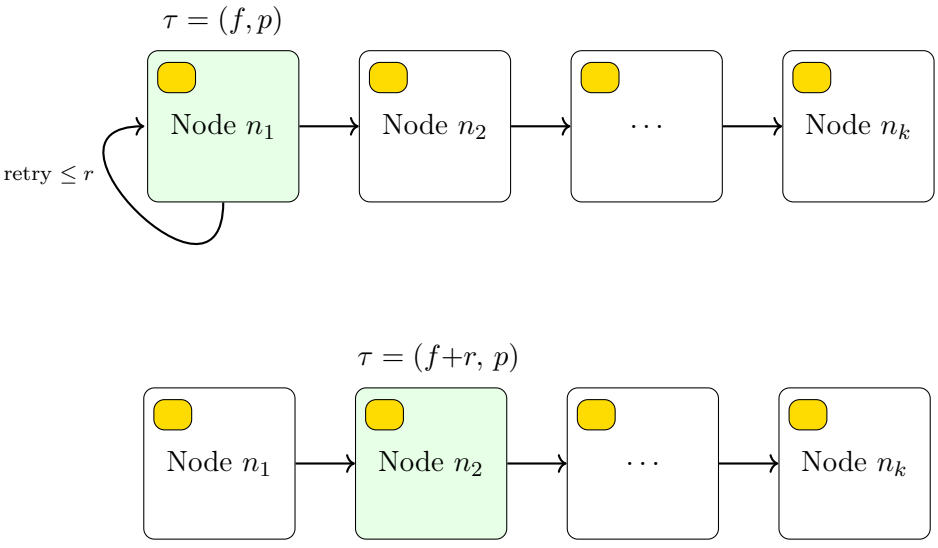


Figure 3.12: Controller decision process in a degradation OR structure, repeating n_1 up to r times before transitioning to n_2.

Listing 3.6 shows the PRISM encoding for the `Degradation OR` composition, which models prioritized goal execution with bounded retries and deterministic fallback. This construct formalizes a structured recovery process in which the controller attempts a sequence of subgoals $(G_i, G_j, \ldots)$ in decreasing order of preference until one succeeds or all fail.

Lines 1–3 and 16 define the module boundaries and the standard goal state variables, `Gn_pursued` and `Gn_achieved`, which indicate whether the parent goal is currently active or has been completed. The parent activation is triggered on line 5 through the `[pursue_Gn]` command, setting `Gn_pursued` to 1 once the goal is selected for pursuit.

Lines 6 and 7 capture the core logic of the degradation mechanism. The first child goal G_i (line 6) is always pursued initially, provided the parent is active and the next alternative G_j is not yet being pursued. The second child G_j (line 7) becomes eligible only when G_i has failed at least twice (`Gi_failed >= 2`), expressing the bounded retry semantics that govern fallback decisions. This conditional activation ensures deterministic priority handling—G_j can only be triggered once the retry threshold for G_i has been exceeded. In cases where no retry limit is defined for a subgoal in the degradation chain, the retry guard (`Gi_failed >= r`) is omitted from the generated PRISM code, resulting in an immediate degradation after the first failure. This guarantees consistent semantics between bounded and unbounded fallback configurations.

Line 9 defines the parent's success condition through the `[achieved_Gn]` command. The parent goal G_n is marked as achieved when any of its subgoals (G_i or G_j) reaches completion, as denoted by the disjunction `(Gi_achieved=1 | Gj_achieved=1)`. This synchronization guarantees that successful fallback execution is properly propagated upward in the goal hierarchy.

Finally, line 11 introduces the `[skip_Gn]` transition, which resets the parent goal to a non-pursued state when no subgoals remain active or retriable. This termination condition prevents indefinite retries and allows the controller to resume higher-level decision cycles.

This implementation operationalizes the semantics of the `Degradation OR` construct –priority-based fallback, bounded retry, and causal dependency among subgoals–thereby fulfilling Desiderata D1, D3, and D4. It enables robust, memory-aware adaptation in which failure-handling and decision persistence are explicitly encoded in the synthesized PRISM controller.

```
1 module Gn
2   Gn_pursued  : [0..1] init 0;
3   Gn_achieved : [0..1] init 0;
4
5   [pursue_Gn] Gn_pursued=0 & Gn_achieved=0 -> (Gn_pursued'=1);
```

```
  [pursue_Gi] Gn_pursued=1 & Gn_achieved=0 & Gj_pursued=0 -> true;
  [pursue_Gj] Gn_pursued=1 & Gn_achieved=0 & Gi_pursued=0 & Gi_failed >= 2 ->
    ↪ true;

  [achieved_Gn] Gn_pursued=1 & (Gi_achieved=1 | Gj_achieved=1)
      -> (Gn_pursued'=0) & (Gn_achieved'=1);

  [skip_Gn] Gn_pursued=1 & Gi_pursued=0 & Gj_pursued=0
      -> (Gn_pursued'=0);
endmodule
```

Listing 3.6: PRISM module for a Degradation-OR goal G_n, where G_j is pursued only if G_i fails after two attempts

3.3.4 Tasks

Tasks constitute the second fundamental element of the model, representing the concrete actions the system executes to fulfill operational objectives. In the proposed EDGE-compliant goal model, tasks mark the operationalization of goal decompositions, bridging the intentional layer of goal reasoning with the executable layer of system behavior.

This separation follows the MORPH principle of decoupling the *Goal Management Layer*—which decides *what* to pursue—from the *Strategy Enactment Layer*, which determines *how* to achieve it [3]. Goals may thus be recursively decomposed into subgoals until reaching one or more terminal *task* nodes. Once selected, each task's execution is delegated to the managed system, marking the transition from reasoning to action.

Tasks as constrained by contextual guards and quantitative properties (e.g., success probability, cost, duration) of their parent goals, capturing operational uncertainty and supporting desiderata D2 (Goal Properties) and D3 (Goal Status Tracking). Their outcomes directly influence the runtime evaluation of goal satisfaction.

In the transformation pipeline, each task is mapped into a single PRISM module (the `ChangeManager`) encoding its probabilistic behavior—such as success and failure transitions—and synchronized with higher-level goals through shared guards and state variables. This alignment preserves consistency between design intent and executable behavior, enabling verification of both individual task reliability and overall goal achievability. Figure 3.3b illustrates a single goal decomposed into two task nodes, each representing an alternative strategy for achieving the same objective.

Listing 3.7 illustrates the structure and semantics of the `ChangeManager` module, which governs the execution-level behavior of tasks within the translated PRISM model. This

module is responsible for synchronizing task-level dynamics—pursuit, achievement, and failure—with the decision logic defined in the corresponding goal modules.

3.3.4.1 PRISM ChangeManager Module

The `ChangeManager` module models the operational layer of the system, where abstract goals are enacted as concrete, executable actions. It captures how tasks are initiated, completed, or retried during runtime, providing the behavioral link between high-level goal reasoning and system-level actuation. Defined between lines 1 (`module ChangeManager`) and 15 (`endmodule`), it governs task pursuit, achievement, and failure through synchronized state transitions that reflect the MORPH architecture's *Strategy Enactment Layer*. Structurally, the module is organized into two logical sections:

1. **Variable definition (lines 2–5)** — Each task in the model, such as `T0` and `T1`, declares two integer variables: `T`i`_pursued` and `T`i`_achieved`. These variables act as Boolean state indicators, tracking whether a task is currently being executed or has successfully completed. Their values are monitored by the higher-level goal modules to determine when goals can be marked as achieved or must be re-evaluated.

2. **Execution transitions (lines 7–10)** — The `[pursue_T`i`]` labels initiate task execution once triggered by the parent goal's pursue condition, aligning operational behavior with goal-level intentions. The `[achieved_T`i`]` transitions mark successful completion of a task, while the `[failed_T`i`]` transitions reset its pursued state (`T`i`_pursued' = 0`) in the event of failure. This pattern ensures that tasks remain synchronized with the status of their corresponding goals, enabling consistent propagation of success or failure information upward through the goal hierarchy.

Semantically, the `ChangeManager` module operationalizes the *Strategy Enactment Layer* of the MORPH reference architecture [3]. It bridges high-level decision logic from the goal modules to executable task behavior, ensuring that probabilistic verification in PRISM can account for both individual task outcomes and their cumulative effect on overall goal satisfaction.

```
1 module ChangeManager
2   T0_pursued : [0..1] init 0;
3   T0_achieved : [0..1] init 0;
4   T1_pursued : [0..1] init 0;
5   T1_achieved : [0..1] init 0;
6
7   [pursue_T0] true -> true;
```

```
8   [achieved_T0] true -> true;
9   [failed_T0] T0_pursued=1 & T0_achieved=0 -> (T0_pursued'=0);
10
11  [pursue_T1] true -> true;
12  [achieved_T1] true -> true;
13  [failed_T1] T1_pursued=1 & T1_achieved=0 -> (T1_pursued'=0);
14  // ... repeats for every task in the model
15 endmodule
```

Listing 3.7: `ChangeManager` module representing tasks T0 and T1

3.3.5 Contextual Guards

Both goals and tasks can be extended with a `context` property that specifies the environmental or system conditions under which the element may be activated. In the PRISM model, this property is encoded as a Boolean guard expression, determining whether a goal can be pursued or a task executed given the current state of relevant resources or variables.

During model interpretation, a dedicated parser extracts and validates the variables referenced in each contextual expression, ensuring syntactic correctness and consistency with the declared resources. These validated expressions are then passed to the template engine during code generation, where they form the conditional logic for the `pursue` commands in the resulting PRISM goal modules (see § 3.5.3.4) and a set of variables in the `System` module (see § 3.5.2).

The inclusion of contextual guards directly supports desideratum D3 (Goal Status Tracking) and D4 (Goal Interdependencies). By making activation and deactivation explicitly dependent on runtime conditions, the model can continuously assess which goals are achievable and which must remain dormant. This mechanism enables adaptive control decisions that reflect the evolving operational context, ensuring that the translated PRISM model captures both the structural dependencies and the dynamic responsiveness of the original goal model.

3.3.6 Resources

Resources constitute the third essential element of an EDGE-compliant goal model. They represent measurable system or environmental variables (e.g. battery level, bandwidth, temperature, or operator availability) that influence the feasibility and prioritization of goals and tasks. By explicitly modeling these quantities, resources provide the contextual basis for runtime reasoning and adaptive decision-making.

From a modeling perspective, each resource is defined with properties specifying its data type (`boolean` or `int`) and valid range, ensuring that its representation in the PRISM model reflects its operational semantics. While goals and tasks may reference resources through contextual guards, resources can only be declared when linked to tasks, since tasks are the elements that interact directly with the environment and modify resource values. Figure 3.13 illustrates a two resources linked to tasks in the EDGE model of the Lab Samples exemplar [15], demonstrating how resources are related to task elements. Table 3.2 summarizes the defining properties for each resource type required by the model.

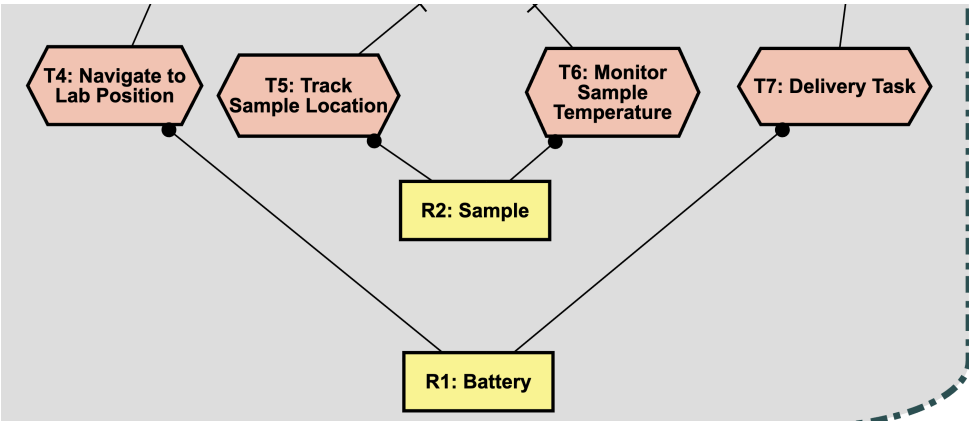


Figure 3.13: Resource example; R1 is an int resource and R2 is a boolean resource

Type	**Required Properties**
`bool`	`initialValue`
`int`	`initialValue`, `lowerBound`, `upperBound`

Table 3.2: Required custom properties for resources based on their type.

This bidirectional relationship captures the feedback loop between reasoning and execution: pursue guards may depend on resource states, whereas tasks update those states as side effects (e.g., consuming energy or releasing memory), potentially triggering strategy re-evaluation. Resources, therefore, play a central role in fulfilling desiderata D3 (Goal Status Tracking) and D4 (Goal Interdependencies), functioning as shared state variables that propagate outcomes across goals. During transformation, resources are encoded as system variables within the `System` module, with guarded commands enforcing contextual constraints. This encoding allows probabilistic model checking to assess how resource fluctuations influence the system's capacity to Maintain or Achieve goal types, thereby validating adaptive behavior under uncertainty.

3.3.6.1 PRISM System Module

The `System` module defines the global state variables representing the system's contextual and environmental configuration, which serve as inputs for the controller's decision logic. These variables include the resources and contextual parameters upon which goals and tasks depend for activation. Declared between lines 1 (`module System`) and 4 (`endmodule`), this module encapsulates the dynamic aspects of the environment that influence goal feasibility and strategy selection.

Each resource is declared according to its type, either as a bounded integer or as a Boolean variable, following the general form:

$$resource\_\langle id \rangle : \langle range \mid bool \rangle \texttt{ init } \langle value \rangle;$$

As illustrated in Listing 3.8, the resource `R1` is modeled as an integer variable with an initial value of 50, while `R2` is a Boolean variable initialized as `true`. These resource definitions allow PRISM to evaluate contextual guards and track state changes induced by task execution, ensuring that adaptive behavior in the synthesized model reflects both internal system dynamics and external environmental conditions.

```
1 module System
2   R1: [0..100] init 50;
3   R2: bool init true;
4 endmodule
```

Listing 3.8: `System` module representing resource declarations

3.3.7 EDGE Goal Model: Required Elements and Relationships

To conclude the first stage of the translation pipeline (*Identification of Semantically Relevant Elements*), we distill the preceding definitions into a concise metamodel. Figure 3.14 summarizes the entities manipulated by our method (Goal, Task, Resource, Link, Runtime Annotation), their attributes, and the cardinalities that constrain valid models. This metamodel serves as the schema contract for the subsequent phases: it captures the semantics required by the EDGE desiderata (D1–D4), ensures traceability to PRISM constructs, and provides the structural basis for a semantics-preserving, template-based translation that will allow the later satisfaction of the desiderata D5.

Goals `Goal` is the primary reasoning element (`goalId`, `name`, `type` $\in$ {achieve, maintain}). Its `supported_props` include `context`, `utility`, and `cost` (D2), while `optional_props`

cover `maintainCondition` (only for *maintain* goals; D3), `maxRetries` (used by degradation/ retry patterns; D1, D3), and `dependsOn` (explicit inter-goal dependency; D4). Each goal may carry **at most one** `Runtime Annotation` that scopes the execution structure over its *own* children (e.g., sequential AND, interleaved AND, alternative/choice/degradation OR). This goal-scoped design keeps execution semantics unambiguous and traceable in the PRISM encoding.

Tasks `Task` represents executable endpoints (`taskId`, `name`). Like goals, tasks admit `context`, `utility`, and `cost` in `supported_props`, allowing quantitative trade-offs and context gating at the enactment level (D2, D3). In PRISM, each task corresponds to enactment logic (Change Manager) synchronized with the parent goal controller.

Resources `Resource` captures observable/controllable quantities (`resourceId`, `name`, `type` $\in$\{int,bool\}). Its `supported_props` (`initialValue`, and for `int`: `lowerBound`, `upperBound`) determine the variable domain used in PRISM. Resources are *declared through their association with tasks* (i.e., introduced where they are actually consumed/updated), while goals and tasks may reference them in `context` guards. This models the feedback loop: guards read resource state; task execution updates it (D3, D4).

Links (AND/OR decompositions) `Link` (`type` $\in$\{AND, OR\}, `parentId`, `childId`) encodes the refinement graph and is the only relationship used to form the goal/task hierarchy. A `Goal` has *one-to-many* outgoing `Link`s to its children (each child is either a `Goal` or a `Task`); each child has *exactly one* incoming `Link`. Disjunctive links define the candidate *variants* required by D1; conjunctive links define joint satisfaction conditions.

Runtime Annotation `Runtime Annotation` (`operator`, `operands`) is attached *exclusively* to a `Goal` (0..1 per goal). The `operator` selects the execution pattern (e.g., `AND(n1;n2)`, `AND(n1#n2)`, `OR(n1|n2)`, `OR(+)` for committed choice, `OR(n1@k->n2)` for degradation with retries), and `operands` enumerate the children to which the pattern applies. Operands must reference the goal's own children (as defined by `Link`s), ensuring locality of control and simplifying traceability in the translation.

Cardinalities and design rationale (i) *Hierarchy*: a `Goal` has 1..n `Link`s to children; each child has exactly 1 parent `Link`. (ii) *Annotation scoping*: a `Goal` has 0..1 `Runtime Annotation` (one pattern per parent), avoiding conflicting schedules. (iii) *Resource use*: tasks introduce/modify resources; goals/tasks may read them via `context` guards. This design locates side effects at enactment while keeping reasoning layers declarative (MORPH alignment).

Traceability to PRISM Attributes under `custom_properties` map to PRISM variables, guards, and rewards (utility/cost). `Link` and `Runtime Annotation` determine the synchronization structure and enablement conditions in controller modules; `Resource` definitions determine variable domains and guard typing. The ERD therefore serves as the schema contract that guarantees a semantics-preserving, template-based translation from extended goal models (D1–D4) to verifiable PRISM specifications.

Figure 3.14 presents the EDGE Goal Model metamodel, which formalizes the abstract syntax and structural constraints of the modeling language used throughout this work. The metamodel defines four main entity types—`Goal`, `Task`, `Resource`, and `Link`—and their relationships within a hierarchical `GoalTree`. Both `Goal` and `Task` specialize the abstract class `Node`, inheriting identifiers and quantitative attributes such as `maxRetries`. Goals capture intentional reasoning elements, decomposed through `Link` relations of type `AND` or `OR`, and optionally enriched with runtime semantics represented by `ExecutionDetails`. Tasks represent operational actions that may depend on contextual `Resource` elements, which can be of integer or boolean type. The metamodel also introduces support for contextual and maintenance guards (`Condition`), inter-goal dependencies (`dependsOn`), and explicit retry semantics (`maxRetries`), ensuring that all constructs required by the EDGE desiderata (D1–D4) are captured in a consistent, machine-processable structure suitable for model-driven transformation.

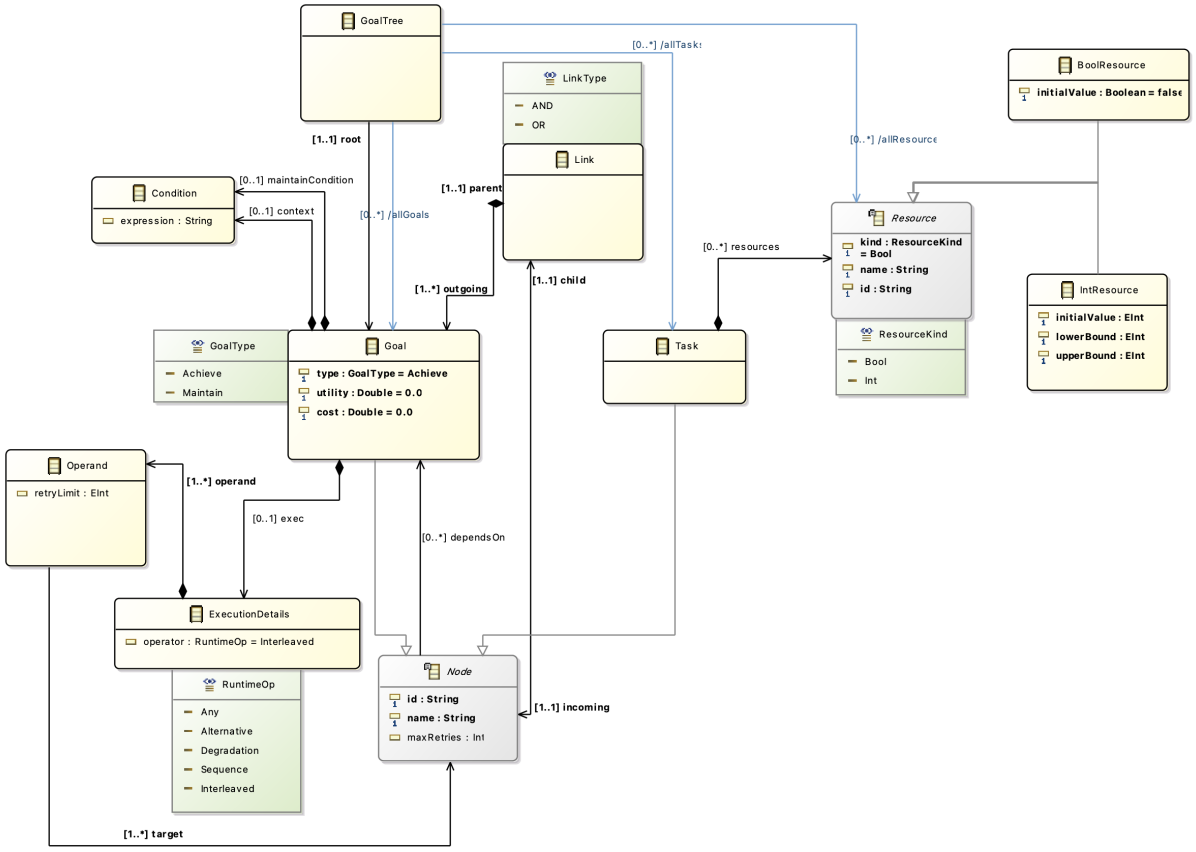


Figure 3.14: EDGE EMF conceptual model

3.4 Phase 2: Construction of the Intermediate Representation (IR)

In the second phase of the methodology, the goal model semantics are captured into an *Intermediate Representation* (IR). The IR provides a language-agnostic abstraction that preserves the semantic extensions identified in section 3.3, serving as the bridge between the conceptual goal model and its formal realization as a PRISM specification. Through this abstraction, the transformation process becomes independent of syntactic constraints, allowing the synthesis stage to operate over a normalized and semantically faithful data structure.

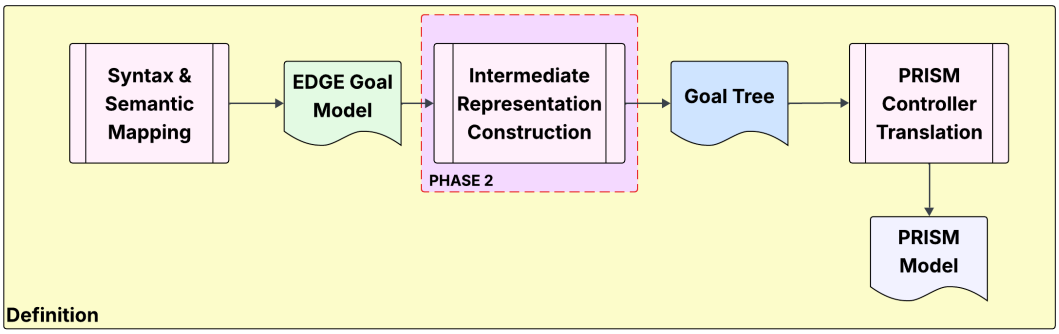


Figure 3.15: Phase 2: construction of a normalized Intermediate Representation (IR).

The IR functions as the *runtime input model* traversed by the template engine to generate target code [13]. It formalizes the high-level semantics of the goal model in a structured and analyzable form, isolating domain semantics from output syntax and enabling reproducible and target-independent generation.

3.4.1 The GoalTree IR.

Figure 3.16 presents the class diagram defining the `GoalTree` object, which underpins the IR structure. It specifies the entities, relationships, and constraints that encode the semantics of goals, tasks, resources, contextual guards, and runtime annotations. Each `GoalNode` represents either a composite goal or a terminal task, linked recursively to its children and optionally associated with `ExecutionDetails` and `Context` elements. All nodes share the same set of `NodeProperties`—such as `utility`, `cost`, `dependsOn`, and `maxRetries`—ensuring uniform handling of quantitative and dependency information.

The root `GoalTree` aggregates multiple `GoalNode` instances, forming a recursive hierarchy where each goal can contain either subgoals or task nodes. Tasks may reference one or more `Resource` elements, each defined by a type (`boolean` or `int`) and their operational bounds (`initialValue`, `lowerBound`, `upperBound`). Cardinalities express these structural constraints: a goal can have one or more children, tasks may depend on zero or more resources, and each `Context` holds exactly one `assertion` and optionally one `maintain` condition. These relationships ensure model validity and preserve the semantics of refinement and contextual dependency required for later synthesis.

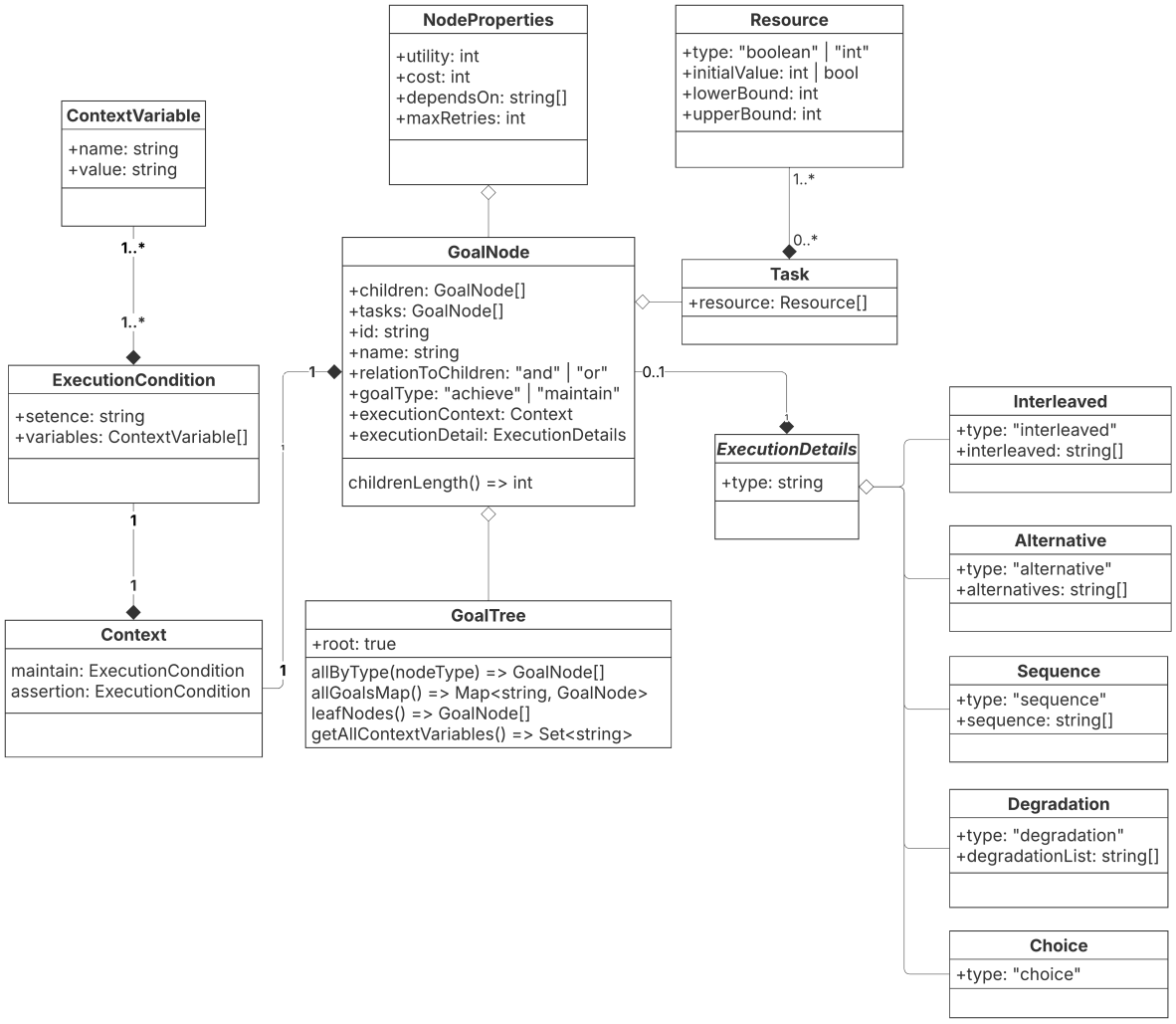


Figure 3.16: Class diagram of the `GoalTree` Intermediate Representation (IR).

The **tree-based design** of the IR mirrors the goal decomposition hierarchy central to Goal-Oriented Requirements Engineering (GORE) and the EDGE framework. Each node's recursive definition supports the refinement and propagation of semantics—from high-level goals down to executable tasks—while enabling deterministic traversal by the transformation engine. The separation of `ExecutionDetails` into a dedicated structure encapsulates the semantics of runtime operators (e.g., sequence, interleaving, degradation), preventing them from being hard-coded into goal logic and allowing reusable encoding patterns across different goal types. Likewise, the `Context` structure isolates activation and maintenance conditions, enabling their validation and reuse across nodes while ensuring consistent interpretation during PRISM generation. This modular architecture enhances clarity, supports incremental verification, and aligns with TBCG's emphasis on separating domain semantics from generative logic.

As a result, the IR becomes the unifying semantic layer of the translator pipeline,

providing the structural and behavioral foundation necessary to address **Desideratum D5 (Support for automating goal selection)** [9]. D5 requires a principled mechanism for mapping extended goal models into formal representations suitable for controller synthesis. The `GoalTree` fulfills this role by encoding goal hierarchies, interdependencies, and quantitative attributes in a machine-readable format, enabling systematic, template-driven translation into executable PRISM modules. By maintaining a coherent, language-agnostic structure, the IR establishes the conceptual basis for reasoning automation, ensuring that the transformation pipeline remains systematic, adaptable, and semantically consistent across subsequent phases.

3.5 Phase 3: Rules for the PRISM controller translation

The final phase of the translation methodology defines the transformation rules that govern the translation engine. These rules formalize how each element of the Intermediate Representation (IR) is rendered into its corresponding PRISM construct, ensuring that the generated model preserves the semantics established in Phase 1 (section 3.3) and structured in Phase 2 (section 3.4). The translation engine traverses the `GoalTree` hierarchy and instantiates parameterized code fragments for goals, tasks, resources and contextual variables, respecting any runtime constraint that should be enforced during the process, producing a verifiable goal controller in PRISM specification.

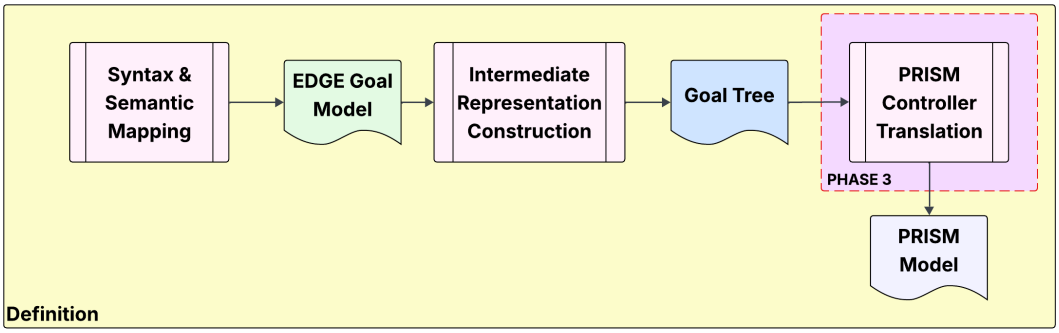


Figure 3.17: Phase 3: Generating the goal controller PRISM model

The resulting output model consists of three main components: (i) the `ChangeManager` module, which encodes task execution behavior and post-execution updates; (ii) the `System` module, which defines resources and contextual variables; and (iii) a set of n goal modules $(G_1, \ldots, G_n)$, each representing a goal node and its associated decision

logic. Together, these modules form the synthesized controller that operationalizes the goal hierarchy into an analyzable probabilistic model.

Listing 3.9 presents the base template for the PRISM output. Tokens delimited by % (e.g., `G%goal.id%`) denote substitution parameters dynamically resolved during code generation—where each placeholder is replaced by the corresponding model value at translation runtime.

```
dtmc

<for each goal in goals>
module G%goal.id%
  %variables%

  %pursue_statements%
  %achieve_statement%
  %skip_statement%
endmodule
%goal_formulas%
</for>

<for each task in tasks>
%task_achievability_constants%
</for>

module ChangeManager
  <for each task in tasks>
    %variables%
    %pursue_statement%
    %achieve_statement%
    %try_statement%
    %failed_statement%
  </for>
endmodule

module System
  %context_variables%
  %resource_variables%
endmodule
```

Listing 3.9: Base template for PRISM output model

In the following subsections, we define the construction rules for each of these components. Each subsection presents the templates used for code generation, the corresponding generation algorithm, and a discussion of how the resulting PRISM modules contribute to the executable model's structure and behavior.

3.5.1 ChangeManager Module Construction Rules

The construction of the `ChangeManager` module constitutes the first transformation stage responsible for encoding the task-level execution logic within the PRISM model. The template engine iterates over all task nodes derived from the `GoalTree` structure and, for each task, generates a standard set of five PRISM statements: two variable declarations, followed by one `pursue`, one `achieved`, and one `failed` transition rule. These elements collectively define the control flow that governs task activation, completion, and error handling at runtime, as illustrated in Listing 3.7.

3.5.1.1 Transition Templates

The templates for the `pursue` and `achieved` transitions are currently modeled as placeholders. Their purpose is to serve as integration points for future runtime decision variables that will enrich the DTMC's control dynamics. The corresponding template structures are shown in Listings 3.10 and 3.11, respectively.

```
[pursue_%task.id%] true -> true;
```

Listing 3.10: Template for pursueTask transition in PRISM syntax

```
[achieved_%task.id%] true -> true;
```

Listing 3.11: Template for achievedTask transition in PRISM syntax

The `failed` transition plays a more significant role in runtime adaptation. It monitors task execution outcomes and resets the task state when a failure is detected. If the failed task belongs to a degradation goal with retry semantics, a bounded failure counter is incremented, allowing re-attempts up to the specified limit. Listing 3.12 presents the parameterized template used for generating this rule.

```
[failed_%task.id%] %task.id%_pursued=1 & %task.id%_achieved=0 -> (%task.id%
    ↪ _pursued'=0);
```

Listing 3.12: Template for failedTask transition in PRISM syntax

The `try` transition complements the failure and achievement mechanisms by encoding the probabilistic outcome of a task attempt. As shown in Listing 3.13, this template

models a single execution trial in which the task is currently being pursued and has not yet succeeded. The transition branches according to the task's achievability value: with probability `Tn_achievable`, the task reaches an achieved state, and with probability `1-Tn_achievable`, the task aborts the attempt and returns to a non-pursued state. This structure mirrors the earlier semantics described for goal achievability formulas, ensuring that each probabilistic trial aligns with the task's declarative success model. The resulting transition provides a principled linkage between the abstract achievability specification and the probabilistc behavior of individual execution attempts.

```
[achieved_%task.id%] %task.id%_pursued=1 & %task.id%_achieved=0 -> %task.id%
    ↪ _achievable: (%task.id%_achieved'=1) + 1-%task.id%_achievable: (%task.id%
    ↪ _pursued'=0);
```

Listing 3.13: Template for try transition in PRISM syntax

3.5.1.2 Generation Algorithm

Algorithm 1 summarizes the generation logic implemented by the template engine. The `changeManagerModule` procedure receives the `GoalTree` as input, extracts all nodes of type `task`, and iteratively constructs the body of the PRISM module. For each task, the algorithm first generates the variable declarations via the `taskVariables` procedure, defining the state variables that represent the task's pursuit, achievement, and failure statuses. It then produces the corresponding transition rules through `taskTransitions`, which combines the templates from Listings 3.10–3.12 into the module structure.

Algorithm 1 `changeManagerModule` generation procedure

```
1:  procedure changeManagerModule(goalTree)
2:      tasks ← goalTree.allByType('task')
3:      moduleBody ← "module ChangeManager"
4:      for all task in tasks do
5:          varDecl ← taskVariables(task)
6:          moduleBody ← moduleBody ∪ varDecl
7:      end for
8:      for all task in tasks do
9:          transitions ← taskTransitions(task)
10:         moduleBody ← moduleBody ∪ transitions
11:     end for
12:     moduleBody ← moduleBody ∪ "endmodule"
13:     return moduleBody
```

end procedure
procedure TASKVARIABLES(task)
    pursuedVar ← `\pursuedVariable(task.id)`
    achievedVar ← `\achievedVariable(task.id)`
    return pursuedVar ∪ achievedVar ∪ failedVar
end procedure
procedure TASKTRANSITIONS(task)
    pursueAction ← `\pursueTask(task)`
    achieveAction ← `\achieveTask(task)`
    tryAction ← `\tryTask(task)`
    failAction ← `\failedTask(task)`
    return pursueAction ∪ achieveAction ∪ failAction ∪ tryAction
end procedure

This algorithm guarantees that all task behaviors are represented consistently in the synthesized `ChangeManager` module. Each generated section corresponds directly to one of the template definitions, ensuring both modularity and traceability within the overall transformation pipeline.

3.5.2 System Module Construction Rules

The `System` module constitutes the environmental layer of the generated PRISM specification. It consolidates the declarations of all `Resource` elements and user-defined `context` variables that provide the state information used by the controller during decision making. While the `ChangeManager` module encodes task behavior, the `System` module defines the static and dynamic variables that constrain those behaviors at runtime, ensuring that adaptation decisions remain consistent with the model's environmental assumptions.

3.5.2.1 Variable Templates

The `System` module is composed exclusively of variable declarations, each derived either from a resource specification in the `GoalTree` or from external context variables defined in the CLI-generated `variables.json` file. Two distinct template formats are used for resource definitions, depending on their type. Boolean resources are declared as simple flags, whereas integer resources include explicit range bounds and initial values. The templates for both cases are presented below, followed by the template for context variables.

```
1 %resource.id%: bool init %value%;
```

Listing 3.14: Template for Boolean resource declaration

```
%resource.id%: int [%lower%..%upper%] init %value%;
```

Listing 3.15: Template for Integer resource declaration

```
%var%: bool init %initVal%;
```

Listing 3.16: Template for Context variable declaration

Boolean and integer resource templates capture the measurable properties of the system or environment (e.g., `batteryLevel`, `isConnected`), while context variables represent logical conditions that gate goal activation or task execution. The uniform syntax allows the template engine to render all environmental data sources as standard PRISM variables, enabling the model checker to reason over them during verification.

3.5.2.2 Generation Algorithm

Algorithm 2 outlines the procedure used to construct the `System` module body. The process begins by collecting all resource nodes and contextual variables from the input goal model and auxiliary files. Each resource is then instantiated using one of the two templates defined above—depending on whether its type is `boolean` or `int`. In parallel, the algorithm parses the `variables.json` file to extract context variables and their initial values, generating one declaration per variable using the context template. Finally, the module is assembled by concatenating all resource and context declarations within the `System` block, forming the final PRISM syntax fragment.

Algorithm 2 `systemModule` generation procedure

```
1:  procedure SYSTEMMODULE(goalTree, fileName)
2:      resources ← GOALTREE.ALLBYTYPE('resource')
3:      variables ← GOALTREE.CONTEXTVARIABLES
4:      moduleBody ← "module System"
5:      for all resource in resources do
6:          decl ← DECLARERESOURCE(resource)
7:          moduleBody ← moduleBody ∪ decl
8:      end for
9:      valuesFile ← GETVARIABLESFILEPATH(fileName)
10:     defaultValues ← READFILE(valuesFile)
11:     for all var in variables do
12:         decl ← DECLARECONTEXTVAR(var, defaultValues)
13:         moduleBody ← moduleBody ∪ decl
```

```
    end for
    moduleBody ← moduleBody ∪ "endmodule"
    return moduleBody
end procedure
procedure DECLARERESOURCE(resource)
    if resource.variable.type == "boolean" then
        return DECLAREBOOLRESOURCE(resource)
    else if resource.variable.type == "int" then
        return DECLAREINTRESOURCE(resource)
    end if
end procedure
procedure DECLAREBOOLRESOURCE(resource)
    return \boolResource(resource.id, resource.value)
end procedure
procedure DECLAREINTRESOURCE(resource)
    return \intResource(resource.id, resource.lower, resource.upper,
resource.value)
end procedure
procedure DECLARECONTEXTVAR(var, defaultValues)
    initVal ← defaultValues[var] or "MISSING_VARIABLE_DEFINITION"
    return \contextVar(var, initVal)
end procedure
```

This procedure guarantees that the generated `System` module accurately reflects the environment defined by the model and its supporting configuration.

3.5.3 Goal Modules Construction Rules

The set of goal modules realizes the *Goal Management Layer* of the controller. Each goal module encapsulates the decision logic that (i) activates itself when its dependencies and context hold, (ii) propagates pursuit to its children (subgoals or tasks) according to the runtime annotation, and (iii) reports completion upward once the achievement condition is met. Listing 3.17 presents the template used to generate a PRISM goal module.

```
module %goal.id%

  %goal.id%_pursued  : [0..1] init 0;
  <if !(%goal.type% == "maintain")>
    %goal.id%_achieved : [0..1] init 0;
```

```
  </if>
  %retry_counters%

  %pursue_statements%

  %achieve_statement%

  %skip_statement%

endmodule
<if %goal.type% == "maintain">
  %maintain_condition_formula%
</if>
%achievability_formula%
```

Listing 3.17: Template for generating a PRISM goal module

In what follows, we describe each part of the template and how it is instantiated by the template engine.

3.5.3.1 Variable templates

Every goal module declares two status variables, `%goal.id%_pursued` and `%goal.id%_achieved`, which track activation and completion, respectively (see Section 3.3.1.4). For degradation goals, additional bounded counters are emitted—one per retry-capable child—to support fallback semantics. Listing 3.18 shows the substitution pattern for `%retry_counters%`.

```
<for each child in goal.variantsWithRetry>
%child.id%_failed : [0..%child.maxRetry%] init 0;
</for>
```

Listing 3.18: Template for generating retry counters for degradation variants

3.5.3.2 Achieve statement

Listing 3.19 shows the template for the `achieve` transition. The token `achieveCondition` compiles to a PRISM boolean formula that reflects the goal's decomposition: conjunction (`&`) for `AND`, disjunction (`|`) for `OR`.

```
[achieved_%goal.id%] %goal.id%_pursued=1 & %achieve_condition% -> (%goal.id%
    ↪ _pursued'=0) & (%goal.id%_achieved'=1);
```

Listing 3.19: Template for `achieveStatement` in PRISM syntax

3.5.3.3 Skip statement

The `skip` transition resets a goal that was selected for pursuit but failed to activate any of its children. As illustrated in Listing 3.20, this transition allows the controller to return the goal to an idle state, enabling re-evaluation in subsequent decision cycles. Its guard is synthesized by the helper function `childrenHasNotBeenPursued`, which emits a conjunction asserting that all child `pursued` variables remain false, thereby preventing premature resets or inconsistent synchronization.

```
[skip_%goal.id%] %goal.id%_pursued=1 & %children_not_pursued_condition%  -> (%goal.id%_pursued'=0);
```

Listing 3.20: Template for `skipStatement` in PRISM syntax

3.5.3.4 Pursue statements

The `pursue` transitions encode both the runtime annotations and the decomposition structure of each goal. Algorithm 3 summarizes the generation procedure. The iteration domain comprises the goal itself (for self-activation) and its corresponding variants or tasks. The flag `isItself` distinguishes between two cases: (a) activation of the goal node itself, which updates its state variable (`%goal.id%_pursued'=1`); and (b) propagation to its subgoals or tasks, which emits `true` on the update side while constructing the appropriate guard expressions. When generating a pursue line for the goal itself, the guard must explicitly incorporate both its dependency array and context activation condition.

Semantic variation arises from the goal's structural type: `OR` goals instantiate one of the runtime semantics–`any`, `alternative`, or `degradation`– whereas `AND` goals represent `sequence` or `interleaved` coordination patterns. In the case of `maintain` goals, the `maintainCondition` is added as an additional guard to enforce continuous satisfaction of the maintenance constraint throughout execution.

In Algorithm 3, the functions `pursueAnyGoal`, `pursueAlternativeGoal`, `pursueDegradationGoal`, `pursueSequentialGoal`, and `pursueInterleavedGoal` are template procedures responsible for constructing the `pursue` transition corresponding to each runtime variant. Each function receives the current goal and the child node under iteration as input and emits the appropriate PRISM statement according to the semantics defined for that variant. Their templates follow the structures presented in section 3.3.3, and their detailed generation algorithms are provided in chapter A.

Algorithm 3 `pursueStatements(goal)`: generate PRISM pursue transitions for a goal and its children

```
1: procedure pursueStatements(goal)
2:     goalsToPursue ← [goal] ∪ goal.children ∪ goal.tasks
3:     pursueLines ← [ ]
4:     for all child in goalsToPursue do
5:         isItself ← (child.id == goal.id)
6:         left ← [pursue_child.id] ∧ pursued(goal.id)=(0 if isItself else 1)
7:         if isItself then
8:             left ← left ∧ achieved(goal.id)=0 ∧ goalDependencyStatement(goal)
   ∧ goal.contextAssertion.sentence
9:             right ← (pursued(goal.id)'=1)
10:        else
11:            right ← true
12:            if goal.relationToChildren == 'or' then
13:                type ← goal.executionDetail.type
14:                if type == 'any' then
15:                    left ← left ∧ pursueAnyGoal(goal, child.id)
16:                else if type == 'alternative' then
17:                    left ← left ∧ pursueAlternativeGoal(goal, child.id)
18:                else if type == 'degradation' then
19:                    left ← left ∧ pursueDegradationGoal(goal, child.id)
20:                end if
21:            else if goal.relationToChildren == 'and' then
22:                if goal.executionDetail.type == 'sequence' then
23:                    left ← left ∧ pursueSequentialGoal(goal, child.id)
24:                else if goal.executionDetail.type == 'interleaved' then
25:                    left ← left ∧ pursueInterleavedGoal(goal, child.id)
26:                end if
27:            end if
28:        end if
29:        if child.type == 'maintain' then
30:            if isItself then
31:                left ← left ∧ child.maintainCondition.sentence
32:            else
33:                left ← left ∧ achievedMaintain(child.id)
```

```
34:             end if
35:         end if
36:         APPEND(pursueLines, left -> right;)
37:     end for
38:     return pursueLines
39: end procedure
40: procedure GOALDEPENDENCYSTATEMENT(goal)
41:     if not goal.dependsOn  then
42:         return ""
43:     end if
44:     condition ← ""
45:     for all dep in goal.dependsOn do
46:         condition ← condition ∪ achieved(dep) = 1
47:         if dep is not last in goal.dependsOn then
48:             condition ← condition ∪ " & "
49:         end if
50:     end for
51:     return " & (" + condition + ")"
52: end procedure
```

Maintain Goals Formula. For `Maintain` goals, an additional formula is declared outside the goal module to encode the condition that must remain true throughout system execution. This formula expresses the invariant that defines the goal's sustained satisfaction state. Its template is presented in Listing 3.21, where the token `%maintainCondition%` is replaced by the contextual guard defined in the goal model. During code generation, this formula is referenced when constructing the parent goal's module—specifically, it is used as a guard to determine whether the maintain goal should remain active, as illustrated in line 33 of Algorithm 3.

```
1 formula %goal.id%_achieved_maintain = %maintainCondition%;
```

Listing 3.21: Template for maintain goal formula in PRISM syntax

Achievability formula The achievability formula, whose template is shown in Listing 3.22, encodes the probability that a goal can be successfully satisfied based on the achievability of its children. Its construction follows the algorithmic process formalized in Algorithm 4. During code generation, this procedure inspects the goal's refinement structure and derives the corresponding expression. If the goal has a single child, the algorithm

forwards that child's achievability value directly, reflecting an uninterrupted dependency. For **AND**-refinements, it computes the product of all child achievability variables, capturing the requirement that every subgoal must succeed. For **OR**-refinements, it synthesizes the customary inclusion–exclusion expression—the sum of children's achievability values minus their product—representing the probability that at least one alternative succeeds. By systematically applying these rules, the algorithm preserves and propagates the quantitative semantics of the goal hierarchy throughout the generated PRISM model.

```
formula %goal.id%_achievable = %achievability_formula%;
```

Listing 3.22: Template for goal achievability formula in PRISM syntax

3.5.3.5 Generation Algorithm

Algorithm 4 `achievableGoalFormula(goal)`: generate achievability formula for a goal

```
procedure ACHIEVABLEGOALFORMULA(goal)
    children ← CHILDRENINCLUDINGTASKS(goal)
    if |children| = 1 then
        child ← children[0]
        expr ← ACHIEVABLEFORMULAVARIABLE(child.id)
    else
        vars ← [ACHIEVABLEFORMULAVARIABLE(c.id) for all c in children]
        productPart ← JOIN(vars, " * ")
        if goal.relationToChildren = 'and' then
            expr ← productPart
        else if goal.relationToChildren = 'or' then
            sumPart ← JOIN(vars, " + ")
            expr ← sumPart − PARENTHESIS(productPart)
        end if
    end if
    return "formula " + goal.id + "_achievable = " + expr + ";"
end procedure
```

3.5.3.6 Generation Algorithm

Algorithm 5 summarizes the generation logic used to produce the PRISM module corresponding to an individual goal. The `goalModule` procedure receives a single `goal` as input and constructs the module in two stages. First, it assembles the structural elements of the module by invoking `variablesDefinition` to declare the state vari-

ables associated with the goal, followed by the generation of transition rules through `pursueStatements`, `achieveStatement`, and `skipStatement`. These transitions encode the operational semantics that govern the goal's activation, completion, and suspension. Second, the procedure generates the semantic formulas placed outside the module: `maintainConditionFormula` (when applicable) and `achievableGoalFormula`. These formulas specify the contextual satisfaction conditions for maintain goals and the quantitative achievability semantics for all goals. Together, the module body and its associated formulas define a self-contained PRISM representation that integrates both the behavioral and declarative aspects of the goal.

Algorithm 5 `goalModule(goal)` generation procedure

```
1: procedure goalModule(goal)
2:     maintainFormula ← maintainConditionFormula(goal)
3:     achievableFormula ← achievableGoalFormula(goal)
4:     formulaStatements ← joinNonEmpty([maintainFormula, achievableFormula], "\n")
5:     vars ← variablesDefinition(goal)
6:     pursueLines ← pursueStatements(goal)
7:     achieveLine ← achieveStatement(goal)
8:     skipLine ← skipStatement(goal)
9:     moduleBody ← "module " ∪ goal.id
10:    moduleBody ← moduleBody ∪ vars
11:    moduleBody ← moduleBody ∪ pursueLines
12:    moduleBody ← moduleBody ∪ achieveLine
13:    moduleBody ← moduleBody ∪ skipLine
14:    moduleBody ← moduleBody ∪ "endmodule"
15:    return moduleBody ∪ formulaStatements
16: end procedure
```

Chapter 4

The Implementation Architecture

Following the translation methodology introduced in Chapter 3, the behavioral specification of an EDGE goal model can be transformed into a PRISM goal controller. Because this process is labor-intensive and error-prone when performed manually, we implement an automated toolchain, `EDGE2PRISM`, that instantiates the phases defined in Chapter 3 while hiding the low-level formalization details from the modeler.

To mitigate this burden and ensure a systematic and semantically consistent transformation, an automated toolchain named `EDGE2PRISM` has been developed. Implemented in **TypeScript** and leveraging `ANTLR` to define the parsing grammars, the tool operationalizes the translation methodology by automatically parsing, interpreting, and generating PRISM specifications directly from EDGE goal models.

The `EDGE2PRISM` translator abstracts the low-level formalization details of probabilistic modeling from system designers, enabling them to focus on modeling adaptive behaviors while ensuring that the generated PRISM models remain analyzable, traceable, and faithful to the original goal semantics. By automating this translation, the tool facilitates the practical adoption of the EDGE framework, allowing designers to concentrate on specifying system behavior and adaptation strategies rather than on the technical intricacies of probabilistic modeling.

4.1 piStar: a GORE tool for EDGE models

The `piStar` tool [24] is a web-based goal-oriented modeling environment that supports the creation and extension of i^* 2.0 models. Designed to lower the entry barrier for researchers and practitioners working with goal modeling, `piStar` provides a visual interface and an extensible JavaScript-based API that facilitate experimentation with modeling extensions such as context annotations, behavioral expressions, and quantitative properties. Its lightweight architecture allows models to be exported as JSON files, which can be read-

ily processed by external transformation pipelines. This interoperability feature makes `piStar` particularly suitable for integrating with model-driven engineering workflows like the one proposed in this work.

Within the EDGE framework, `piStar` serves as the graphical modeling front-end used to construct goal models that conform to the methodological requirements defined in Chapter 3. Among its supported elements are the primitives required for EDGE-compliant modeling—goals, tasks, refinements (AND/OR links), and resources—as well as a flexible `customProperties` field that allows the inclusion of additional attributes such as `context`, `maintain`, `dependsOn`, and quantitative properties like `utility` and `cost`. These features make `piStar` an effective platform for encoding the semantic extensions described by the EDGE metamodel (Figure 3.14), ensuring that the behavioral and structural constructs required for translation into PRISM are represented explicitly within the model.

All EDGE goal models used throughout this work were developed in `piStar`. The models presented earlier—namely, the Tele Assistance System (TAS) in Figure 5.1 and the Lab Sample Logistics exemplar in Figure 5.3—were designed in this environment following the structural constraints of the EDGE metamodel. These models include runtime annotations defining goal sequencing, interleaving, and degradation patterns, as well as contextual and maintenance conditions that govern their activation and persistence.

Once authored in `piStar`, the resulting JSON goal model serves as the primary input for the translation pipeline implemented by the `EDGE2PRISM` toolchain. The translator parses and validates the model against the construction rules established in Chapter 3, generating an intermediate `GoalTree` representation that preserves the semantics of all contextual guards, runtime annotations, and inter-goal dependencies.

4.2 Toolchain Architecture

The `EDGE2PRISM` package encapsulates the entire translation pipeline within a unified command-line interface (CLI) capable of ingesting EDGE goal models in `piStar` format and producing corresponding PRISM specifications for their synthesized goal controllers. The tool is implemented in **TypeScript** and has as external dependency solely the `ANTLR` tool and library, used to create parsers for the goal models according to the grammar rules defined in Section 3.5.

Figure 4.1 depicts the architectural organization of the `EDGE2PRISM` toolchain, structured into the same separation of concerns defined in Section 3.2 —parsing, representation, and generation. The `GoalTree` package loads the `piStar` model and builds the IR, while the `templateEngine` applies the translation rules to emit PRISM modules. The figure below presents the concrete organization of these implementation packages.

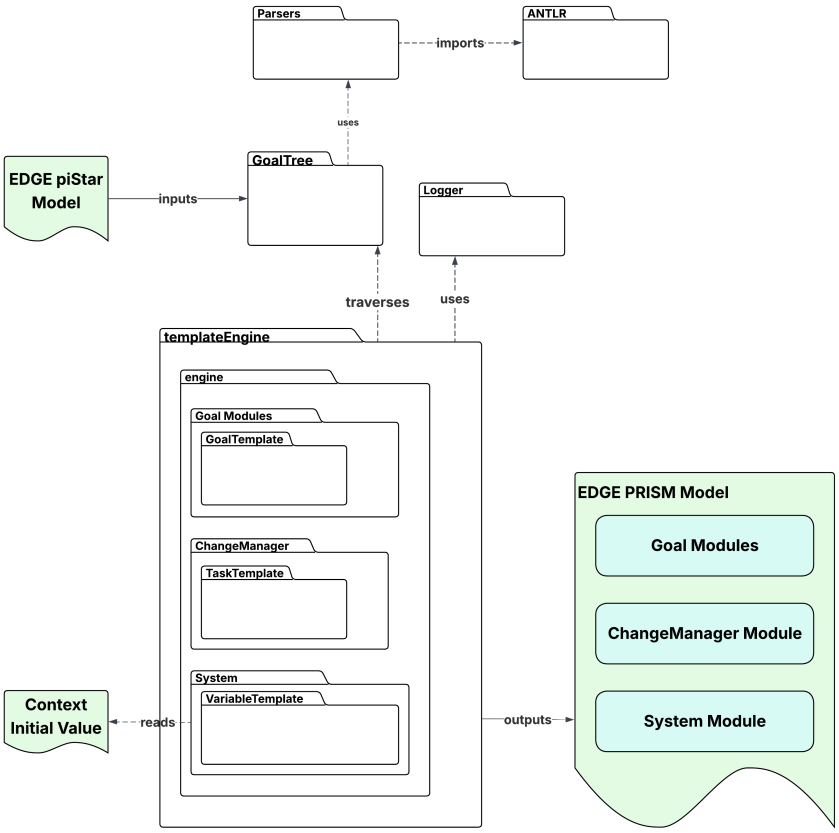


Figure 4.1: UML Package Diagram representing the architecture of the translator toolchain

4.3 Implementation

The implementation of `EDGE2PRISM` mirrors the methodological phases defined in Chapter 3. Accordingly, the codebase is organized into modular components that handle parsing, intermediate representation, and PRISM generation. The main packages are:

Package **/antlr** — contains the generated TypeScript classes derived from the ANTLR grammars, serving as the foundation for the **parsers** module;

Package **/GoalTree** — responsible for loading the `piStar` model and constructing the intermediate representation (IR) used by the **templateEngine**;

Package **/logger** — implements logging utilities that ensure traceability and facilitate debugging throughout the translation process;

Package **/parsers** — provides helper functions for parsing Boolean expressions (context and maintain conditions) and extracting goal details (e.g., identifier, name, and execution type) using the grammars defined in Section 3.5;

Package **/templateEngine** — traverses the **GoalTree** structure and applies the corresponding emission rules to generate the PRISM model for the goal controller. The template engine requires an auxiliary file `scenarioInitialConditions.json` that defines the initial value for each of the context variables present in the model.

The execution pipeline follows the three methodological phases introduced in Chapter 3. Below we detail how each phase is realized in the tool.

Stage 1 **Syntax & Semantic Mapping** — performed in `piStar` [24], a goal modeling tool that provides the constructs and attributes required to build an EDGE-compliant goal model;

Stage 2 **Intermediate Representation Construction** — the generation of a structured `GoalTree` derived from the `piStar` model, serving as the semantic bridge between conceptual and executable artifacts; and

Stage 3 **PRISM controller translation** — the code generation phase inside the `templateEngine`, which transforms the intermediate representation into a synthesized PRISM specification for the goal controller, following the emission rules defined in Section 3.5.

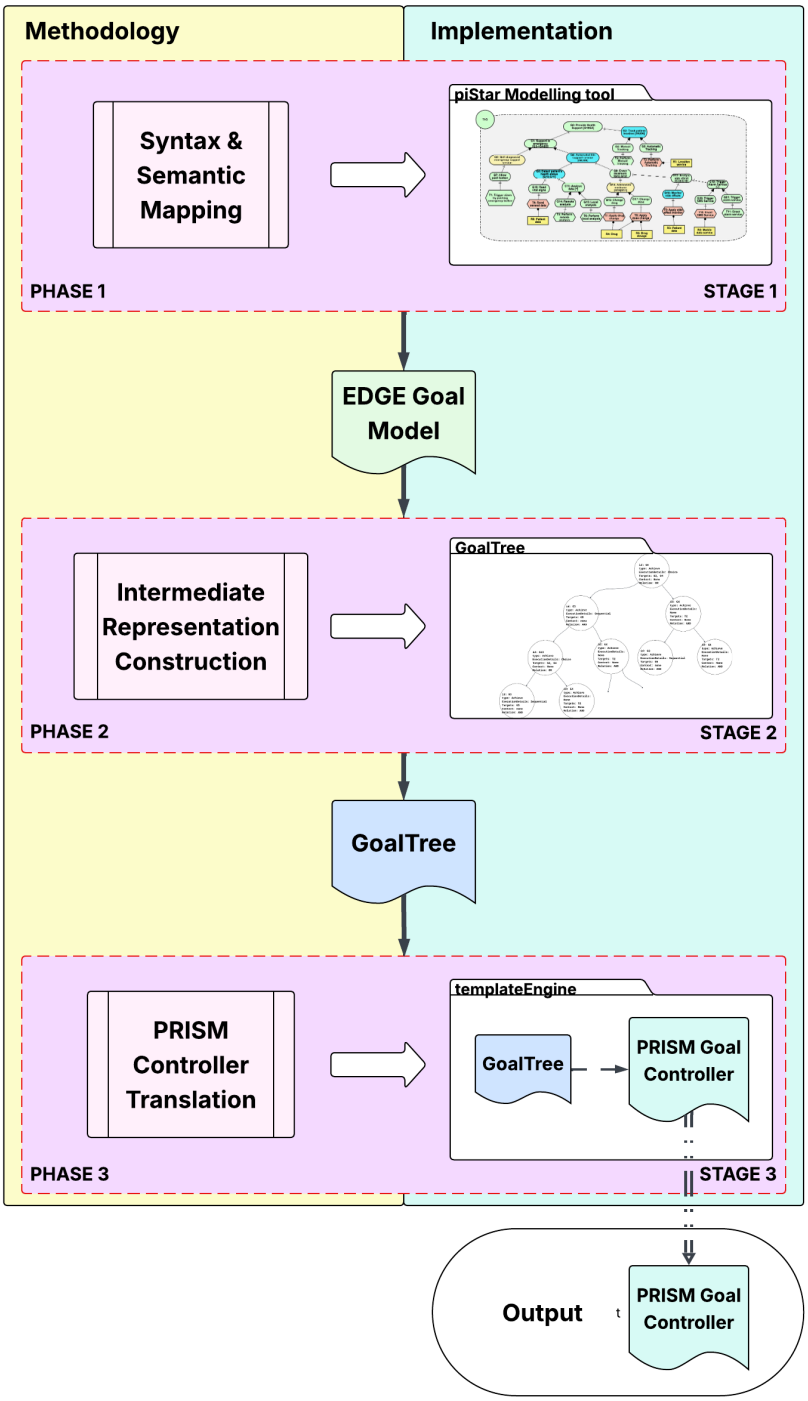


Figure 4.2: Mapping of methodological phases to implementation stages in the `EDGE2PRISM` tool.

4.3.1 Stage 1: Syntax & Semantic Mapping

The first stage of the translation process consists of preparing the input goal models for automated transformation. This step is carried out using the `piStar` modeling tool,

which provides the constructs required to represent EDGE-compliant goal models. in Chapter 5 we depict how we've constructed the EDGE goal models for the TAS and LSL systems under the model language, incorporating all necessary elements such as goals, tasks, resources, contextual guards, and runtime annotations.

Once the model is complete and exported, it serves as the input to the `EDGE2PRISM` tool, which begins the translation by validating the model structure and semantics (first step of **Stage 2**). If any malformation or inconsistency is detected, the process halts and produces detailed warnings and errors, allowing the designer to trace, debug, and correct the model before proceeding to the subsequent translation stages.

4.3.2 Stage 2: Intermediate Representation Construction

The second stage of the translation pipeline receives the `piStar` model, loaded from its JSON representation, and converts it into the Intermediate Representation (`GoalTree`) that captures the structural hierarchy and semantics of the goal model. During this transformation, the `/GoalTree` package validates the model against the syntactic and semantic constraints defined by the EDGE metamodel. The process enforces several structural requirements—for instance, it raises exceptions if more than one root goal is detected, if mandatory properties are missing for a node type, or if a goal lacks the necessary refinements—thereby ensuring that only well-formed models proceed to subsequent stages.

The construction of the `GoalTree` follows an iterative and recursive traversal of the input model, progressively instantiating each node and its relationships. The process begins with the `goalNameParser`, which extracts the goal's identifier, name, and execution details from its textual label. Subsequently, each node's `customProperties` are normalized and mapped onto the appropriate attributes of the internal representation, ensuring that all EDGE-specific semantics—such as context, maintain conditions, and runtime annotations—are correctly captured.

Once instantiated, each node is classified according to its type (`goal`, `task`, or `resource`) and validated against the structural constraints defined by the EDGE metamodel. This validation phase ensures that only tasks declare resources, that leaf goals are properly refined, and that resource nodes do not contain children. The procedure then recurses over the model's dependency graph, linking each node to its corresponding parent and establishing the correct AND/OR relations among subgoals.

As formalized in Algorithm 6, this process systematically traverses the model hierarchy, creating a well-formed and semantically consistent intermediate representation. The resulting `GoalTree` preserves both the structural organization and the behavioral semantics of the original goal model, serving as a normalized, machine-readable artifact ready for template-based code generation in the subsequent translation stage.

Algorithm 6 CREATETREE Procedure

```
1: procedure createTree(model)
2:     trees ← [ ]
3:     for all actor in model.actors do
4:         rootNode ← findRootNode(actor)
5:         tree ← createTreeFromRoot(rootNode, model)
6:         for all node in allNodes(tree) do
7:             node.parent ← findParentForNode(tree, node)
8:         end for
9:         append(trees, tree)
10:    end for
11:    return trees
12: end procedure
13: procedure createTreeFromRoot(node, model)
14:    (children, relation) ← nodeChildren(node, model)
15:    return createNode(node, relation, children)
16: end procedure
17: procedure nodeChildren(node, model)
18:    childLinks ← all links in model where link.target = node.id
19:    if childLinks is empty then
20:        return ([], null)
21:    end if
22:    relation ← assertAllRelationsAreTheSame(childLinks)
23:    children ← [ ]
24:    for all link in childLinks do
25:        childNode ← node with id = link.source
26:        (grandChildren, rel) ← nodeChildren(childNode, model)
27:        nodeObj ← createNode(childNode, rel, grandChildren)
28:        append(children, nodeObj)
29:    end for
30:    return (children, relation)
31: end procedure
32: procedure createNode(node, relation, children)
33:    (id, goalName, executionDetail) ← getGoalDetail(node.text)
34:    nodeType ← convertIStarType(node)
35:    if nodeType = 'resource' and children.length > 0 then
```

```
        throw WRONG_MODEL
    end if
    if nodeType = 'goal' and children.length = 0 then
        throw WRONG_MODEL
    end if
    if nodeType ≠ 'task' and children.resources.length > 0 then
        throw WRONG_MODEL
    end if
    properties ← normalizeProps(customProperties)
    return GOALNODE(id, goalName, executionDetail, relation, children, properties)
end procedure
```

4.3.3 Stage 3: PRISM Controller Generation

With the `GoalTree` constructed, the `/templateEngine` package performs the final stage of the translation pipeline: the automated synthesis of the PRISM model. This stage operationalizes the semantics encoded in the intermediate representation by traversing the hierarchical structure and applying the corresponding code templates. The generation process is divided into three sequential steps: (i) generation of the `Goal` modules, (ii) generation of the `ChangeManager` module, and (iii) generation of the `System` module. The algorithms underlying each of these procedures are detailed in Section 3.5—with specific implementations described for the goal modules in section 3.5.3, the ChangeManager in section 3.5.1.2, and the System module in section 3.5.2.2.

In the first step, the engine iterates through all goal nodes contained in the `GoalTree`, generating one PRISM module per goal. Each module includes its state variables (`pursued`, `achieved`), as well as the guarded commands corresponding to pursuit, achievement, and skip transitions. These elements are synthesized according to the goal's type (`Achieve` or `Maintain`), its decomposition relation (`AND` or `OR`), and any runtime annotations that determine sequencing, interleaving, or degradation behaviors. The resulting fragments are concatenated into a single output string that forms the body of the PRISM specification for the goal management layer.

Next, the `ChangeManager` module is generated by traversing all task nodes in the IR. Each task contributes a set of standard statements that model its operational behavior—variable declarations for pursuit and completion, a `pursue` transition, and `achieved` and an optional `failed` transitions to capture execution outcomes. This module serves as the behavioral substrate of the synthesized controller, implementing the enactment semantics that link high-level goals to executable actions.

Finally, the `System` module is produced by extracting all contextual and resource variables referenced across the model. These variables are then matched against an auxiliary configuration file (`scenarioInitialConditions.json`), which provides the initial values for each context parameter.

The template-based emission rules specified in Section 3.5 are then applied to produce the final PRISM specification. All emission events are logged through the `/logger` package to support debugging and inspection during implementation. The overall sequence of interactions—beginning with the user's translation request and culminating in the synthesis of the PRISM specification—is illustrated in Figure 4.3.

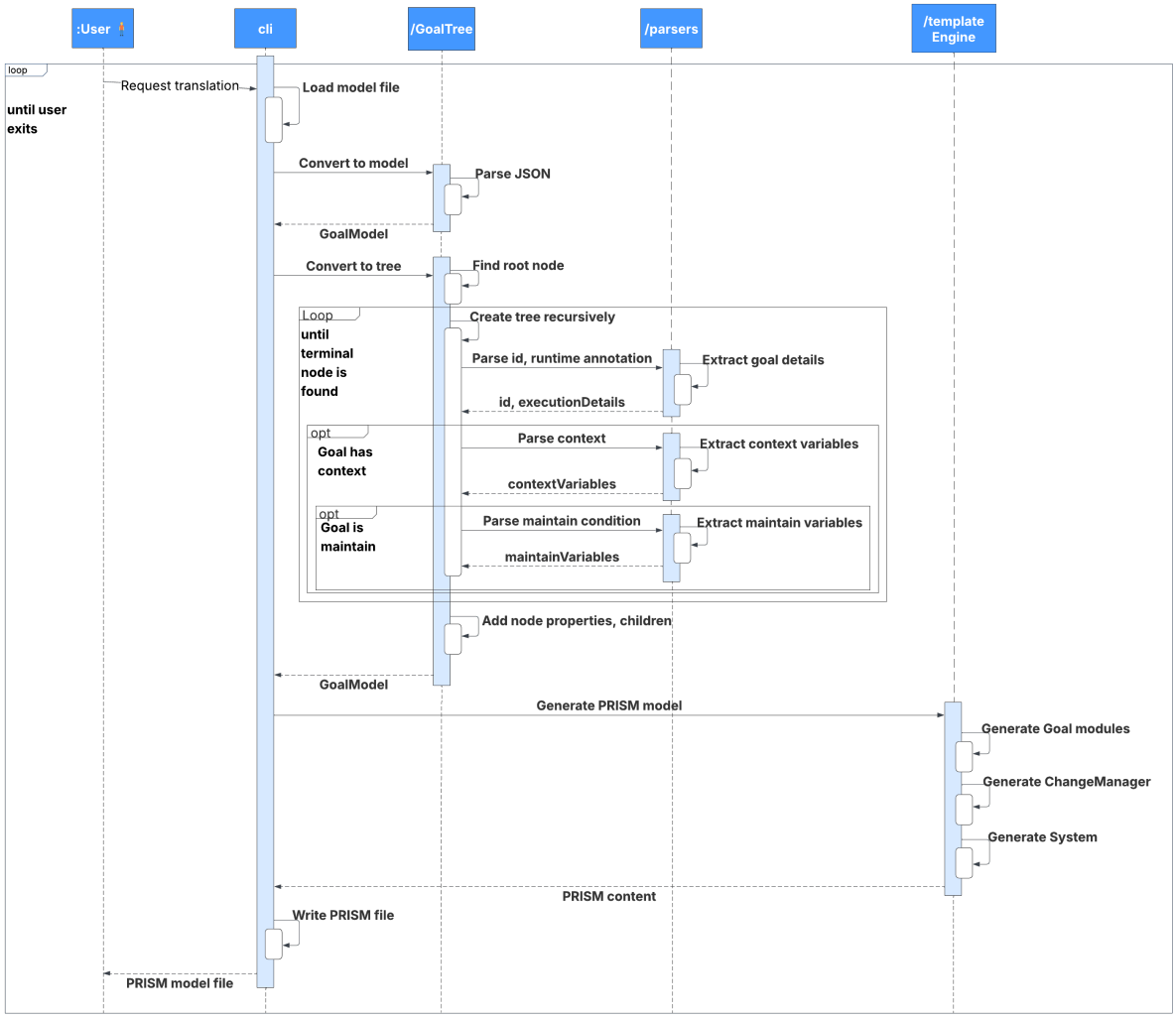


Figure 4.3: Sequence diagram for the translation process showing how packages communicate

Chapter 5

Modeling SAS Artifacts with EDGE

To assess the applicability and effectiveness of the proposed translation methodology, we examine two representative self-adaptive systems (SAS): the **TeleAssistance Service (TAS)** [14] and the **Lab Samples Logistics (LSL)** scenario from the RoboMax exemplars [15]. Both systems embody the challenges central to goal-based adaptation: context-driven activation, dependency management, and runtime verification under uncertainty –making them ideal candidates for evaluating semantics– preserving translation into PRISM. TAS exemplifies adaptive decision-making in healthcare monitoring, where environmental and physiological conditions determine goal activation and emergency escalation. The Lab Samples Logistics case, in turn, illustrates a cyber-physical workflow coordinating robotic agents and human personnel to ensure timely, secure transport of biological samples.

The **TeleAssistance Service (TAS)** is a widely adopted benchmark for self-adaptive software because it concentrates, in a compact healthcare workflow, the core challenges of goal-based adaptation: safety-critical decisions under uncertainty, multi-service coordination (sensing, analysis, treatment, and emergency response), and continuous operation under contextual constraints (privacy, connectivity, energy). Its mix of human-in-the-loop actions (panic button, manual tracking) and automated behaviors (continuous monitoring, treatment enactment) makes it suitable for testing how models express both discretionary and reactive adaptation. Moreover, TAS exposes rich variability (alternative strategies, degradations, and sequencing) and clear, observable events that can be encoded as guards and runtime operators, providing an excellent proving ground for semantics-preserving translation to PRISM and for evaluating quantitative trade-offs such as utility, cost, and reliability.

The **Lab Samples Logistics (LSL)** scenario, in turn, offers a complementary cyber-physical perspective on adaptation. It models an autonomous system that coordinates robots, nurses, and laboratory devices to ensure timely and traceable transport of bi-

ological samples. Unlike TAS, which emphasizes decision-making under uncertainty in a service-oriented context, LSL foregrounds spatial coordination, safety constraints, and real-time resource management. Its adaptive behavior involves reacting to environmental conditions—such as battery levels, drawer access authorization, or communication failures—while maintaining operational continuity through fallback procedures. The combination of human and robotic actors, regulated access control, and timing requirements makes LSL particularly suited to evaluate how the EDGE framework captures distributed autonomy and runtime constraint enforcement within a verifiable, model-driven structure.

By applying our methodology to these distinct yet complementary domains, we demonstrate how enriched goal models capture dynamic operational intent and how their systematic translation supports formal reasoning and probabilistic verification within the EDGE framework. In the following subsections, we detail the application of our methodology to each system, illustrating how the artifacts developed in each phase were successively constructed, transformed, and employed in the subsequent stages of the translation process.

5.1 Case 1: Tele-Assistance System (TAS)

The Tele-Assistance Service (TAS) scenario represents a distributed healthcare support system designed to monitor patients remotely and react to emergencies. Patients periodically transmit vital parameters to the system, which analyzes the data to detect critical conditions and, if needed, escalates responses through medical and alert services.

The exemplar proposed by Weyns & Calinescu [14] defines the system and its requirements as follows:

> "The Tele Assistance System (TAS) provides health support to chronic-condition patients within their homes. It integrates sensors embedded in a wearable device with third-party healthcare, pharmacy, and emergency services. [...] TAS periodically measures a patient's vital parameters and sends them to a remote medical analysis service. The analysis result determines whether the system triggers a pharmacy service to deliver new medication or adjust dosage, or whether it invokes an alarm service to request emergency assistance such as dispatching an ambulance. The alarm service may also be activated directly by the patient using a panic button on the wearable device."

5.1.1 Constructing the EDGE Goal Model

To formalize the TAS scenario within the EDGE framework, the system's functional and adaptive requirements are decomposed into their constituent behavioral elements—goals, tasks, and resources—each representing a distinct layer of intentional reasoning or operational enactment. This decomposition bridges the stakeholder-oriented perspective of

GORE models with the formal structure of the EDGE metamodel, providing a systematic mapping from descriptive system objectives to analyzable constructs. The resulting goal model shown in Figure 5.1 establishes the foundation for the subsequent translation into PRISM, where these elements are rendered as verifiable modules of the adaptive goal controller.

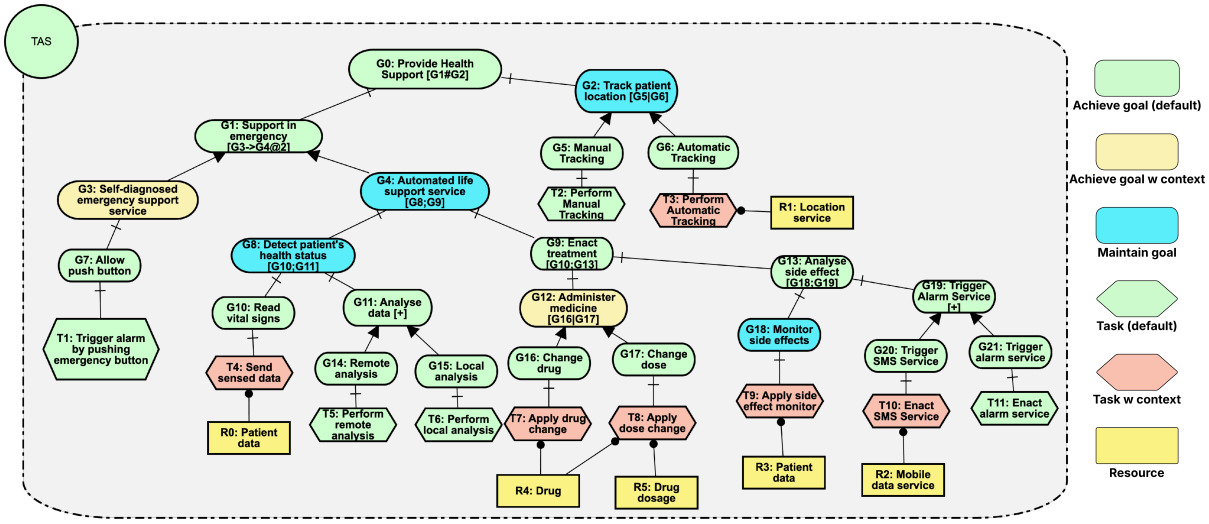


Figure 5.1: Goal model for TAS (Tele Assistance System)

The mapping in Table 5.1 bridges the TAS requirements with their formal representation in the EDGE metamodel. Each row links a behavioral event or functional requisite from the scenario to its corresponding goal, task, resource, and contextual guard in the model. This shows how key system functions—such as monitoring, analysis, treatment adaptation, and emergency handling—are grounded in the structural and semantic elements of EDGE, establishing a clear correspondence between real-world adaptive behavior and its goal model representation.

Table 5.1: Mapping of TAS textual requirements to EDGE model components

TAS Requirement / Event	**EDGE Goal**	**EDGE Task**	**Resource(s)**	**Context Guard**
Patient receives continuous health support	G0: Provide Health Support	—	—	—
Trigger manual emergency	G3: Self-diagnosed emergency support service	T1: Trigger alarm by pushing emergency button	—	`privacyEnabled | battery > 20`

Continued on next page

Table 5.1 (continued)

TAS Requirement / Event	EDGE Goal	EDGE Task	Resource(s)	Context Guard
Automated life support	G4: Automated life support service	—	—	`enoughBattery & highReliability` (maintain: `inEmergency`)
Monitor patient's vital signs periodically	G8: Detect patient's health status (Maintain)	T4: Send sensed data	R0: Patient data	`sensorAvailable` (maintain: `enoughBattery & highReliability`)
Track patient's location	G2: Track patient location (Maintain)	T2: Perform Manual Tracking, T3: Perform Automatic Tracking	R1: Location service	`patientTracking`, T2→`privacyEnabled`, T3→`R1>50`
Analyze received data	G11: Analyse data	T5: Perform remote analysis, T6: Perform local analysis	R0: Patient data	—
Adjust treatment based on results	G12: Administer medicine	T7: Apply drug change, T8: Apply dose change	R4: Drug, R5: Drug dosage	`pharmacyAvailable & atHome`
Handle side effects after treatment	G13: Analyse side effect, G18: Monitor side effects (Maintain)	T9: Apply side effect monitor	R3: Patient data	G18→`medicationApplied` (maintain: `inSideEffectWindow`), T9→`R3=true`
Escalate in emergencies	G19: Trigger Alarm Service	T10: Enact SMS Service, T11: Enact alarm service	R2: Mobile data service	G19→`inEmergency`, T10→`R4>35`, T11→`networkAvailable`

The workflow integrates several key events that map directly to the goal model's contextual guards and runtime conditions. Periodic monitoring of patient vitals (`sensorAvailable` → "Read vital signs" → T4: Send sensed data) ensures continuous assessment under sufficient system reliability and battery conditions (`enoughBattery & highReliability`). User-initiated alarms are triggered manually when privacy and battery constraints permit (`privacyEnabled | battery > 20` → T1: Trigger alarm by pushing emergency button), while automated and manual tracking of the patient's location depend on `patientTracking` status, with manual tracking guarded by `privacyEnabled` and automatic tracking activated when `R1>50`. Data analysis tasks (`T5` and `T6`) operate over patient information resources, feeding into treatment adaptation goals that become active when `pharmacyAvailable & atHome` hold. Finally, emergency escalation services (`T10`, `T11`) are pursued in response

to `inEmergency` conditions, provided that the network is operational (`networkAvailable`) and sufficient communication resources (`R4>35`) are available, ensuring reliable alert delivery through SMS and alarm channels.

The final element of the EDGE goal model concerns the *runtime annotations* declared by goals with multiple variants, whether connected through AND or OR decompositions. These annotations define the temporal and behavioral constraints that govern how subgoals are selected and coordinated during execution, complementing the contextual guards that regulate goal activation. By encoding the ordering, concurrency, or fallback logic among child goals, runtime annotations determine the adaptive decision flow that the controller follows at runtime. Table 5.2 summarizes the annotations defined for each goal in the TAS model, outlining their types, targets, and rationale. This mapping clarifies how each goal's runtime semantics—such as sequential, interleaved, or degradable execution—shape the adaptation behavior of the system in the generated PRISM specification.

Table 5.2: Goals and Runtime Annotations in the TAS Model

ID	**Name**	**Type**	**Runtime Annotation**	**Targets**	**Rationale**
G0	Provide Health Support	Achieve	[G1#G2] Interleaved	G1, G2	Top-level goal combining emergency support and tracking in parallel.
G1	Support in emergency	Achieve	[G3→G4@2] Degradation	G3, G4	Fallback from manual to automated life support after two failed attempts.
G2	Track patient location	Maintain	[G5 | G6] Alternative	G5, G6	Switches between manual and automatic tracking by context.
G4	Automated life support service	Maintain	[G8; G9] Sequential	G8, G9	Continuous life support: detect health, then enact treatment.
G8	Detect patient's health status	Maintain	[G10; G11] Sequential	G10, G11	Monitor: read sensors, then analyze data.
G9	Enact treatment	Achieve	[G10; G13] Sequential	G10, G13	Analyze data then apply treatment.
G11	Analyse data	Achieve	[+] Choice	T5, T6	Concurrent local or remote analyses.

Continued on next page

Table 5.2 (continued)

ID	Name	Type	Runtime Annotation	Targets	Rationale
G12	Administer medicine	Achieve	[G16 | G17] Alternative	G16, G17	Choose between drug or dose change.
G13	Analyse side effect	Achieve	[G18; G19] Sequential	G18, G19	Monitor side effects and trigger alerts if needed.
G19	Trigger Alarm Service	Achieve	[+] Choice	T10, T11	Send alerts via SMS or alarm services.

5.1.2 Building the IR for TAS EDGE

The second phase of the translation pipeline instantiates the `GoalTree` Intermediate Representation (IR) using the structural elements extracted from the TAS goal model (Figure 5.1). This step materializes the abstract goal hierarchy defined in Phase 1 into a structured, machine-readable model that serves as input to the template-based synthesis engine. While the goal model expresses the system's adaptive logic graphically, the IR provides an explicit data structure encoding every node's attributes—goal type, decomposition relation, execution semantics, context guards, and quantitative properties—according to the class diagram in Figure 3.16.

In the instantiated IR of the TAS goal model, each `GoalNode` object corresponds directly to a goal in the EDGE model, carrying the semantic information required for subsequent synthesis. The representation captures not only the hierarchical relations among goals but also the contextual, quantitative, and behavioral attributes defined in the original model. Specifically:

- Each `GoalNode` stores its `goalType` (*Achieve* or *Maintain*), its `relationToChildren` (*AND* or *OR*), and the associated `executionDetail`, which defines the runtime operator governing its child goals (e.g., *Sequence*, *Interleaved*, *Degradation*, *Any* or *Alternative*).

- The `Context` element of each goal contains an `context` field, representing the activation condition under which the goal can be pursued, and optionally a `maintain` field, expressing a persistent condition that must remain true throughout execution for *Maintain* goals.

- Each `Task` node maintains references to its associated `Resource` instances (`R0`–`R4`), defining the measurable system or environmental quantities required for task execution.

- Quantitative attributes inherited from the metamodel—`utility`, `cost`, `dependsOn`, and `maxRetries`—are instantiated with the values provided in the goal model, ensuring that the IR preserves both structural and behavioral information necessary for probabilistic reasoning.

In this structured instantiation all semantic elements of the TAS model are explicitly represented in the IR, forming a structured specification for the subsequent template-based translation into PRISM. Figure 5.2 depict two branches of the tree for goal `G1` as an example.

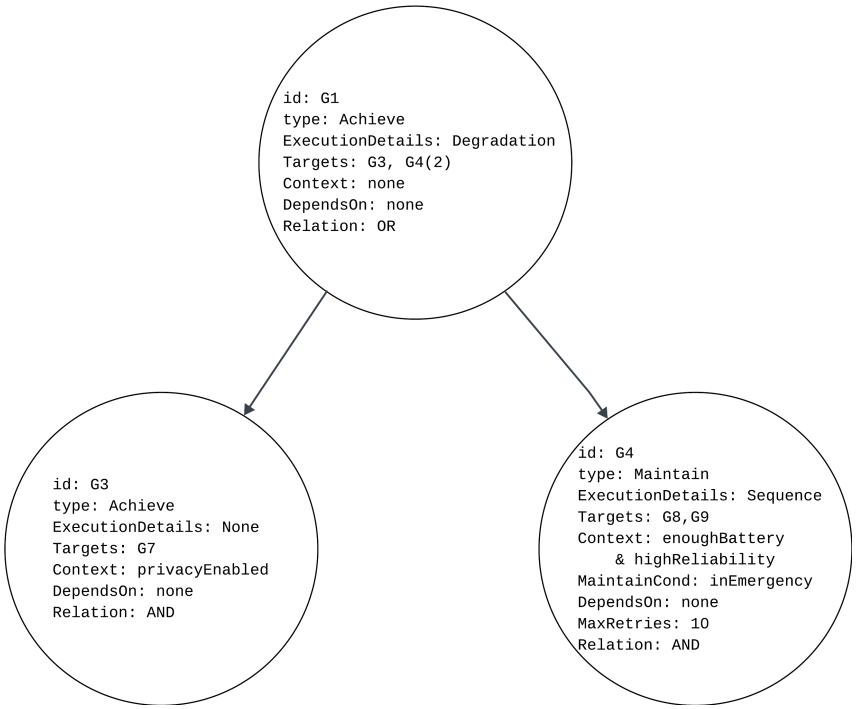


Figure 5.2: G1 IR subtree instance

5.1.3 Generating The PRISM Goal Controller

The final step of the transformation process consists in generating the PRISM specification that implements the synthesized EDGE Goal Controller. To complete this stage, each node type—`Goal`, `Task`, and `Resource`—within the `GoalTree` produced in the previous phase (see Section 5.1.2) is traversed by the template engine to emit the corresponding PRISM module definitions (Goals, ChangeManager and System modules).

Goal Modules The generation of PRISM goal modules follows the construction rules described in Section 3.5. For each goal, the template engine produces (i) the state variables that represent its lifecycle, (ii) its activation or `pursue` transition, and (iii) the propagation

of pursuit to its subgoals or tasks according to the runtime annotation semantics defined in Section 3.3.3. To illustrate the application of these transformation rules, we present the resulting PRISM modules for a representative subset of goals—G0, G1, G2, G4, G11, and G12—which collectively cover all the structural and behavioral variants present in the TAS model configuration.

G0: Provide Health Support The goal G0, shown in Listing 5.1, represents an **Interleaved AND** composition with two child goals, G1 and G2. It serves as the top-level coordinator, initiating both subgoals in parallel when active. As shown in line 5, G0 has no contextual preconditions, allowing its activation without dependency on external guards. Lines 6–7 encode the interleaving semantics, enabling either subgoal to be pursued independently while enforcing that G2's activation depends on its maintain condition (G2_achieved_maintain). Line 9 defines the synchronization rule for completion: the parent goal is achieved only when both children reach their achieved states (AND joint), note that since G2 is a maintain goal the achieve condition checks for its achieved_maintain formula. Finally, line 12 implements the skip transition, resetting G0 when neither child has been pursued after activation, ensuring that the goal can re-enter the decision cycle under new runtime conditions.

```
1  module G0
2    G0_pursued : [0..1] init 0;
3    G0_achieved : [0..1] init 0;
4
5    [pursue_G0] G0_pursued=0 & G0_achieved=0 -> (G0_pursued'=1);
6    [pursue_G1] G0_pursued=1 & G0_achieved=0 -> true;
7    [pursue_G2] G0_pursued=1 & G0_achieved=0 & G2_achieved_maintain=false -> true;
8
9    [achieved_G0] G0_pursued=1 & (G1_achieved=1 & G2_achieved_maintain=true) -> (
       ↪ G0_pursued'=0) & (G0_achieved'=1);
10
11   [skip_G0] G0_pursued=1 & G1_pursued=0 & G2_pursued=0 -> (G0_pursued'=0);
12 endmodule
```

Listing 5.1: PRISM module for Goal G0 (Interleaved)

G1: Support in emergency The goal G1, shown in Listing 5.2, represents a **Degradation OR** composition with two child goals, G3 and G4. A third variable, G4_failed, is introduced to count the number of times G4 has failed, enabling bounded retry semantics before degradation occurs. Line 6 activates the goal without any contex-

tual guard, allowing it to initiate as soon as its parent enables it. Lines 7–8 implement the degradation logic: `G4` is attempted first and can be retried up to two times before degrading to `G3`, ensuring controlled fallback behavior. Line 8 updates the `G4_failed` counter everytime G4 is pursued. Line 10 defines the achievement synchronization through an `OR` joint, marking `G1` as achieved once either `G3` or `G4]_achieved_maintain` reaches true. Finally, line 13 encodes the skip transition, resetting the goal if neither subgoal is pursued after activation—allowing the controller to re-evaluate the situation in the next decision cycle.

```
1  module G1
2    G1_pursued : [0..1] init 0;
3    G1_achieved : [0..1] init 0;
4    G4_failed : [0..10] init 0;
5
6    [pursue_G1] G1_pursued=0 & G1_achieved=0 -> (G1_pursued'=1);
7    [pursue_G3] G1_pursued=1 & G1_achieved=0 & G1_pursued=0 & G4_failed >= 2 ->
       ↪ true;
8    [pursue_G4] G1_pursued=1 & G1_achieved=0 & G1_pursued=0 & G4_achieved_maintain
       ↪ =false -> G4_failed=min(10, G4_failed+1);
9
10   [achieved_G1] G1_pursued=1 & (G3_achieved=1 | G4_achieved_maintain=true) -> (
       ↪ G1_pursued'=0) & (G1_achieved'=1);
11
12   [skip_G1] G1_pursued=1 & G3_pursued=0 & G4_pursued=0 -> (G1_pursued'=0);
13 endmodule
```

Listing 5.2: PRISM module for Goal G1 (Degradation OR)

G2: Track patient location The goal `G2`, shown in Listing 5.3, is a **Maintain** goal with an **Alternative OR** decomposition between its two children: `G5` (manual tracking) and `G6` (automatic tracking). Line 4 defines the activation condition for `G2`, which becomes active when its parent goal is enabled, the `achieved_maintain` condition evaluates to `false`, and the contextual guard `patientTracking` holds. Lines 5–6 encode the *alternative* execution semantics: each child's pursue label is guarded to enforce mutual exclusivity—`G5` may only be pursued if `G6` is inactive, and vice versa—ensuring that only one variant is selected at a time. Line 9 synchronizes achievement through an `OR` condition, marking `G2` as achieved once either subgoal completes, while line 11 defines the `skip` transition that resets the goal when no child is pursued within the current decision frame.

Finally, the maintain semantics are captured by the formula `G2_achieved_maintain = highPrecision`, which substitutes the standard achievement condition for `G2`. This formulation ensures that `G2` is considered continuously achieved only while the `highPrecision` condition remains true, allowing the controller to automatically re-activate tracking when precision degrades below the defined threshold.

```
1  module G2
2    G2_pursued : [0..1] init 0;
3
4    [pursue_G2] G2_pursued=0 & G2_achieved_maintain=false & (G4_achieved=1) &
       ↪ patientTracking -> (G2_pursued'=1);
5    [pursue_G5] G2_pursued=1 & G2_achieved_maintain=false & G6_pursued=0 -> true;
6    [pursue_G6] G2_pursued=1 & G2_achieved_maintain=false & G5_pursued=0 -> true;
7
8    [achieved_G2] G2_achieved_maintain=true & (G5_achieved=1 | G6_achieved=1) -> (
       ↪ G2_pursued'=0) & (G2_achieved'=1);
9
10   [skip_G2] G2_pursued=1 & G5_pursued=0 & G6_pursued=0 -> (G2_pursued'=0);
11 endmodule
12
13 formula G2_achieved_maintain = highPrecision;
```

Listing 5.3: PRISM module for Goal G2 (Alternative OR with Maintain Condition)

G4: Automated Life Support Service The goal `G4`, shown in Listing 5.4, implements a **Sequential AND** of type **Maintain** composition with two children, `G8` ("Detect patient's health status") and `G9` ("Enact treatment"). Activation occurs in line 5 when both contextual conditions —`enoughBattery` and `highReliability`—are satisfied, ensuring that automated life support can only proceed under reliable system and power conditions. Lines 6–7 encode the sequential behavior: `G8` must complete before `G9` can be pursued, reflecting the logical dependency between health monitoring and treatment execution. Line 9 synchronizes achievement through an `AND` joint, marking `G4` as achieved only when both child goals have been successfully completed. Finally, line 11 defines the skip transition, resetting the goal if neither child is pursued after activation, thereby allowing `G4` to re-enter the decision cycle under updated runtime conditions.

```
1  module G4
2    G4_pursued : [0..1] init 0;
3    G4_failed : [0..10] init 0;
```

```
4
5   [pursue_G4] G4_pursued=0 & G4_achieved_maintain=false & enoughBattery &
      ↪ highReliability -> (G4_pursued'=1);
6   [pursue_G8] G4_pursued=1 & G4_achieved_maintain=false & G8_achieved=0 &
      ↪ G8_achieved_maintain -> true;
7   [pursue_G9] G4_pursued=1 & G4_achieved_maintain=false & G8_achieved=1 -> true;
8
9   [achieved_G4] G4_achieved_maintain=true & (G8_achieved_maintain=true &
      ↪ G9_achieved=1) -> (G4_pursued'=0) & (G4_achieved'=1);
10
11  [skip_G4] G4_pursued=1 & G8_pursued=0 & G9_pursued=0 -> (G4_pursued'=0);
12 endmodule
13 formula G4_achieved_maintain = inEmergency;
```

Listing 5.4: PRISM module for Goal G4 (Sequential AND)

G11: Analyse Data The goal `G11`, shown in Listing 5.5, represents a **Choice OR** composition with two alternative children, `G14` (remote analysis) and `G15` (local analysis). A dedicated variable `G11_chosen`, declared in line 4, stores the selected branch, ensuring that once a variant is chosen, it remains consistent across future decision cycles. Line 6 activates the goal independently of any contextual guard, allowing the controller to select a variant whenever `G11` becomes eligible. Lines 7–8 encode the choice semantics: only one of the two subgoals can be pursued at a time, and the selected option is recorded in `G11_chosen` to maintain decision persistence across frames. Line 10 synchronizes achievement through an `OR` joint, marking `G11` as achieved when either child succeeds, while line 13 implements the skip transition, resetting the goal if no child was activated after `pursue_G11` was fired—allowing reevaluation in subsequent cycles.

```
1  module G11
2    G11_pursued : [0..1] init 0;
3    G11_achieved : [0..1] init 0;
4    G11_chosen : [0..1] init 0;
5
6    [pursue_G11] G11_pursued=0 & G11_achieved=0 -> (G11_pursued'=1);
7    [pursue_G14] G11_pursued=1 & G11_achieved=0 & G11_chosen!=1 -> G11_chosen=0;
8    [pursue_G15] G11_pursued=1 & G11_achieved=0 & G11_chosen!=0 -> G11_chosen=1;
9
10   [achieved_G11] G11_pursued=1 & (G14_achieved=1 | G15_achieved=1) -> (
       ↪ G11_pursued'=0) & (G11_achieved'=1);
11
```

```
12   [skip_G11] G11_pursued=1 & G14_pursued=0 & G15_pursued=0 -> (G11_pursued'=0);
13 endmodule
```

Listing 5.5: PRISM module for Goal G11 (Choice OR with persistent selection)

G12: Administer Medicine The goal `G12`, shown in Listing 5.6, is an **Alternative OR** goal of type **Achieve** with two children: `G16` ("Change drug") and `G17` ("Change dose"). Activation occurs in line 5 when both contextual guards—`pharmacyAvailable` and `atHome`—evaluate to true, ensuring that treatment adjustments are performed only when the patient is at home and pharmacy services are available. Lines 6–7 encode the alternative behavior: only one of the two subgoals can be pursued at a time, with mutual exclusion conditions (`G17_pursued=0` for `G16` and vice versa) preventing concurrent selection. This ensures that exactly one adaptation strategy—drug substitution or dosage adjustment—is executed during each decision frame. Line 9 synchronizes goal achievement through an `OR` condition, marking `G12` as achieved once either child completes successfully, while the skip transition in line 13 resets the goal if no child is pursued after `pursue_G12` is fired. Unlike `G2`, `G12` does not define a maintain formula, as it represents a discrete achieving behavior rather than a continuous condition to preserve.

```
1  module G12
2    G12_pursued : [0..1] init 0;
3    G12_achieved : [0..1] init 0;
4
5    [pursue_G12] G12_pursued=0 & G12_achieved=0 & (G9_achieved=1) &
       ↪ pharmacyAvailable & atHome -> (G12_pursued'=1);
6    [pursue_G16] G12_pursued=1 & G12_achieved=0 & G17_pursued=0 -> true;
7    [pursue_G17] G12_pursued=1 & G12_achieved=0 & G16_pursued=0 -> true;
8
9    [achieved_G12] G12_pursued=1 & (G16_achieved=1 | G17_achieved=1) -> (
       ↪ G12_pursued'=0) & (G12_achieved'=1);
10
11   [skip_G12] G12_pursued=1 & G16_pursued=0 & G17_pursued=0 -> (G12_pursued'=0);
12 endmodule
```

Listing 5.6: PRISM module for Goal G12 (Alternative OR with contextual guard)

ChangeManager Module The `ChangeManager` module orchestrates task-level execution and corresponds to the *Strategy Management* and *Strategy Enactment* layers in MORPH. Each task declares two status variables—`Ti_pursued` and `Ti_achieved`—along

with three synchronization commands: `[pursue_Ti]`, `[achieved_Ti]`, and `[failed_Ti]`. For tasks involved in degradation goals, an additional bounded counter `Ti_failed` is declared to support retry semantics.

In Listing 5.7, lines 2–3 define the status variables for `T1`, while lines 7–9 define those for `T5`, including the extra variable `T5_failed : [0..8] init 0` to record bounded retries. Lines 11–13 specify the default `pursue`, `achieved`, and `failed` transitions for `T1`. The corresponding transitions for `T5` are shown in lines 18–20, where the `[failed_T5]` rule (line 20) resets the pursuit flag and increments the failure counter using `min(8, T5_failed+1)`. Lines 1 and 21 delimit the module. Repeated declarations and transitions for additional tasks are omitted for brevity.

```
1  module ChangeManager
2    T1_pursued : [0..1] init 0;
3    T1_achieved : [0..1] init 0;
4    // ... rest of the variables
5
6    // T5 declares maxRetries property
7    T5_pursued : [0..1] init 0;
8    T5_achieved : [0..1] init 0;
9    T5_failed: [0..8] init 0;
10
11   [pursue_T1] true -> true;
12   [achieved_T1] true -> true;
13   [failed_T1] T1_pursued=1 & T1_achieved=0 -> (T1_pursued'=0);
14
15   // ... repeat for each task
16
17   // T5 updates the counter when it fails
18   [pursue_T5] true -> true;
19   [achieved_T5] true -> true;
20   [failed_T5] T5_pursued=1 & T5_achieved=0 -> (T5_pursued'=0) & (T5_failed'=min
       ↪ (8, T5_failed+1));
21 endmodule
```

Listing 5.7: ChangeManager PRISM module

System module The `System` module provides the global, declarative state on which all goal-level decisions depend. It contains *only* variable declarations—no transitions—thus serving as the shared context read by `pursue` guards across goal modules. Boolean variables encode propositional conditions (e.g., service availability, runtime modes), whereas

bounded integers capture quantitative resources with explicit domains to constrain the state space and ensure type safety. Initial values establish the reference configuration from which PRISM explores the controller's behavior.

Listing 5.8 shows the TAS system module: lines 2–12 declare the contextual booleans used in activation and maintain guards; lines 13–18 declare resources with their types and bounds (`R1`, `R2`, `R4`, `R5` as integers; `R0`, `R3` as booleans). Lines 1 and 19 delimit the module.

```
1  module System
2    pharmacyAvailable: bool init true;
3    atHome: bool init false;
4    inSideEffectWindow: bool init false;
5    medicationApplied: bool init false;
6    inEmergency: bool init false;
7    highPrecision: bool init false;
8    patientTracking: bool init false;
9    privacyEnabled: bool init false;
10   enoughBattery: bool init true;
11   highReliability: bool init false;
12   sensorAvailable: bool init true;
13   R0: bool init true;
14   R1: [0..100] init 50;
15   R2: [0..100] init 50;
16   R3: bool init true;
17   R4: [1..4] init 1;
18   R5: [0..10] init 0;
19 endmodule
```

Listing 5.8: System PRISM module for TAS

Taken together, the goal, `ChangeManager`, and `System` modules shown above instantiate a translation of the TAS goal model into an executable PRISM specification. They operationalize the EDGE desiderata—capturing alternative variants, quantitative/contextual guards, status tracking, and inter-goal dependencies—while maintaining a clean separation between decision logic and enactment consistent with MORPH. The full TAS controller (including all modules and auxiliary formulas) is provided in Chapter B for reproducibility and further analysis. We next apply the same translation methodology to the Lab Samples Logistics (LSL) exemplar, where concerns such as, secure handling, and resource constraints shape the enriched goal model and its PRISM realization.

5.2 Case 2: Lab Samples Logistics (LSL)

The Lab Samples Logistics (LSL) scenario exemplifies a healthcare-related robotic mission within a hospital environment, focusing on the secure and timely transport of biological samples from patient floors to the laboratory. The system supports nurses in requesting and tracking deliveries, ensuring that each sample—identified by a barcode and registered in a tracking system—is handled under strict security and authorization protocols. Transport drawers are locked and can only be opened by specific robots or authorized personnel, maintaining both safety and data integrity. Upon arrival, laboratory robots or technicians handle, scan, and log each sample for analysis, while timing information between collection and delivery is continuously monitored to comply with clinical standards [15].

This case extends the repository of robotic mission adaptation exemplars proposed by Askarpour *et al.* [15] (original exemplar quoted bellow), illustrating adaptation and coordination challenges in medical logistics under security and resources constraints.

> "Lab samples should be transported from patient floors to the laboratory. A nurse responsible for collecting a sample can request a delivery to the laboratory. Each sample before collection is identified with a barcode. The nurse inserts data about the sample in a track system. Samples are transported in secure locked drawers. In the laboratory, the sample can be picked by a robotic arm or laboratory personnel. Only authorized personnel or specific robots in specific locations should be able to open the locked drawers. The robotic arm picks up samples, scans the barcode in each sample, sorts, and loads the samples into the entry-module of the analysis machines. Information about where and when a given sample was picked should be logged securely. The time between sample collection and laboratory delivery should be tracked to be sure that it goes according to laboratory protocols."

5.2.1 Constructing the EDGE Goal Model

To formalize the Lab Samples Logistics (LSL) scenario within the EDGE framework, the narrative description of the laboratory transport process was decomposed into its constituent behavioral elements: goals, tasks, and resources. This decomposition captures the system's operational intent—secure, traceable, and adaptive transport of biological samples from patient wards to the laboratory—and provides the structural basis for the verifiable model encoded in PRISM. Following the same methodology adopted for the TAS case, the LSL model bridges the descriptive workflow of hospital logistics with the analyzable constructs of the EDGE metamodel, thereby linking organizational intent to runtime adaptation logic.

The resulting goal model, shown in Figure 5.3, captures the main adaptive elements of the LSL system. The top-level goal `G0: Provide Sample Delivery` defines the overall

mission, refined into two interleaved subgoals: `G1: Monitor Sample Track System` and `G2: Deliver Sample and Pickup Sample if Needed`. This decomposition reflects the dual requirement of maintaining continuous awareness of sample status while coordinating the physical delivery and retrieval processes. Each subgoal encapsulates multiple refinement layers that represent both human and robotic actions, contextual conditions, and monitored system states.

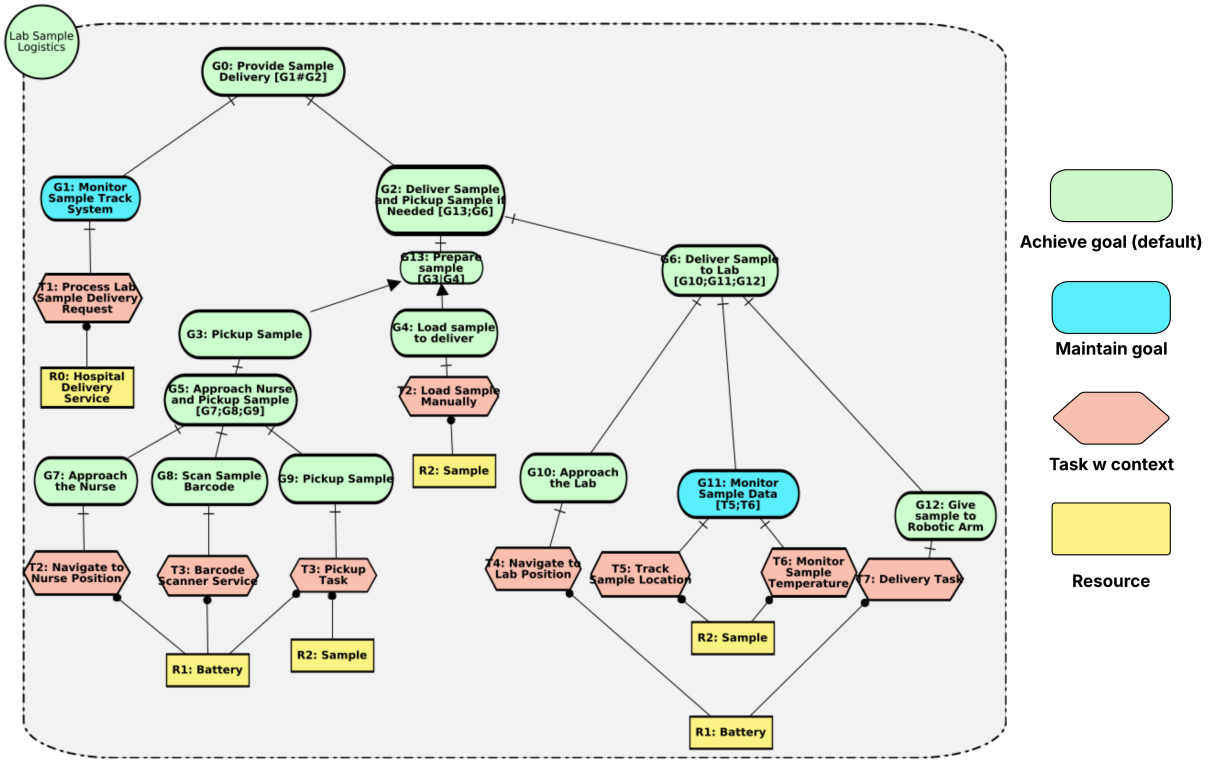


Figure 5.3: EDGE Goal model for LSL

The mapping in Table 5.3 links narrative-level activities—such as sample pickup, barcode scanning, and delivery confirmation—to their formal EDGE counterparts. Each mapping specifies the associated goal, task, and resource dependencies, together with the contextual guards that regulate activation and execution. For example, the robot's pickup task (`T3: Pickup Task`) executes under sufficient battery charge and valid sample availability (`R1>50 & R2=true`). Delivery tasks (`T7: Delivery Task`) execute under `R1>30`, ensuring that the system maintains energy efficiency throughout transport. Maintaining the sample's traceability and condition is represented by `G1: Monitor Sample Track System` and `G11: Monitor Sample Data`, both modeled as *maintain* goals whose activation depends on context assertions such as `sampleLoaded` and `analysisEnded`.

The model integrates a set of runtime guards that connect the operational flow to the adaptive reasoning cycle. For instance, a maintain goal such as `G11: Monitor Sample`

Data continuously verifies environmental conditions during transport, while the success of T6: Monitor Sample Temperature updates the contextual variable analysisEnded, enabling downstream goals related to laboratory handling. Similarly, degradation and interleaving annotations govern how delivery and pickup processes proceed under varying conditions such as battery depletion or unavailable personnel.

Table 5.3: Mapping of LSL textual requirements to EDGE model components

LSL Requirement / Event	**EDGE Goal**	**EDGE Task**	**Resource(s)**	**Context Guard**
Request delivery for collected sample	G0: Provide Sample Delivery	T1: Process Lab Sample Delivery Request	R0: Hospital Delivery Service	R0=true
Monitor tracking system during transport	G1: Monitor Sample Track System (Maintain)	—	R0: Hospital Delivery Service	sampleLoaded (maintain: sampleLoaded)
Load and prepare sample for transport	G4: Load Sample to Deliver	T2: Load Sample Manually	R2: Sample	R2=true (sideEffect: sampleLoaded=true)
Pickup sample from nurse station	G5: Approach Nurse and Pickup Sample	T2: Navigate to Nurse Position, T3: Pickup Task	R1: Battery, R2: Sample	R1>60, R1>50 & R2=true (sideEffect: sampleLoaded=true)
Scan and verify barcode	G8: Scan Sample Barcode	T3: Barcode Scanner Service	R1: Battery	R1>=20
Deliver sample to laboratory	G6: Deliver Sample to Lab	T4: Navigate to Lab Position, T7: Delivery Task	R1: Battery, R2: Sample	R1>=50, R1>30 (sideEffect: sampleLoaded=false)
Monitor environmental conditions	G11: Monitor Sample Data (Maintain)	T5: Track Sample Location, T6: Monitor Sample Temperature	R1: Battery, R2: Sample	sample=true, R2=true (maintain: analysisEnded=true)
Deliver sample to robotic arm	G12: Give Sample to Robotic Arm	—	—	analysisEnded=true

Runtime annotations defined for LSL goals are summarized in Table 5.4. These specify the temporal relations and adaptation strategies between subgoals, encoding sequencing, concurrency, and fallback logic within the model. For example, the top-level goal G0 combines monitoring and delivery processes in parallel ([G1#G2]), while G3: Pickup

`Sample` enforces a sequential order between preparation and transport actions (`[G5;G6]`). Monitoring and delivery sequences (`[G10;G11;G12]`) ensure that environmental checks precede handover to the robotic arm, aligning adaptive control with physical process safety.

Table 5.4: Goals and Runtime Annotations in the LSL Model

ID	**Name**	**Type**	**Runtime Annotation**	**Targets**	**Rationale**
G0	Provide Sample Delivery	Achieve	[G1#G2] Interleaved	G1, G2	Execute monitoring and delivery in parallel to ensure traceability during transport.
G2	Deliver and Pickup Sample if Needed	Achieve	[G13;G6] Sequential	G13, G6	Prepare and deliver samples in a structured workflow.
G3	Pickup Sample	Achieve	[G5;G6] Sequential	G5, G6	Pickup and deliver samples in defined order.
G5	Approach Nurse and Pickup Sample	Achieve	[G7;G8;G9] Sequential	G7, G8, G9	Approach, scan barcode, and collect the sample.
G6	Deliver Sample to Lab	Achieve	[G10;G11;G12] Sequential	G10, G11, G12	Approach lab, monitor conditions, and hand over sample.
G11	Monitor Sample Data	Maintain	[T5;T6] Sequential	T5, T6	Track location and temperature until analysis completes.
G13	Prepare Sample	Achieve	[+] Choice	T2, T3	Choose between manual or automated sample preparation.

The LSL goal model then formalizes the adaptive coordination of robotic and human agents in the hospital logistics workflow. It embeds safety and resource requirements as context-aware activation and maintain conditions, while runtime annotations define adaptive control policies that manage resource availability and operational sequencing. This structured representation provides a verifiable foundation for model checking and runtime assurance within the EDGE–PRISM toolchain.

5.2.2 Building the IR for LSL EDGE

The second phase of the translation pipeline instantiates the `GoalTree` Intermediate Representation (IR) from the structural elements of the Lab Samples Logistics (LSL) goal model. As in the TAS case, this step materializes the abstract goal hierarchy into a machine-readable structure consumed by the template-based synthesis engine. While the graphical model expresses adaptive logic declaratively, the IR encodes, for every node, its goal type, refinement relation, execution semantics, context guards, and quantitative attributes, in accordance with the IR class diagram introduced previously. The top-level mission `G0: Provide Sample Delivery` is refined into an interleaving of monitoring and delivery concerns (`[G1#G2]`), which drives the shape of the IR root and its immediate children.

IR instantiation overview. Each `GoalNode` in the IR corresponds one-to-one with an EDGE goal; edges represent AND/OR refinements with explicit runtime operators captured in `executionDetail`. In the LSL model: (i) `G0` is decomposed into `G1` and `G2` with *Interleaved* execution (`[G1#G2]`), (ii) `G3` enforces a *Sequence* between preparation and delivery (`[G5;G6]`), (iii) `G5` sequences approach, scan, and pickup (`[G7;G8;G9]`), and (iv) `G6` sequences approach-lab, monitor-in-transit, and handover (`[G10;G11;G12]`). These runtime operators are recorded in `executionDetail` fields for the corresponding `GoalNode`s.

Context and maintain constraints. The IR attaches a `Context` object to each goal. Two goals are modeled as *maintain*: `G1: Monitor Sample Track System` maintains `sampleLoaded`, and `G11: Monitor Sample Data` maintains `analysisEnded`. Accordingly, the IR captures `context` and (when applicable) `maintain` fields: for `G1`, `context = sampleLoaded` and `maintain = sampleLoaded`; for `G11`, `context = sampleLoaded` with `maintain = analysisEnded`. These invariants regulate pursuit and persistence conditions of the corresponding `GoalNode`s.

Tasks and resource bindings. `Task` nodes in the IR reference the resources required for execution and encode their guards as Boolean/arithmetic predicates. In the LSL model, the nurse-delivery request is processed by `T1` guarded by `R0=true`; pickup and navigation rely on battery thresholds (`T2: R1>60`, `T3: R1>50 & R2=true`, `Barcode Scanner: R1` ≥ 20); delivery and lab navigation are guarded by `R1>30` and `R1` ≥ 50, respectively. These guards populate the `Task.context` fields, and the associated resource references (`R0`, `R1`, `R2`) are stored under `Task.resources`. `R1` is an integer battery with

bounds $[0, 100]$ and initial value 100; `R2` (sample presence) and `R0` (delivery service availability) are Boolean resources with initial `true`.

Quantitative and structural attributes. Quantitative attributes defined by the metamodel (`utility`, `cost`, `dependsOn`, `maxRetries`) are included in the IR schema; where the LSL source model specifies values they are copied verbatim, otherwise they remain at default. The `relationToChildren` for `G0`, `G3`, `G5`, and `G6` is *AND* (with their operators recorded in `executionDetail`), while `G13` represents a choice (`[+]`), captured as *OR/* at the IR level.

Figures 5.4 illustrate the instantiated IR for the G3 branch.

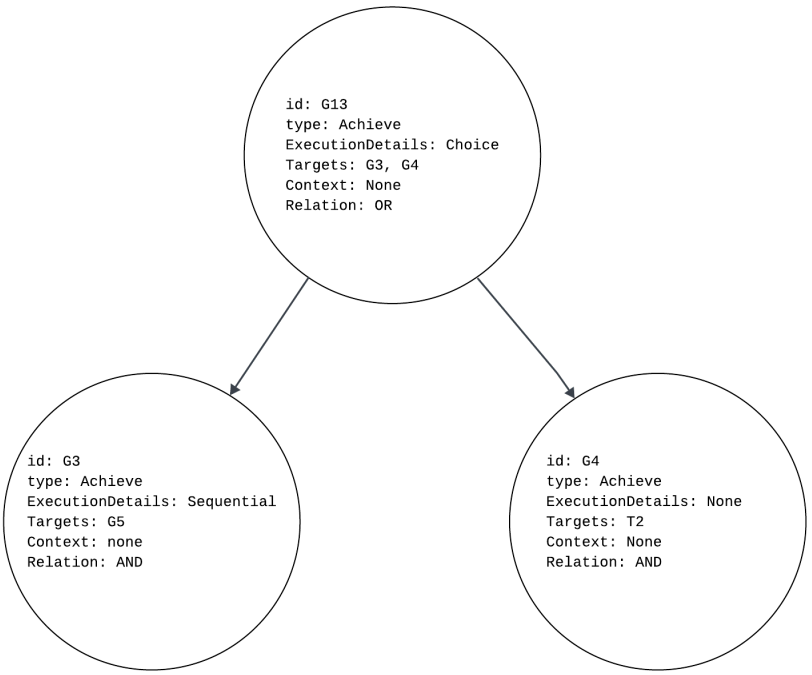


Figure 5.4: LSL IR subtree instance for `G1: Monitor Sample Track System`.

5.2.3 Generating the PRISM Goal Controller

The final stage synthesizes the PRISM specification of the goal controller. Following the procedure in section 5.1.3, we traverse the `GoalTree` and emit the corresponding PRISM artifacts for each node type—*Goal*, *Task*, and *Resource*. Concretely, the generator instantiates (i) one PRISM module per goal, encoding pursue/achieve/skip logic and runtime annotations; (ii) a `ChangeManager` module, capturing task-level execution and failure counters; and (iii) a `System` module, declaring contextual and resource variables with their initial values and domains. The remainder of this subsection details the con-

struction of these three modules and their synchronization semantics in the synthesized controller for the LSL example.

Goal Modules The generation of PRISM goal modules for the Lab Samples Logistics (LSL) scenario follows the same construction principles outlined in Section 3.5. Each goal is translated into a PRISM module that captures its decision dynamics through (i) lifecycle state variables, (ii) activation (`pursue`) transitions, and (iii) the propagation of pursuit to its child goals or tasks according to the runtime annotation semantics defined in Section 3.3.3.

To demonstrate the application of these transformation rules, we present the PRISM modules generated for a representative subset of goals—`G0`, `G1`, `G2`, `G6`, and `G13`. These modules capture the principal structural and behavioral variants within the LSL model: from high-level orchestration (`G0: Provide Sample Delivery`) to continuous process supervision (`G1: Monitor Sample Track System`), adaptive logistics coordination (`G2: Deliver Sample and Pickup Sample if Needed`), procedural handling of physical transport (`G6: Deliver Sample to Lab`), and concurrent preparation workflows (`G13: Prepare Sample`). Together, these goals illustrate how the LSL model integrates sequential, interleaved, and maintain-type compositions to ensure reliable, and context-aware operation of robotic sample delivery in a hospital environment.

G0: Provide Sample Delivery The goal `G0`, depicted in Listing 5.9, represents an **Interleaved AND** composition with two child goals, `G1` and `G2`. Lines 2–3 declare the lifecycle variables that track the pursuit and achievement status of the goal. Line 5 activates `G0`, which, in the context of the LSL model, requires no specific contextual guard. Lines 6–7 encode the interleaved decision logic: `G1`, being a *maintain*-type goal, can only be pursued when its maintenance condition evaluates to true, whereas `G2` may be pursued unconditionally. Line 9 defines the synchronization condition, marking `G0` as achieved once both subgoals complete successfully, and line 11 implements the skip transition, resetting the goal when no children are selected within the current decision frame. This configuration captures the concurrent yet coordinated orchestration between the transport and handling processes that underpin reliable sample delivery in the logistics workflow.

```
1 module G0
2   G0_pursued : [0..1] init 0;
3   G0_achieved : [0..1] init 0;
4
5   [pursue_G0] G0_pursued=0 & G0_achieved=0 -> (G0_pursued'=1);
6   [pursue_G1] G0_pursued=1 & G0_achieved=0 & G1_achieved_maintain -> true;
```

```
7   [pursue_G2] G0_pursued=1 & G0_achieved=0 -> true;
8
9   [achieved_G0] G0_pursued=1 & (G1_achieved=1 & G2_achieved=1) -> (G0_pursued'
      ↪ =0) & (G0_achieved'=1);
10
11  [skip_G0] G0_pursued=1 & G1_pursued=0 & G2_pursued=0 -> (G0_pursued'=0);
12 endmodule
```

Listing 5.9: PRISM module for Goal G0 (Interleaved AND)

G1: Monitor Sample Tracking System The goal `G1`, shown in Listing 5.10, represents an **AND Maintain** goal without an explicit runtime annotation. Since it contains only a single child, the semantics default to an interleaved structure with a single subordinate task. Lines 2–3 define the lifecycle state variables for pursuit and achievement. Line 5 governs the activation of `G1`, which is pursued only when the contextual condition `sampleLoaded` evaluates to true. Once activated, it invariably triggers the execution of task `T1` in sequence. Line 8 defines the synchronization condition for goal achievement, marking `G1` as completed once `T1` succeeds, while line 10 resets the goal in cases where the task is not pursued. Finally, `G1` declares a *maintain* formula to be referenced by its parent goal, ensuring the continuous validity of the tracking process throughout sample transport.

```
1  module G1
2    G1_pursued : [0..1] init 0;
3    G1_achieved : [0..1] init 0;
4
5    [pursue_G1] G1_pursued=0 & G1_achieved=0 & sampleLoaded -> (G1_pursued'=1);
6    [pursue_T1] G1_pursued=1 & G1_achieved=0 -> true;
7
8    [achieved_G1] G1_pursued=1 & (T1_achieved=1) -> (G1_pursued'=0) & (
       ↪ G1_achieved'=1);
9
10   [skip_G1] G1_pursued=1 & T1_pursued=0 -> (G1_pursued'=0);
11 endmodule
12 formula G1_achieved_maintain = sampleLoaded;
```

Listing 5.10: PRISM module for Goal G1 (Achieve with contextual guard)

G2: Deliver Sample and Pickup Sample if Needed The goal `G2`, depicted in Listing 5.11, is a **Sequential AND** goal composed of two child goals: `G13` and `G6`.

Lines 2–3 declare the lifecycle variables governing pursuit and achievement. Line 5 activates the goal whenever it is selected, as no contextual guard constrains its activation. Lines 6–7 encode the sequential execution semantics, ensuring that `G6` can only be pursued after `G13` has been achieved, thereby enforcing the prescribed order of delivery followed by pickup. Line 9 defines the synchronization condition under the `AND` relation, marking the goal as achieved once both child goals have successfully completed. Finally, line 12 implements the skip transition, resetting the goal's state when no child is pursued within the current decision frame. This structure reflects the controlled, order-sensitive coordination between delivery and retrieval operations essential to maintaining workflow continuity in the logistics process.

```
1  module G2
2    G2_pursued : [0..1] init 0;
3    G2_achieved : [0..1] init 0;
4
5    [pursue_G2] G2_pursued=0 & G2_achieved=0 -> (G2_pursued'=1);
6    [pursue_G13] G2_pursued=1 & G2_achieved=0 & G13_achieved=0 -> true;
7    [pursue_G6]  G2_pursued=1 & G2_achieved=0 & G13_achieved=1 & G6_achieved=0 ->
       ↪ true;
8
9    [achieved_G2] G2_pursued=1 & (G13_achieved=1 & G6_achieved=1)
10       -> (G2_pursued'=0) & (G2_achieved'=1);
11
12   [skip_G2] G2_pursued=1 & G13_pursued=0 & G6_pursued=0
13       -> (G2_pursued'=0);
14 endmodule
```

Listing 5.11: PRISM module for Goal G2 (Sequential AND)

G6: Deliver Sample to Laboratory The goal `G6`, shown in Listing 5.12, represents a **Sequential AND** composition with three child goals: `G10`, `G11`, and `G12`. Its semantics extend the structure demonstrated for `G2`, illustrating that the sequential coordination mechanism generalizes to any number of subgoals, not merely pairs. Lines 2–3 declare the pursuit and achievement variables that define the goal's lifecycle. Line 5 activates the goal whenever it is selected, while lines 6–8 encode the ordered execution: `G10` is pursued first, followed by `G11`—which requires both `G10` to be achieved and its own maintain condition to hold—and finally `G12`, which depends on the successful completion of both preceding subgoals. Line 10 specifies the synchronization of achievement under the `AND` relation, marking the goal as completed once all three subgoals succeed. Finally,

line 12 defines the skip transition, resetting the goal if no children are pursued in the current decision frame. This module exemplifies a multi-stage sequential coordination, ensuring that the delivery, verification, and confirmation phases of the sample transfer process occur in the correct order and under validated conditions.

```
1  module G6
2    G6_pursued : [0..1] init 0;
3    G6_achieved : [0..1] init 0;
4
5
6
7    [pursue_G6] G6_pursued=0 & G6_achieved=0 -> (G6_pursued'=1);
8    [pursue_G10] G6_pursued=1 & G6_achieved=0 & G10_achieved=0 -> true;
9    [pursue_G11] G6_pursued=1 & G6_achieved=0 & G10_achieved=1 & G12_achieved=0 &
       ↪ G11_achieved_maintain -> true;
10   [pursue_G12] G6_pursued=1 & G6_achieved=0 & G10_achieved=1 & G11_achieved=1 ->
       ↪  true;
11
12   [achieved_G6] G6_pursued=1 & (G10_achieved=1 & G11_achieved=1 & G12_achieved
       ↪ =1) -> (G6_pursued'=0) & (G6_achieved'=1);
13
14   [skip_G6] G6_pursued=1 & G10_pursued=0 & G11_pursued=0 & G12_pursued=0 -> (
       ↪ G6_pursued'=0);
15 endmodule
```

Listing 5.12: PRISM module for Goal G6 (Sequential AND with Maintain Condition)

G13: Prepare Sample The goal `G13`, shown in Listing 5.13, represents an **Alternative OR** composition with two variants, `G3` and `G4`. This configuration models a re-evaluative decision process in which the controller dynamically determines, at each decision frame, which child goal should be pursued based on the current system context. Lines 2–4 declare the module's lifecycle variables, including the auxiliary variable `G13_chosen`, which records the last selected alternative to maintain decision consistency across frames. Line 5 governs the activation of the goal itself, while lines 6–7 encode the mutual exclusion conditions that ensure only one of the two alternatives can be pursued at a time. Line 10 defines the synchronization of achievement under the `OR` relation, marking the goal as achieved once either of the selected variants successfully completes. Finally, line 13 introduces the skip transition, resetting the goal when no variant is pursued within the current decision frame. This module exemplifies the decision-making flexibility embedded in the LSL model, allowing the system to select between equivalent preparation

strategies based on evolving runtime conditions while ensuring mutual exclusivity and behavioral consistency.

```
module G13
  G13_pursued : [0..1] init 0;
  G13_achieved : [0..1] init 0;
  G13_chosen : [0..2] init 0;

  [pursue_G13] G13_pursued=0 & G13_achieved=0 -> (G13_pursued'=1);
  [pursue_G3]  G13_pursued=1 & G13_achieved=0 & G13_chosen!=1 -> true;
  [pursue_G4]  G13_pursued=1 & G13_achieved=0 & G13_chosen!=0 -> true;

  [achieved_G13] G13_pursued=1 & (G3_achieved=1 | G4_achieved=1)
      -> (G13_pursued'=0) & (G13_achieved'=1);

  [skip_G13] G13_pursued=1 & G3_pursued=0 & G4_pursued=0
      -> (G13_pursued'=0);
endmodule
```

Listing 5.13: PRISM module for Goal G13 (Choice OR with Persistent Selection)

ChangeManager Module The `ChangeManager` module encapsulates the task-level orchestration for the LSL goal controller, following the same structural and behavioral principles described in Section 5.1.3. Each task node is represented by two Boolean state variables—`pursued` and `achieved`—which track the task's execution lifecycle. In addition, three synchronization labels are declared for each task: `[pursue_Tn]`, `[achieved_Tn]`, and `[failed_Tn]`, which coordinate the interaction between task execution and higher-level goal modules. These labels ensure consistent propagation of task status across the controller hierarchy, maintaining coherence between operational enactment and goal reasoning. Listing 5.14 illustrates the general structure of the `ChangeManager` module, exemplified for tasks `T1` and `T2`; additional tasks follow the same declaration and synchronization pattern.

```
module ChangeManager

  T1_pursued : [0..1] init 0;
  T1_achieved : [0..1] init 0;

  T2_pursued : [0..1] init 0;
  T2_achieved : [0..1] init 0;

```

```
9   // ... repeats declaration for each task
10
11  [pursue_T1] true -> true;
12  [achieved_T1] true -> true;
13  [failed_T1] T1_pursued=1 & T1_achieved=0 -> (T1_pursued'=0);
14
15
16  [pursue_T2] true -> true;
17  [achieved_T2] true -> true;
18  [failed_T2] T2_pursued=1 & T2_achieved=0 -> (T2_pursued'=0);
19
20  // ... repeats labels for each task
21 endmodule
```

Listing 5.14: ChangeManager PRISM module

System module The `System` module for the LSL controller follows the same structure described for the TAS model in Section 5.1.3. It defines the contextual and resource variables used throughout the controller to evaluate activation and maintenance conditions for goals and tasks. An important detail is that, although some resources (such as `R1: Battery` and `R2: Sample`) appear multiple times across the goal hierarchy, they correspond to single shared variable identifiers declared globally within this module.

Listing 5.15 presents the PRISM declaration of the `System` module. Lines 2–3 define the Boolean context variables used in pursue and maintain guards; lines 4–6 declare the system resources, specifying their types and ranges (`R1` as an integer, `R0` and `R2` as Booleans). Lines 1 and 7 delimit the module.

```
1 module System
2   sampleLoaded: bool init false;
3   analysisEnded: bool init false;
4   R0: bool init true;
5   R1: [0..100] init 100;
6   R2: bool init true;
7 endmodule
```

Listing 5.15: System PRISM module for TAS

In summary, the LSL case demonstrates how narrative logistics requirements can be rendered as an analyzable, execution-ready goal controller by instantiating EDGE constructs (goals, tasks, resources), enforcing context/maintain guards, and encoding runtime annotations for sequencing, interleaving, and choice within PRISM. The synthe-

sized controller preserves the intended coordination between human and robotic activities—covering preparation, pickup, transport, monitoring, and handover—while ensuring traceability and safety through explicit guards and synchronization rules. For completeness and reproducibility, the full PRISM specification generated for LSL is provided in Chapter C. In the next chapter, we present a concrete implementation of the proposed translation methodology, detailing the toolchain, templates, and code artifacts that automate the synthesis of EDGE goal controllers from enriched goal models.

Chapter 6

Translation Evaluation

Evaluating the proposed goal-to-PRISM translation method requires returning to the central motivation of this dissertation: ensuring that enriched goal models remain traceable, semantically coherent, and formally verifiable when transformed into probabilistic controllers. The methodological path designed in Chapter 4—model parsing, semantic interpretation, template-based synthesis, and PRISM serialization—now culminates in a systematic analysis of the generated artifact. This chapter presents the empirical evidence obtained from applying the translator to the case studies first introduced in Chapter 5 (TAS and LSL) (and an additional toy model for a drone delivery system that helps on the evaluation of all the possible elements the syntax can express) and conducting probabilistic verification through PCTL properties, thereby operationalizing the evaluation criteria established by the research questions.

Before diving into the results, it is useful to briefly revisit the translation pipeline that guides the evaluation: The enriched i* model is first parsed into an intermediate, runtime-aware structure that captures the semantics of the syntactical elements present in the model. This structure is then compiled into modular DTMC specifications, instantiating the EDGE semantics in a PRISM controller. The resulting PRISM model then provides a verifiable representation of the system's runtime controller, enabling both structural inspection and probabilistic analysis. The evaluation presented in this chapter assesses whether the transformation preserves the intent and behavior encoded in the original goal model.

6.1 Goal–Question–Metric

The evaluation phase of the translation pipeline relies on the Goal–Question–Metric (GQM) [25] framework to structure its evaluation. GQM provides a disciplined way to relate the overarching aims of the method to specific questions and measurable outcomes.

It suits this empirical setting because the translator influences several complementary dimensions: structural traceability, semantic completeness, and the behavioral validity of the synthesized controller. Each of these dimensions corresponds directly to one of the research questions defined in Chapter 3.

By breaking each research question into sub-questions and associated metrics, GQM keeps the evaluation aligned with the intended capabilities of the translation pipeline and ensures that each result can be tied back to an explicit objective. This structure also supports transparency: every metric reported later in the chapter emerges from a clearly stated goal and a question designed to probe that goal.

The tables that follow summarize the three GQM organization that guide this evaluation. Each triad specifies (i) the validation goal derived from one research question, (ii) the questions used to refine that goal, and (iii) the metrics that supply observable evidence. These triads also determine the organization of the subsequent sections.

Table 6.1: GQM Table 1 — Goal 1 (RQ1): Traceability

Goal G1	**G1 Questions**	**Metrics**
Evaluate whether the translation preserves traceability between the goal model and the synthesized controller.	Q1.1: Are all goals, tasks, and resources represented in the PRISM model?	M1 Coverage ratio (nodes/-tasks/resources) M2 Missing-element count
	Q1.2: Is the goal hierarchy and execution structure preserved?	M3 Correctness of parent/child mappings M4 Guard/synchronization consistency

Table 6.2: GQM Table 2 — Goal 2 (RQ2): Expressiveness & Semantic Completeness

Goal G2	**G2 Questions**	**Metrics**
Assess whether the extended syntax and runtime semantics are expressively captured and semantically preserved in the generated goal controller	Q2.1: Are all enriched constructs used in the model translated without loss?	M5 Construct-to-template coverage M6 Per-construct LoC generated
	Q2.2: Do achieve and maintain goals behave as intended?	M7 Achievability and Maintain formulas M8 Presence of maintain guards
	Q2.3: Is the resulting PRISM model complete and analyzable?	M9 PRISM compilation feasibility M10 Generation error count M11 Model size metrics

Table 6.3: GQM Table 3 — Goal 3 (RQ3): Verification via PCTL

Goal G3	**G3 Questions**	**Metrics**
Determine whether PCTL properties confirm semantics, liveness, and safety of the synthesized controller.	Q3.1: Do liveness properties hold?	M12 Probabilities for eventual achievement M13 Number of satisfied liveness queries
	Q3.2: Do safety properties hold?	M14 Probability of unsafe state reachability M15 Deadlock absence
	Q3.3: Do PCTL queries validate fallback and degradation behavior?	M16: Path-level semantic confirmation M17: Fallback and degradation activation validity

The next sections present the results organized by goals G1–G3: first, an assessment of structural and semantic traceability (G1); second, an evaluation of the expressiveness and completeness of the extended syntax (G2); and third, a set of quantitative and qualitative PCTL verification results confirming model semantics, liveness, and safety (G3). This structure ensures that each research question is addressed with explicit supporting evidence derived from measurable transformation and verification outcomes.

For each of the case studies introduced in Chapter 5, the evaluation follows the same procedure: we extract and report the metrics specified in Tables 6.1, 6.2, and 6.3. This guarantees that all examples, regardless of their scale or modeling style, are examined under the same criteria, and that the resulting comparison highlights how the translation method behaves across different goal-modeling contexts.

In addition to the two guiding exemplars, a third synthetic model is included to exercise the full range of semantics proposed in this work. This model intentionally departs from the formal constraints and domain-specific requirements of the other case studies: it is compact, built around brittle or rapidly changing conditions, and combines multiple runtime annotations—such as maintain loops, context-dependent progress, and small retry budgets—into a single structure. Its purpose is not to represent a realistic system but to provide a goal model where all semantic constructs can be activated and observed in isolation. This allows the validation to fully probe the translation pipeline, confirming that even edge-case behaviors can be captured, translated, and verified consistently.

6.1.1 Experimentation Setup

All experiments were conducted on a MacBook Pro equipped with an Apple M2 Max processor and 32GB of RAM. For parsing and model checking the generated PRISM specifications, we employed the Storm model checker [26] rather than the native PRISM command-line interface. This choice was motivated by Storm's substantially more efficient optimization pipeline, which significantly reduced verification times—particularly for the larger and more structurally complex models analyzed in this study.

The metrics reported in the following subsections were obtained from multiple instrumentation sources. Structural and translation-related data were extracted from the logs produced by the translator and by the translation validator, a dedicated package developed for assessing and collecting information about the generated PRISM goal controllers. Probabilistic verification metrics were gathered directly from Storm's model-checking output. Together, these artifacts support a comprehensive and reproducible evaluation of the translation pipeline and the resulting controller models.

6.1.2 TAS validation

In this subsection, we apply the GQM-based validation procedure to the *TAS* case study introduced in Chapter 5. The goal is to examine how the translation pipeline behaves when confronted with a model whose execution logic is driven by time-critical assistance services and their adaptation requirements. Following the structure established earlier in this chapter, the analysis proceeds by evaluating the metrics associated with goals G1–G3 (Tables 6.1 to 6.3). For each goal, we will extract and discuss the relevant evidence from the translated PRISM model and the corresponding PCTL verification results. The subsections below outline the validation flow and indicate the tables that will constitute the final evaluation.

G1: Structural and Semantic Traceability

This part of the validation will assess whether the structural elements of the *TAS* model—including goals, tasks, refinement patterns, and runtime annotations—are accurately preserved during translation. The analysis will follow the three questions defined under G1 in Table 6.1, examining coverage, hierarchy preservation, and annotation mapping.

M1–M2: Coverage Metrics This paragraph will describe how coverage is computed by comparing the original TAS goal model elements (goals, tasks, resources) to the structures emitted in the PRISM controller. It will also explain how missing elements are detected and recorded from the validator logs.

Table 6.4: M1–M2 Coverage Metrics for the TAS Model

Node Type	**Goal Model #**	**PRISM #**	**Missing #**
Goals (modules)	22	22	0
Tasks (variables)	11	22	0
Tasks (transitions)	11	33	0
Resources (variables)	6	6	0

M3–M4: Structural Consistency Metrics This paragraph will explain how structural consistency is checked by aligning each parent goal with its children and verifying that the corresponding PRISM `pursue` actions and achievement conditions respect the intended refinement links (AND/OR/sequence/choice/degradation, etc.) in the TAS model.

Table 6.5: M3 Parent/child mapping for the TAS model

Goal (i*)	**Children (i*)**	**Children label in PRISM**	**Mapped Correctly?**
G0	G1, G2	pursue_G1, pursue_G2	✓
G1	G3, G4	pursue_G3, pursue_G4	✓
G2	G5, G6	pursue_G5, pursue_G6	✓
G3	G7	pursue_G7	✓
G4	G8, G9	pursue_G8, pursue_G9	✓
G5	T2	pursue_T2	✓
G6	T3	pursue_T3	✓
G7	T1	pursue_T1	✓
G8	G10, G11	pursue_G10, pursue_G11	✓
G9	G13, G12	pursue_G13, pursue_G12	✓
G10	T4	pursue_T4	✓
G11	G14, G15	pursue_G14, pursue_G15	✓
G12	G16, G17	pursue_G16, pursue_G17	✓
G13	G18, G19	pursue_G18, pursue_G19	✓
G14	T5	pursue_T5	✓
G15	T6	pursue_T6	✓
G16	T7	pursue_T7	✓
G17	T8	pursue_T8	✓
G18	T9	pursue_T9	✓
G19	G20, G21	pursue_G20, pursue_G21	✓
G20	T10	pursue_T10	✓

Goal (i*)	Children (i*)	Children label in PRISM	Mapped Correctly?
G21	T11	pursue_T11	✓

Table 6.6: M4 Synchronization consistency for the TAS model

Goal (i*)	i* Link	PRISM Achieve Joint	Mapped
G0	AND(G1, G2)	G1_achieved & G2_achieved_maintain	✓
G1	OR(G3, G4)	G3_achieved | G4_achieved_maintain	✓
G2	OR(G5, G6)	G5_achieved | G6_achieved	✓
G3	AND(G7)	G7_achieved	✓
G4	AND(G8, G9)	G8_achieved_maintain & G9_achieved	✓
G5	AND(T2)	T2_achieved	✓
G6	AND(T3)	T3_achieved	✓
G7	AND(T1)	T1_achieved	✓
G8	AND(G10, G11)	G10_achieved & G11_achieved	✓
G9	AND(G13, G12)	G13_achieved & G12_achieved	✓
G10	AND(T4)	T4_achieved	✓
G11	OR(G14, G15)	G14_achieved | G15_achieved	✓
G12	OR(G16, G17)	G16_achieved | G17_achieved	✓
G13	AND(G18, G19)	G18_achieved_maintain & G19_achieved	✓
G14	AND(T5)	T5_achieved	✓
G15	AND(T6)	T6_achieved	✓
G16	AND(T7)	T7_achieved	✓
G17	AND(T8)	T8_achieved	✓
G18	AND(T9)	T9_achieved	✓
G19	OR(G20, G21)	G20_achieved | G21_achieved	✓
G20	AND(T10)	T10_achieved	✓
G21	AND(T11)	T11_achieved	✓

Table 6.7: M4 Guard and context consistency for the TAS model

Node (i*)	Context Assertion	Maintain Condition	PRISM Label	PRISM Formula	Mapped
G12	`pharmacyAvailable & atHome`	–	`pursue_G12`	–	✓
G18	`medicationApplied`	`inSideEffectWindow`	`pursue_G18`	`G18_achieved_maintain`	✓
G19	`inEmergency`	–	`pursue_G19`	–	✓
G2	`patientTracking`	`highPrecision`	`pursue_G2`	`G2_achieved_maintain`	✓
G3	`privacyEnabled`	–	`pursue_G3`	–	✓
G4	`enoughBattery & highReliability`	`inEmergency`	`pursue_G4`	`G4_achieved_maintain`	✓
G5	`privacyEnabled`	–	`pursue_G5`	–	✓
G6	`privacyEnabled=false`	–	`pursue_G6`	–	✓
G8	`sensorAvailable`	`enoughBattery & highReliability`	`pursue_G8`	`G8_achieved_maintain`	✓
T10	`R4>35`	–	`pursue_T10`	–	✓
T11	`networkAvailable`	–	`pursue_T11`	–	✓
T2	`privacyEnabled`	–	`pursue_T2`	–	✓
T3	`R1>50`	–	`pursue_T3`	–	✓
T4	`R0=true`	–	`pursue_T4`	–	✓
T9	`R3=true`	–	`pursue_T9`	–	✓

Table 6.8: Annotation-mapping completeness for TAS

G2: Expressiveness and Semantic Completeness

This part will evaluate whether the enriched semantics required by *TAS* are fully expressible in the extended syntax and translated without loss. Metrics will address construct coverage (e.g., choice, sequence, interleaving, degradation, alternative), model completeness, and compilation feasibility.

M5–M6: Construct Utilization This paragraph will describe how runtime annotations in the TAS goal model are exposed in PRISM through module headers and structure, and how the validator inspects these headers to confirm that every annotated construct has a corresponding PRISM template instance.

Table 6.9: M5 Construct-to-template coverage for TAS

Name	EDGE Annotation	Goal Model #	PRISM Modules #	Matches
choice	`+`	2	2	✓
degradation	`->`	1	1	✓
sequence	`;`	4	4	✓
interleaved	`#`	1	1	✓
alternative	`|`	2	2	✓
default	–	12	12	✓

Table 6.10: M6 Lines of PRISM code per goal for TAS

Goal	Children	Annotation	Runtime Type	LoC	Mapped
G0	2	#	interleaved	12	✓
G1	2	+	choice	13	✓
G2	2	|	alternative	11	✓
G3	1	;	sequence	11	✓
G4	2	-	default	11	✓
G5	1	-	default	11	✓
G6	1	-	default	11	✓
G7	1	-	default	11	✓
G8	2	-	default	11	✓
G9	2	;	sequence	12	✓
G10	1	-	default	11	✓
G11	2	|	alternative	13	✓
G12	2	|	alternative	12	✓
G13	2	;	sequence	12	✓
G14	1	-	default	12	✓
G15	1	-	default	11	✓
G16	1	-	default	11	✓
G17	1	-	default	11	✓
G18	1	->	degradation	10	✓
G19	2	+	choice	13	✓
G20	1	-	default	11	✓
G21	1	-	default	11	✓

M7: Achievability and Maintain Formulas This paragraph will mirror the previous case studies, explaining how each TAS goal must declare an `achieved` (and, when necessary, `achieved_maintain`) formula and how the validator confirms presence and structural correctness with respect to refinement semantics.

Table 6.11: M7 Goal-level formula completeness for TAS

Goal	Type	Expected Formulas	Emitted Formulas	Matches
G0	achieve	1	1	✓
G1	achieve	1	1	✓
G2	maintain	2	2	✓
G3	achieve	1	1	✓
G4	maintain	2	2	✓
G5	achieve	1	1	✓
G6	achieve	1	1	✓
G7	achieve	1	1	✓
G8	maintain	2	2	✓
G9	achieve	1	1	✓
G10	achieve	1	1	✓
G11	achieve	1	1	✓
G12	achieve	1	1	✓
G13	achieve	1	1	✓
G14	achieve	1	1	✓
G15	achieve	1	1	✓
G16	achieve	1	1	✓
G17	achieve	1	1	✓
G18	maintain	2	2	✓
G19	achieve	1	1	✓
G20	achieve	1	1	✓
G21	achieve	1	1	✓

M8: Maintain-Goal Behavior This paragraph will describe how maintain goals in *TAS* are validated by inspecting parent `pursue` guards and ensuring that maintain goals are only triggered when their sustaining conditions are violated. A TAS-specific PRISM excerpt will be included to illustrate the guard pattern.

```
module G18
  G18_pursued : [0..1] init 0;

  [pursue_G18] G18_pursued=0 & G18_achieved_maintain=false & medicationApplied
    ↪ -> (G18_pursued'=1);
  [pursue_T9] G18_pursued=1 & G18_achieved_maintain=false & R3=true -> true;

  [achieved_G18] G18_pursued=1 & (T9_achieved=1) -> (G18_pursued'=0);

  [skip_G18] G18_pursued=1 & T9_pursued=0 -> (G18_pursued'=0);
endmodule

formula G18_achieved_maintain = inSideEffectWindow;
formula G18_achievable = T9_achievable;
```

Listing 6.1: Example maintain-goal interaction for TAS

M9–M11: Model Completeness and Compilation Feasibility This paragraph will follow the same pattern as for the other case studies, summarizing how Storm (or PRISM) is used to check compilation feasibility, generation errors, and model-size metrics for the *TAS* DTMC.

Table 6.12: M9–M11 Completeness and Compilation Feasibility Metrics (PRISM Output)

Metric	Value	Pass?
M9: PRISM Compilation Feasibility	Parsed and constructed DTMC successfully	✓
M10: Generation Errors	0 errors, 0 warnings	✓
M11: Model Size Metrics	States: 3,934; Transitions: 18,376; Labels: 2	✓

G3: Verification via PCTL

This section evaluates the behavioral semantics of the translated DTMC for *TAS* using the PCTL properties defined for the case study. As in the previous subsections, the validation is organized along metrics M12–M17, which cover liveness, safety, and semantical validation at path level. The PCTL file for *TAS* encodes reachability probabilities for key assistance goals, quantitative characterisations of unsafe scenarios, and a collection of invariants that capture refinement, choice, maintain, and degradation semantics across the goal hierarchy. Storm successfully model-checks all properties over the generated DTMC, and the resulting evidence is interpreted in light of the system-level behaviour specified in Listing 5.8.

A crucial aspect of this case study is that, although the translation pipeline automatically emits the declarations for the system variables that appear in guards, assertions, and maintain conditions, the `System` module in Listing 5.8 is *behaviourally configured* by a PRISM specialist. The specialist chooses initial values (e.g., `inEmergency=false`, `patientTracking=true`, `privacyEnabled=true`) and defines how each task label (`[achieved_Ti]`) updates context and resource variables through deterministic and probabilistic transitions. These choices are not meant as a calibrated clinical model, but as a verification-oriented environment that makes the relevant assistance scenarios and failure modes observable in the DTMC, so that the PCTL properties can exercise the semantics of alternative, degradation, and maintain goals.

Compared to the *LabSamples* and *DroneDelivery* case studies, the *TAS* model is structurally more complex at the system level: it combines numerous Boolean context flags (e.g., `inEmergency`, `highPrecision`, `inSideEffectWindow`, `networkAvailable`) with multiple bounded resources (`R1`, `R2`, `R4`, `R5`), all of which interact with guards on goals and tasks. This richness has a direct impact on the PCTL results: if initial valuations or update rules make a guard permanently unsatisfiable (for example, the current bounds `R1:`

`[0..5]` and `R4: [1..4]` combined with guards `R1>50` and `R4>35`), then the corresponding behaviours and goals become unreachable, and the probabilities of related liveness properties collapse to zero or invariants hold only vacuously. Conversely, by setting variables so that many branches are initially enabled (e.g., making `patientTracking` and `sensorAvailable` true, and providing maximum resource levels), the specialist can explore a broader portion of the state space and obtain more informative probability values for progress properties and quantitative risk measures.

```
module System
  pharmacyAvailable: bool init true;
  atHome: bool init true;
  inSideEffectWindow: bool init false;   // keep false so G18_achieved_maintain
    ↪ = false
  medicationApplied: bool init true;     // enables pursue_G18
  inEmergency: bool init false;          // so G4_achieved_maintain = false and
    ↪ pursue_G4 is enabled
  highPrecision: bool init false;        // so G2_achieved_maintain = false and
    ↪ pursue_G2 can start
  patientTracking: bool init true;       // enables pursue_G2
  privacyEnabled: bool init true;        // enables G3 and G5 branch (G6 will
    ↪ require later flip by T3)
  enoughBattery: bool init true;         // needed for pursue_G4 and
    ↪ G8_achieved_maintain
  highReliability: bool init true;       // needed for pursue_G4 and
    ↪ G8_achieved_maintain
  sensorAvailable: bool init true;       // enables pursue_G8
  networkAvailable: bool init true;      // enables pursue_T11 / G21

  R0: bool init true;                    // enables pursue_T4
  R1: [0..5] init 5;                     // max value, though guard R1>50 is
    ↪ still unsat
  R2: [0..5] init 5;                     // max mobile data
  R3: bool init true;                    // enables pursue_T9
  R4: [1..4] init 4;                     // max value, though guard R4>35 is
    ↪ still unsat
  R5: [0..10] init 10;                   // max dosage value

  // T1: Trigger alarm by pushing emergency button -> raise emergency
  [achieved_T1] true ->
    (inEmergency'=true);
```

```
25
26  // T2: Manual tracking -> enable privacy, start tracking with low precision
27  [achieved_T2] true ->
28    (privacyEnabled'=true) &
29    (patientTracking'=true) &
30    (highPrecision'=false);
31
32  // T3: Automatic tracking -> disable privacy, start tracking with high
      ↪ precision and improve R1
33  [achieved_T3] true ->
34    (privacyEnabled'=false) &
35    (patientTracking'=true) &
36    (highPrecision'=true) &
37    (R1'=min(5,R1+1));
38
39  // T4: Send sensed data -> confirm patient data available
40  [achieved_T4] true ->
41    (R0'=true);
42
43  // T5: Perform remote analysis -> tends to increase reliability (
      ↪ probabilistically)
44  [achieved_T5] true ->
45    0.9 : (highReliability'=true) +
46    0.1 : (highReliability'=false);
47
48  // T6: Perform local analysis -> may drain battery
49  [achieved_T6] true ->
50    0.7 : (enoughBattery'=true) +
51    0.3 : (enoughBattery'=false);
52
53  // T7: Apply drug change -> mark medication applied, move drug id, open side-
      ↪ effect window, ensure at home & pharmacy
54  [achieved_T7] true ->
55    (medicationApplied'=true) &
56    (pharmacyAvailable'=true) &
57    (atHome'=true) &
58    (inSideEffectWindow'=true) &
59    (R4'=min(4,R4+1));
60
```

```
61   // T8: Apply dose change -> adjust dosage, mark medication applied, open side-
       ↪ effect window
62   [achieved_T8] true ->
63     (medicationApplied'=true) &
64     (inSideEffectWindow'=true) &
65     (R5'=min(10,R5+2));
66
67   // T9: Apply side effect monitor -> consume patient data flag and close side-
       ↪ effect window
68   [achieved_T9] true ->
69     (R3'=false) &
70     (inSideEffectWindow'=false);
71
72   // T10: Enact SMS Service -> consume some mobile data and occasionally lose
       ↪ network
73   [achieved_T10] true ->
74     0.95 : (R2'=max(0,R2-1)) & (networkAvailable'=true) +
75     0.05 : (R2'=max(0,R2-1)) & (networkAvailable'=false);
76
77   // T11: Enact alarm service -> resolve emergency
78   [achieved_T11] true ->
79     (inEmergency'=false);
80 endmodule
```

Listing 6.2: TAS system-level context and resource dynamics

Overall, the *TAS* case study highlights that PCTL-based validation is highly sensitive to the configuration of the `System` module: the same automatically generated goal-and-task structure can induce quite different quantitative behaviours depending on how environment variables are initialised and updated.

The manual configuration in Listing 6.2 is therefore an integral part of the validation setup, providing a complex yet controlled environment in which metrics M12–M17 can meaningfully probe the semantics of the enriched goal model.

M12–M13: Liveness Properties The liveness fragment of the property set quantifies the probability of eventually reaching key assistance goals and intermediate progress conditions. These properties are of the form `P=? [ F ... ]` (quantitative reachability) and `P>0 [ F ... ]` (existential liveness). Table 6.13 groups the main results. For several subgoals—such as emergency support (`G1`), manual tracking (`G5`), and medicine administration (`G12`)—the model assigns strictly positive reachability, with some goals

being almost surely reachable (probability 1). Other behaviors, such as side-effect analysis (`G13`) or alarm triggering (`G19`, `G21`), remain reachable but with comparatively low probability, reflecting rare but possible execution paths.

In contrast, some additional probes (not listed in the table) highlight unreachable or structurally disabled progress conditions, such as `G0_achieved=1`, `G6_achieved=1`, and `G20_achieved=1`, whose reachability probability evaluates to 0 in the current DTMC. These zero-probability results are consistent with the fact that the corresponding labels are not instantiated in any reachable state of the constructed model.

Table 6.13: Liveness property results for TAS (M12–M13)

Liveness Query	**Explanation**	**Result**	**Satisfied?**	**Checking Time**
P=? [F (G1_achieved = 1)]	Eventually achieve emergency support (`G1`)	1.000	✓	0.355s
P=? [F (G3_achieved = 1)]	Eventually achieve self-diagnosed emergency support (`G3`)	≈ 0.623	✓	100.380s
P=? [F (G5_achieved = 1)]	Eventually achieve manual tracking (`G5`)	1.000	✓	0.401s
P=? [F ((`enoughBattery` & `highReliability`) = true)]	Eventually reach a state with enough battery and high reliability	1.000	✓	0.255s
P=? [F (G9_achieved = 1)]	Eventually achieve enactment of treatment (`G9`)	≈ 0.016	✓	100.390s
P=? [F (G12_achieved = 1)]	Eventually achieve medicine administration (`G12`)	≈ 0.732	✓	0.656s
P=? [F (G13_achieved = 1)]	Eventually complete side-effect analysis (`G13`)	≈ 0.026	✓	68.342s

Liveness Query	Explanation	Result	Satisfied?	Checking Time
P=? [F (`inSideEffectWindow` = true)]	Eventually enter the side-effect monitoring window	≈ 0.797	✓	0.352s
P=? [F (G19_achieved = 1)]	Eventually achieve the alarm-goal `G19`	≈ 0.037	✓	47.646s
P=? [F (G21_achieved = 1)]	Eventually achieve the alarm-service goal `G21`	≈ 0.055	✓	20.071s
P>0 [F (`inEmergency` = true)]	There exists a path where an emergency is eventually raised	true	✓	0.372s
P>0 [F (G2_pursued = 1)]	There exists a path where patient tracking (`G2`) is eventually pursued	true	✓	0.306s
P>0 [F (`inSideEffectWindow` = true)]	There exists a path that enters the side-effect window	true	✓	0.350s
P>0 [F (G18_pursued = 1)]	There exists a path where side-effect monitoring (`G18`) becomes active	true	✓	0.406s
P>0 [F (T11_pursued = 1 & T11_achieved = 1)]	There exists a path where the alarm task `T11` is executed successfully	true	✓	0.385s

M14–M15: Safety Properties Safety properties in the TAS case study quantify the probability of reaching explicitly encoded "unsafe" conditions and record deadlock freedom. Rather than enforcing safety as a hard requirement, these properties act as diagnostic metrics: they measure how often emergencies remain unhandled, how often network or monitoring assumptions fail, and whether problematic configurations are reachable at

all.

Table 6.14 summarizes the results. Two of the unsafe scenarios—unhandled emergencies without any alarm goal achieved, and medication applied with no side-effect monitoring active—are reachable with probability 1 from the initial state. This indicates that the current DTMC deliberately includes executions in which these undesired patterns occur. By contrast, the second unsafe pattern, which combines an emergency, absent network availability, and neither SMS nor alarm task being pursued, has probability 0 and is therefore unreachable. Finally, Storm reports no deadlock states in the model, so the DTMC is deadlock free.

Table 6.14: Safety property results for TAS (M14–M15)

Safety Query	**Explanation**	**Condition / Probability**	**Satisfied?**	**Checking Time**
P=? [F ((`inEmergency` = true & G19_achieved = 0) & G21_achieved = 0)]	Probability of ever being in emergency with neither alarm goal achieved	1.000	✓	0.315s
P=? [F (((`inEmergency` = true & `networkAvailable` = false) & T10_pursued = 0) & T11_pursued = 0)]	Probability of emergency with no network and no SMS/alarm task active	0.000	✓	0.266s
P=? [F ((`medicationApplied` = true & `inSideEffectWindow` = false) & G18_pursued = 0)]	Probability that medication is applied, the window has closed, and side-effects are never monitored	1.000	✓	0.420s
Deadlock states	Number of reachable deadlock states in the DTMC (Storm label `deadlock`)	0	✓	–

M16–M17: Semantic Validation at Path-level Properties Semantic Validation at Path-level Properties enforce the intended control-flow semantics of the TAS goal model: interleaving at the root level, alternative refinements for tracking and alarm delivery, sequencing of treatment stages, choice constraints with sticky selection variables, and

degradation behavior for emergency support. All such properties are encoded as universal PCTL invariants (`P>=1 [ G (...) ]`) or existential liveness constraints (`P>0 [ F (...) ]`) and are satisfied by the generated DTMC according to Storm.

Table 6.15 groups representative properties by construct. At the top level, `G0` is constrained so that, whenever it is achieved, both emergency support (`G1`) and high-precision tracking are in place, and its control-flow does not terminate prematurely while both subgoals remain inactive. The alternative refinement for `G2` ensures that manual and automatic tracking (`G5`, `G6`) are never pursued in parallel and that their activation respects the privacy context. Sequencing for `G8` and `G9` enforces that vital-signs reading precedes data analysis, and side-effect analysis precedes medicine administration, with `G9_achieved` only when both subgoals are achieved. The two choice goals, `G11` and `G19`, use internal variables (`G11_chosen`, `G19_chosen`) to record which variant has been selected and to prevent simultaneous pursuit of both branches. Goal–task sanity invariants guarantee that each basic goal is only achieved when its underlying task has been achieved, and that remote/local analysis tasks are never executed in parallel. Finally, degradation properties for `G1` confirm that the primary strategy `G4` is attempted while its failure counter is below the threshold, and that the fallback `G3` is activated only after sufficient failures, with at least one execution path where this degradation pattern actually occurs.

Table 6.15: Semantic validation properties for TAS (M16–M17)

Semantic Validation Query	**Explanation**	**Condition**	**Satisfied?**	**Checking Time**
P>=1 [G ((G0_achieved = 1) => (G1_achieved = 1 & `highPrecision` = true))]	Achieving `G0` implies emergency support and high-precision tracking	invariant holds	✓	0.276s
P>=1 [G (G0_pursued = 1 => !((G0_achieved = 1) & (G1_pursued = 0 & G2_pursued = 0)))]	When `G0` is active, it does not simultaneously complete and idle all children	invariant holds	✓	0.272s
P>=1 [G (G2_pursued = 1 => !((G5_pursued = 1) & (G6_pursued = 1)))]	Manual and automatic tracking are never pursued in parallel	invariant holds	✓	0.270s
P>=1 [G (G5_pursued = 1 => `privacyEnabled` = true)]	Manual tracking requires privacy to be enabled	invariant holds	✓	0.268s

Semantic Validation Query	Explanation	Condition	Satisfied?	Checking Time
P>=1 [G (G6_pursued = 1 => `privacyEnabled` = false)]	Automatic tracking requires privacy to be disabled	invariant holds	✓	0.270s
P>=1 [G (((G11_pursued = 1) ‖ (G11_achieved = 1)) => G10_achieved = 1)]	Data analysis (`G11`) only proceeds after vital-signs reading (`G10`)	invariant holds	✓	0.265s
P>=1 [G (G9_achieved = 1 => (G13_achieved = 1 & G12_achieved = 1))]	Enacting treatment (`G9`) requires both side-effect analysis and medicine administration	invariant holds	✓	0.267s
P>=1 [G !((G14_pursued = 1 & G15_pursued = 1))]	Remote and local analysis variants are never pursued in parallel	invariant holds	✓	0.266s
P>=1 [G (G14_pursued = 1 => G11_chosen = 1)]	Pursuing `G14` (remote analysis) matches choice value 1	invariant holds	✓	0.271s
P>=1 [G (G15_pursued = 1 => G11_chosen = 2)]	Pursuing `G15` (local analysis) matches choice value 2	invariant holds	✓	0.276s
P>=1 [G (((G11_pursued = 1 & G11_achieved = 0 & G11_chosen = 1) => G15_pursued = 0))]	While `G11` is active with branch 1 chosen, branch 2 is not pursued	invariant holds	✓	0.267s
P>=1 [G (((G11_pursued = 1 & G11_achieved = 0 & G11_chosen = 2) => G14_pursued = 0))]	While `G11` is active with branch 2 chosen, branch 1 is not pursued	invariant holds	✓	0.267s
P>=1 [G !((G20_pursued = 1 & G21_pursued = 1))]	SMS and alarm-mode goals (`G20`, `G21`) are never pursued in parallel	invariant holds	✓	0.274s

Semantic Validation Query	Explanation	Condition	Satisfied?	Checking Time
P>=1 [G (G20_pursued = 1 => G19_chosen = 1)]	Pursuing `G20` (SMS service) matches choice value 1	invariant holds	✓	0.275s
P>=1 [G (G21_pursued = 1 => G19_chosen = 2)]	Pursuing `G21` (alarm service) matches choice value 2	invariant holds	✓	0.263s
P>=1 [G (((G19_pursued = 1 & G19_achieved = 0 & G19_chosen = 1) => G21_pursued = 0))]	With alarm choice value 1, the other alarm branch remains inactive	invariant holds	✓	0.268s
P>=1 [G (((G19_pursued = 1 & G19_achieved = 0 & G19_chosen = 2) => G20_pursued = 0))]	With alarm choice value 2, the SMS branch remains inactive	invariant holds	✓	0.272s
P>=1 [G (G7_achieved = 1 => T1_achieved = 1)]	Achieving goal `G7` requires task `T1` to be achieved	invariant holds	✓	0.266s
P>=1 [G (G5_achieved = 1 => T2_achieved = 1)]	Achieving manual tracking `G5` requires task `T2`	invariant holds	✓	0.266s
P>=1 [G (G6_achieved = 1 => T3_achieved = 1)]	Achieving automatic tracking `G6` requires task `T3`	invariant holds	✓	0.270s
P>=1 [G (G10_achieved = 1 => T4_achieved = 1)]	Achieving `G10` requires task `T4`	invariant holds	✓	0.271s
P>=1 [G (G14_achieved = 1 => T5_achieved = 1)]	Achieving `G14` requires task `T5`	invariant holds	✓	0.268s
P>=1 [G (G15_achieved = 1 => T6_achieved = 1)]	Achieving `G15` requires task `T6`	invariant holds	✓	0.267s
P>=1 [G (G16_achieved = 1 => T7_achieved = 1)]	Achieving `G16` requires task `T7`	invariant holds	✓	0.266s
P>=1 [G (G17_achieved = 1 => T8_achieved = 1)]	Achieving `G17` requires task `T8`	invariant holds	✓	0.270s

Semantic Validation Query	Explanation	Condition	Satisfied?	Checking Time
P>=1 [G (G20_achieved = 1 => T10_achieved = 1)]	Achieving `G20` requires task `T10`	invariant holds	✓	0.270s
P>=1 [G (G21_achieved = 1 => T11_achieved = 1)]	Achieving `G21` requires task `T11`	invariant holds	✓	0.272s
P>=1 [G !((T5_pursued = 1 & T6_pursued = 1))]	Remote and local analysis tasks `T5`/`T6` are never pursued in parallel	invariant holds	✓	0.271s
P>0 [F (G1_pursued = 1 & G4_pursued = 1 & G4_failed < 2)]	There exists a path where the primary strategy `G4` is attempted while under the failure threshold	> 0	✓	0.260s
P>=1 [G (((G3_pursued = 1 & G1_pursued = 1 & G1_achieved = 0) => G4_failed $\geq$ 2))]	Fallback `G3` is only pursued once `G4` has failed enough times	invariant holds	✓	0.268s
P>0 [F (G1_pursued = 1 & G4_failed $\geq$ 2 & G3_pursued = 1)]	There exists a path where `G4` fails often enough and fallback `G3` is activated	> 0	✓	0.322s

Goal–Question–Metric Results Taken together, the GQM-based validation demonstrates that the translation pipeline satisfies all three validation goals—structural traceability (G1), semantic completeness (G2), and behavioural correctness (G3)—across all evaluated case studies. Under G1, the coverage, hierarchy, and guard-preservation metrics show that the translator maintains a faithful, lossless correspondence between the conceptual goal model and the generated PRISM controller: every goal, task, refinement link, and runtime annotation appears in the emitted specification, with parent–child relations and activation conditions preserved exactly. Under G2, the construct-to-template mapping and formula completeness metrics confirm that the enriched execution semantics of the modelling language—including sequence, interleaving, alternative choice, degradation, and maintain behaviour—are fully expressible in the extended syntax and translated without omission. The validator consistently finds one-to-one correspondences between

annotated constructs and instantiated PRISM modules, and all required achievability and maintain formulas are present and structurally consistent with the refinement semantics. Finally, the PCTL verification metrics in G3 confirm that the behavioural properties expected from the original goal models are satisfied by the DTMCs constructed from the generated controllers: liveness is preserved where intended, safety invariants hold across reachable states, and path-level correctness results show that key semantic patterns—such as ordered progression in sequences, mutual exclusion in choices, bounded fallback in degradation, and context-sensitive maintenance—manifest as intended in the executable model.

Across all cases, the GQM analysis reveals a coherent picture: the translation method not only preserves the structural and semantic content of the enriched goal models but also yields controllers whose operational behaviour matches the intended runtime semantics when interpreted through PCTL. Instances where probabilities evaluate to zero or invariants hold vacuously are attributable to deliberate modelling choices in the System module rather than to shortcomings in the translation pipeline itself. As a whole, the GQM framework provides strong evidence that the pipeline is sound, expressive, and semantically aligned with the expectations encoded in the original goal models, thereby validating the proposed approach for automated controller synthesis and probabilistic verification.

6.1.3 LabSamples validation

In this subsection, we apply the GQM-based validation procedure to the *LabSamples* case study introduced in Chapter 5. The aim is to observe how the translation pipeline behaves for a compact, well-scoped model whose execution is driven by laboratory sample handling and analysis. Following the structure established earlier in this chapter, we organize the discussion around goals G1–G3 (Tables 6.1 to 6.3). For each goal, we draw evidence from the generated PRISM controller and from the PCTL properties evaluated over the resulting DTMC. The subsections below follow the metric structure and point to the tables that contain the detailed results.

G1: Structural and Semantic Traceability

This part of the validation examines whether the structural elements of the *LabSamples* model—goals, tasks, refinement patterns, and runtime annotations in LSL—are preserved by the translation. The analysis instantiates the questions defined under G1 in Table 6.1, focusing on coverage, parent–child hierarchy, and the mapping between enriched annotations and PRISM modules.

M1–M2: Coverage Metrics After the goal controller is generated, but before the PRISM file is written, the tool performs a reverse pass over the emitted artefacts. It reconstructs an internal view of the controller and compares it against the original `GoalTree`. Any goal, task, or resource that is not found on the PRISM side is logged into a JSON report in the `/logs` folder. The coverage metrics in Table 6.16 are derived directly from this report: for *LabSamples*, the numbers of goals, task variables, task transitions, and resource variables in the PRISM model match the counts in the goal model, and no missing elements are reported.

Table 6.16: M1–M2 Coverage Metrics for the LabSamples Model

Node Type	**Goal Model #**	**PRISM #**	**Missing #**
Goals (modules)	14	14	0
Tasks (variables)	9	18	0
Tasks (transitions)	9	27	0
Resources (variables)	3	3	0

M3–M4: Structural Consistency Metrics Structural consistency is assessed by checking that the parent–child relations and refinement links in the goal model are reflected in the controller.

For M3, we manually align each goal with its declared children and inspect the corresponding `pursue_Gi` actions in the PRISM modules. Table 6.17 records this alignment: for every goal, each child in the i* model has a matching `pursue` label, and no spurious children appear in the generated controller.

For M4, we examine the joint achievement conditions associated with refinement links. For AND refinements, the `achieved` formula of the parent should combine the children's `achieved` (or `achieved_maintain`) predicates in the way prescribed by LSL; for OR refinements, the joint condition should use disjunction. Table 6.18 summarizes this check: the i* link type (e.g., `AND(G7, G8, G9)` or `OR(G3, G4)`) is compared with the corresponding conjunction or disjunction in the parent's `achieved` formula, including the use of the maintain variant where applicable (e.g., `G1_achieved_maintain` and `G11_achieved_maintain`).

In addition, guard and context consistency is inspected at the level of individual nodes. Table 6.19 relates the context assertions and maintain conditions expressed in LSL to the guards and formulas appearing in PRISM. For maintain goals, the table captures the link between the LSL maintain condition (e.g., `!sampleLoaded`) and the corresponding `achieved_maintain` formula. For tasks, it records how resource and context assertions are propagated into the `pursue` guards. This check is based directly on the generated

code and the LSL annotations, without assuming any additional semantics beyond those defined earlier in the chapter.

Table 6.17: M3 Parent/child mapping for the LabSamples model

Goal (i*)	**Children (i*)**	**Children label in PRISM**	**Mapped?**
G0	G1, G2	pursue_G1, pursue_G2	✓
G1	T1	pursue_T1	✓
G2	G13, G6	pursue_G13, pursue_G6	✓
G3	G5	pursue_G5	✓
G4	T5	pursue_T5	✓
G5	G7, G8, G9	pursue_G7, pursue_G8, pursue_G9	✓
G6	G10, G11, G12	pursue_G10, pursue_G11, pursue_G12	✓
G7	T2	pursue_T2	✓
G8	T3	pursue_T3	✓
G9	T4	pursue_T4	✓
G10	T6	pursue_T6	✓
G11	T7, T8	pursue_T7, pursue_T8	✓
G12	T9	pursue_T9	✓
G13	G3, G4	pursue_G3, pursue_G4	✓

Table 6.18: M4 Synchronization consistency for the LabSamples model

Goal (i*)	**i* Link**	**PRISM Achieve Joint**	**Mapped**
G0	`AND(G1, G2)`	`G1_achieved_maintain & G2_achieved`	✓
G1	`AND(T1)`	`T1_achieved`	✓
G2	`AND(G13, G6)`	`G13_achieved & G6_achieved`	✓
G3	`AND(G5)`	`G5_achieved`	✓
G4	`AND(T5)`	`T5_achieved`	✓
G5	`AND(G7, G8, G9)`	`G7_achieved & G8_achieved & G9_achieved`	✓
G6	`AND(G10, G11, G12)`	`G10_achieved & G11_achieved_maintain & G12_achieved`	✓
G7	`AND(T2)`	`T2_achieved`	✓
G8	`AND(T3)`	`T3_achieved`	✓
G9	`AND(T4)`	`T4_achieved`	✓
G10	`AND(T6)`	`T6_achieved`	✓
G11	`AND(T7, T8)`	`T7_achieved & T8_achieved`	✓
G12	`AND(T9)`	`T9_achieved`	✓
G13	`OR(G3, G4)`	`G3_achieved|G4_achieved`	✓

Table 6.19: M4 Guard consistency for the LabSamples model

Node (i*)	Context Assertion	Maintain Condition	PRISM Label	PRISM Formula	Mapped
G1	sampleLoaded	!sampleLoaded	pursue_G1	G1_achieved_maintain	✓
G11	sampleLoaded	analysisEnded	pursue_G11	G11_achieved_maintain	✓
T1	R0=true	–	pursue_T1	–	✓
T2	R1>=2	–	pursue_T2	–	✓
T3	R1>=1	–	pursue_T3	–	✓
T4	R1>=2&R2=true	–	pursue_T4	–	✓
T5	R2=true	–	pursue_T5	–	✓
T6	R1>=2	–	pursue_T6	–	✓
T7	sampleLoaded	–	pursue_T7	–	✓
T9	R1>=2	–	pursue_T9	–	✓

G2: Expressiveness and Semantic Completeness

This part evaluates whether the semantic extensions required by *LabSamples* (such as sequence, interleaving, and maintain behaviours) are representable in the extended syntax and whether they are translated into PRISM without dropping constructs. The metrics in G2 address construct coverage, the use of templates at the goal level, and basic model completeness.

M5–M6: Construct Utilization Once LSL annotations have been compiled into PRISM modules and guards, recovering the original construct type purely from the guards and commands would be cumbersome. To make the relationship explicit, the translator emits a header for each goal module, recording the goal identifier, its name, and the execution type derived from LSL (e.g., `sequence`, `choice`, `interleaved`). Listing 6.3 shows such a header for goal G2, which is typed as a `sequence` over children G13 and G6.

```
1  // ID: G2
2  // Name: Deliver Sample and Pickup Sample if Needed
3  // Type: sequence
4  // Relation to children: and
5  // Children: G13, G6
6  module G2
7    G2_pursued : [0..1] init 0;
8    G2_achieved : [0..1] init 0;
9
10   [pursue_G2]  G2_pursued=0 & G2_achieved=0 -> (G2_pursued'=1);
11   [pursue_G13] G2_pursued=1 & G2_achieved=0 & G13_achieved=0 -> true;
12   [pursue_G6]  G2_pursued=1 & G2_achieved=0 & G13_achieved=1 -> true;
13
```

```
    [achieved_G2] G2_pursued=1 & (G13_achieved=1 & G6_achieved=1)
        -> (G2_pursued'=0) & (G2_achieved'=1);

    [skip_G2] G2_pursued=1 & G13_pursued=0 & G6_pursued=0 -> (G2_pursued'=0);
endmodule
```

Listing 6.3: PRISM module for G2 with auxiliary header

When the PRISM validator parses a goal module, it reads this header and uses the recorded type to increment internal counters that track which templates are exercised in the model. Table 6.20 reports, for each construct kind (choice, sequence, interleaved, etc.), how many instances are expected from the LSL-annotated goal model and how many corresponding modules were emitted. For *LabSamples*, the counts match for all construct types, and no construct present in the goal model is missing from the PRISM side.

Table 6.20: M5 Construct-to-template coverage

Name	**EDGE Annotation**	**Goal Model #**	**PRISM Modules #**	**Matches**
choice	+	1	1	✓
degradation	->	0	0	✓
sequence	;	5	5	✓
interleaved	#	1	1	✓
alternative	|	0	0	✓
default	–	7	7	✓

M6 adds a more local view by relating each individual goal to its runtime type and code footprint. Table 6.21 lists, for every goal, the number of children, the annotation in the goal model, the runtime type derived at translation time, and the number of PRISM lines in the corresponding module. This view is not intended as a complexity metric but as a sanity check: for instance, sequence goals with more children tend to exhibit more `pursue` lines than basic goals, and the table lets us visually corroborate that the emitted modules follow the expected patterns for each construct.

Table 6.21: M6 Lines of PRISM code per goal

Goal	Children	Annotation	Runtime Type	LoC	Mapped
G0	2	#	interleaved	12	✓
G1	1	-	default	10	✓
G10	1	-	default	11	✓
G11	2	;	sequence	11	✓
G12	1	-	default	11	✓
G13	2	+	choice	13	✓
G2	2	;	sequence	12	✓
G3	1	;	sequence	11	✓
G4	1	-	default	11	✓
G5	3	;	sequence	13	✓
G6	3	;	sequence	13	✓
G7	1	-	default	11	✓
G8	1	-	default	11	✓
G9	1	-	default	11	✓

M7: Achievability and Maintain Formulas Every goal module must declare an `achieved` formula that captures its achievability in terms of the achievability of its children. This requirement ensures that the structural semantics of the goal model—and, in particular, the refinement operators linking children to their parent—are faithfully encoded in PRISM. Maintain goals introduce an additional obligation: they must declare a dedicated `achieved_maintain` formula, which replaces the standard `achieved` variable and expresses whether the maintain condition still holds. These two formulas together define the full lifecycle of a maintain goal, from its activation to the point where its sustaining condition is restored.

Table 6.22 summarizes the validator's output for all goals in the *LabSamples* model. The automated validator checks whether each goal includes the correct number of formulas (one for achievability, or two when maintain semantics are present) and whether all required formulas appear in the generated module. However, the validator cannot determine whether the logical content of the formula correctly reflects the refinement operator used in the goal model. For this reason, a manual verification step is required to ensure that each `achieved` formula matches the intended AND/OR/link semantics of the parent goal. This step confirms not only the presence but also the correctness of the formula structure, ensuring that the behavioral semantics of refinement are preserved in the translated DTMC.

Table 6.22: M7 Goal-level formula completeness

Goal	Type	Expected Formulas	Emitted Formulas	Matches
G1	maintain	2	2	✓
G2	achieve	1	1	✓
G3	achieve	1	1	✓
G4	achieve	1	1	✓
G5	achieve	1	1	✓
G6	achieve	1	1	✓
G7	achieve	1	1	✓
G8	achieve	1	1	✓
G9	achieve	1	1	✓
G10	achieve	1	1	✓
G11	maintain	2	2	✓
G12	achieve	1	1	✓
G13	achieve	1	1	✓

M8: Maintain-Goal Behavior The behaviour of maintain goals depends not only on their internal formulas but also on how they are triggered by their parents. The goal tree used by the translator does not maintain explicit parent context for maintain activation, so this aspect is checked manually on the PRISM side.

The check consists of reviewing the parent modules of maintain goals and confirming that `pursue` transitions for maintain children are enabled only when the maintain condition has been violated. Concretely, the parent guard should include a conjunct of the form `Gi_achieved_maintain=false` before dispatching the maintain goal, ensuring that the maintain goal is not re-activated while its condition holds. Listing 6.4 illustrates this pattern for G0 and G1: `pursue_G1` requires both that G0 is active and that `G1_achieved_maintain=false`, while the achievement of G0 depends on `G1_achieved_maintain=true`. This pattern is used as a reference when inspecting other maintain relations in the model.

```
module G0
  G0_pursued : [0..1] init 0;
  G0_achieved : [0..1] init 0;

  [pursue_G0] G0_pursued=0 & G0_achieved=0 -> (G0_pursued'=1);
  [pursue_G1] G0_pursued=1 & G0_achieved=0 & G1_achieved_maintain=false -> true;
  [pursue_G2] G0_pursued=1 & G0_achieved=0 -> true;

  [achieved_G0] G0_pursued=1 & (G1_achieved_maintain=true & G2_achieved=1) -> (
    ↪ G0_pursued'=0) & (G0_achieved'=1);

  [skip_G0] G0_pursued=1 & G1_pursued=0 & G2_pursued=0 -> (G0_pursued'=0);
endmodule
```

```
formula G0_achievable = G1_achievable * G2_achievable;
module G1
  G1_pursued : [0..1] init 0;

  [pursue_G1] G1_pursued=0 & G1_achieved_maintain=false & sampleLoaded -> (
    ↪ G1_pursued'=1);
  [pursue_T1] G1_pursued=1 & G1_achieved_maintain=false & R0=true -> true;

  [achieved_G1] G1_pursued=1 & (T1_achieved=1) -> (G1_pursued'=0);

  [skip_G1] G1_pursued=1 & T1_pursued=0 -> (G1_pursued'=0);
endmodule
formula G1_achieved_maintain = !sampleLoaded;
formula G1_achievable = T1_achievable;
```

Listing 6.4: PRISM module for G2 with auxiliary header

M9–M11: Model Completeness and Compilation Feasibility Completeness and compilation feasibility are assessed using the *Storm* model checker applied to the generated PRISM model.

M9 records whether Storm can parse the PRISM file and construct a DTMC. For the *LabSamples* controller, Storm successfully parses the model and produces a discrete-time Markov chain, indicating that the file is syntactically and semantically acceptable to the tool.

M10 summarizes the error and warning counts produced during model construction. In this case, Storm does not report missing variables, malformed updates, or incompatible declarations, and the validator detects no inconsistencies in its own checks, so the reported error and warning counts are zero.

M11 reports basic size metrics for the constructed DTMC: number of states, number of transitions, and number of labelled state sets. For *LabSamples*, Storm reports 17,521 states, 107,705 transitions, and 38 state labels, as shown in Table 6.23. These values characterize the resulting controller and provide a reference point for comparing against other case studies in later sections.

Table 6.23: M9–M11 Completeness and Compilation Feasibility Metrics (Storm Output)

Metric	Value	Pass?
M9: PRISM Compilation Feasibility	Parsed and constructed DTMC successfully	✓
M10: Generation Errors	0 errors, 0 warnings	✓
M11: Model Size Metrics	States: 17,521; Transitions: 107,705; Labels: 38	✓

G3: Verification via PCTL

This subsection evaluates the behavioural semantics of the *LabSamples* controller by analysing the PCTL properties in `labSamplesWithSideEffect.props`. Metrics M12–M17 group the properties into liveness, safety, and path-level correctness. Storm reports all properties in the file as satisfied on the DTMC constructed from the generated PRISM model.

A key aspect of this analysis is the role of the `System` module, which encodes the environment and context variables `sampleLoaded`, `analysisEnded`, `R0`, `R1`, and `R2`. These variables form part of the global state space over which the DTMC is built and are directly updated by the task completions in the `ChangeManager` module (e.g., `[achieved_T2]` decreasing `R1`, `[achieved_T4]` and `[achieved_T5]` setting `sampleLoaded=true` and `R2=false`, `[achieved_T8]` forcing `analysisEnded=true`, and `[achieved_T9]` setting `sampleLoaded=false`). Because maintain and guard conditions in the goal modules are defined in terms of these variables (for instance, `G1_achieved_maintain = !sampleLoaded` and `G11_achieved_maintain = analysisEnded`, or resource guards such as `R1>=2` for several tasks), the initial valuations and their possible updates determine which regions of the state space are actually reachable and therefore which behaviours the PCTL formulas can observe.

If the initial state of the `System` module were too restrictive, or if some of the side-effect transitions were never enabled (for example, if a task such as T9 could never be achieved, leaving `sampleLoaded=true` forever), then the DTMC would omit all states where those context changes occur. In that situation, liveness queries that depend on those contexts (such as eventual `!sampleLoaded`, low-resource states for `R1`, or `analysisEnded=true`) would either have probability 0 or would be trivially satisfied only in a degenerate part of the model, and maintain properties tied to those variables would never switch between their active and quiescent modes. Conversely, the current configuration of the `System` and `ChangeManager` modules ensures that the relevant context transitions are reachable with positive probability, so the reported PCTL results genuinely exercise the intended maintain, resource, and context-dependent behaviours of the LabSamples controller.

```
module System
  sampleLoaded: bool init true;
  analysisEnded: bool init true;
  R0: bool init true;
  R1: [0..5] init 5;
  R2: bool init true;

  [achieved_T2] true -> (R1' = max(0, R1-1));
  [achieved_T4] true -> (sampleLoaded' = true) & (R2' = false);
```

```
  [achieved_T5] true -> (sampleLoaded' = true) & (R2' = false);
  [achieved_T8] true -> (analysisEnded' = true);
  [achieved_T9] true -> (sampleLoaded' = false);
endmodule
```

Listing 6.5: System module for LabSamples

M12–M13: Liveness Properties The liveness fragment checks whether key goals and intermediate conditions are eventually reachable with non-zero probability. The queries in Table 6.24 are of the form `P=? [F ...]` or `P>0 [F...]` and range from the top-level workflow completion (`G0_achieved=1`) to the success of specific subchains (e.g., the sequential execution of G13 followed by G6 inside G2) and context changes (such as reaching `!sampleLoaded` or specific resource levels for `R1`). Storm evaluates the top-level completion probability as approximately 0.096 and all remaining `P>0` queries as `true`, indicating that the DTMC contains execution paths along which each progress condition is eventually met.

Table 6.24: Liveness property results for LabSamples (M12–M13)

Liveness Query	**Explanation**	**Result**	**Satisfied?**	**Checking Time**
P=? [F G0_achieved = 1]	Probability of eventually completing the entire workflow	≈ 0.096	✓	0.010s
P>0 [F G2_achieved = 1]	G2 (tracking) eventually becomes achievable	true	✓	0.002s
P>0 [F (G1_pursued = 1 & G2_pursued = 0)]	G1 becomes active while G2 is inactive	true	✓	0.002s
P>0 [F (G2_pursued = 1 & G1_pursued = 0)]	G2 becomes active while G1 is inactive	true	✓	0.002s
P>0 [F (G1_pursued = 1 & sampleLoaded)]	Maintain-goal G1 becomes active when sample is loaded	true	✓	0.002s
P>0 [F (!sampleLoaded)]	Context transition to unloaded sample is reachable	true	✓	0.001s

Liveness Query	Explanation	Result	Satisfied?	Checking Time
P>0 [F (G2_achieved=1 & G13_achieved=1 & G6_achieved=1)]	Full sequential chain of G2 (G13 ; G6) eventually succeeds	true	✓	0.002s
P>0 [F (G13_pursued = 1 & G3_pursued = 1)]	Choice branch G3 reachable when G13 is pursued	true	✓	0.001s
P>0 [F (G13_pursued = 1 & G4_pursued = 1)]	Choice branch G4 reachable when G13 is pursued	true	✓	0.001s
P>0 [F (G4_achieved = 1 & T5_achieved = 1)]	G4 achieves through task T5	true	✓	0.001s
P>0 [F (G10_achieved = 1)]	First element of G6's sequence eventually completes	true	✓	0.001s
P>0 [F (G6_achieved = 1 & analysisEnded)]	Completion of G6 aligns with context condition `analysisEnded`	true	✓	0.002s
P>0 [F (G12_achieved = 1 & T9_achieved = 1)]	Final sequence element G12 eventually completes through T9	true	✓	0.001s
P>0 [F (T4_achieved = 1 & sampleLoaded)]	Task T4 executes successfully under loaded-sample context	true	✓	0.001s
P>0 [F (T5_achieved = 1 & sampleLoaded)]	Task T5 succeeds under loaded-sample context	true	✓	0.001s
P>0 [F (T9_achieved = 1 & !sampleLoaded)]	Task T9 completes under opposite context	true	✓	0.001s
P>0 [F (R1 < 5)]	System can reach a reduced-resource state (R1<5)	true	✓	0.001s

Liveness Query	Explanation	Result	Satisfied?	Checking Time
P>0 [F (R1 ≤ 2)]	System can reach a minimal-resource state	true	✓	0.001s

M14–M15: Safety Properties Safety properties capture invariants and deadlock freedom. The invariants in Table 6.25 follow the shape `P>=1 [ G (...) ]` and include, for example, bounds on the resource variable `R1`, the requirement that G0's success implies the success of G2, mutual exclusion between choice branches (G3 and G4), and ordering conditions over sequential chains (e.g., G7 and G8 preceding G9, or G10 and G12 preceding G6). Storm reports these invariants as `true` in all reachable states of the DTMC.

Deadlock freedom is checked via Storm's built-in deadlock detection. For the *LabSamples* model, the label `deadlock` is reported with zero states, so the constructed DTMC contains no states without outgoing transitions under the analysed configuration.

Table 6.25: Safety property results for LabSamples (M14–M15)

Safety Query	Explanation	Condition / Prob.	Satisfied?	Checking Time
P>=1 [G (R1>=0 & R1<=5)]	Resource R1 always stays within valid bounds	true	✓	0.001s
P>=1 [G (G0_achieved=1 => G2_achieved=1)]	Root success implies success of required child G2	true	✓	0.001s
P>=1 [G !(G3_pursued=1 & G4_pursued=1)]	Mutually exclusive choice between G3 and G4	true	✓	0.001s
P>=1 [G (G13_achieved=1 => (G3_achieved=1 | G4_achieved=1))]	Achieving G13 implies one variant succeeded	true	✓	0.001s
P>=1 [G (G8_pursued=1 => G7_achieved=1)]	Sequential chain G7→G8 is preserved	true	✓	0.001s
P>=1 [G (G9_pursued=1 => (G7_achieved=1 & G8_achieved=1))]	G9 only runs after both G7 and G8 are achieved	true	✓	0.001s
P>=1 [G (G6_achieved=1 => (G10_achieved=1 & G12_achieved=1))]	G6 achievement implies all its sequential children succeeded	true	✓	0.001s
Deadlock states	Storm's deadlock detector reports reachable deadlocks	0	✓	–

M16–M17: Semantic Validation at Path-level Properties These properties refine the above checks by focusing on the detailed semantics of interleaving, sequence, choice, maintain behaviour, and resource-guarded tasks, following the LSL definitions for these constructs. Interleaving properties show that G1 and G2 can be pursued independently in different orders, as in Table 6.26, where the queries `P>0 [ F (G1_pursued=1 &`

`G2_pursued=0) ]` and `P>0 [ F (G2_pursued=1 & G1_pursued=0) ]` both evaluate to `true`. Sequence- and completion-related properties (for example, the implications relating `G6_pursued` and `G6_achieved` to the achievement of their children) show that parent goals are consistently tied to the completion of their sequential components.

Maintain-goal correctness is expressed using filter-based queries, which prevent reactivation when the maintain condition holds, and additional eventuality conditions that relate maintain flags to their corresponding context variables. For instance, the filter over `G1_pursued` and `G1_achieved_maintain` encodes that G1 is not re-pursued when `sampleLoaded` is already in the desired state.

Resource-guarded correctness uses further filter formulas to block the next-step pursuit of tasks T6, T7, and T9 under unacceptable resource or context conditions. Storm evaluates these filters as valid, meaning that under the specified state predicates, the probability of violating the guard in the next step is zero. Collectively, the properties in Table 6.26 provide a path-level view that the controller's behaviour is consistent with the sequencing, choice, maintain, and resource assertions specified in the original LSL-annotated goal model.

Table 6.26: Semantic validation properties for LabSamples (M16–M17)

Semantic Validation Query	**Explanation**	**Condition**	**Satisfied?**	**Checking Time**
$\mathbf{P}_{>0}\big[\,\mathbf{F}\,(G1_{\text{pursued}} = 1 \wedge G2_{\text{pursued}} = 0)\big]$	Interleaving allows G1 to run while G2 is inactive	> 0	✓	0.002s
$\mathbf{P}_{>0}\big[\,\mathbf{F}\,(G2_{\text{pursued}} = 1 \wedge G1_{\text{pursued}} = 0)\big]$	Interleaving allows G2 to run while G1 is inactive	> 0	✓	0.002s
$\mathbf{P}_{\geq 1}\big[\,\mathbf{G}\,(G6_{\text{pursued}} = 1 \Rightarrow G13_{\text{achieved}} = 1)\big]$	G6's sequence starts only after G13 completes	invariant holds	✓	0.001s

Semantic Validation Query	Explanation	Condition	Satisfied?	Checking Time
$\text{filter}\big(\text{forall},$ $\mathbf{P}_{\leq 0}\big[\mathbf{X}(G1_{\text{pursued}} = 1)\big],$ $G1_{\text{pursued}} = 0$ $\wedge\, G1_{\text{achieved\_maintain}} = true\big)$	Maintain goal G1 is not re-activated when its condition holds	valid	✓	0.000s
$\mathbf{P}_{\geq 1}\Big[\,\mathbf{G}\big(G6_{\text{achieved}} = 1 \Rightarrow$ $\big(G10_{\text{achieved}} = 1 \wedge$ $G12_{\text{achieved}} = 1\big)\big)\Big]$	Achieving G6 implies full sequential chain success	invariant holds	✓	0.001s
$\mathbf{P}_{>0}\big[\,\mathbf{F}\,(G6_{\text{achieved}} = 1 \wedge$ $analysisEnded)\big]$	G6 completion co-occurs with the context condition `analysisEnded`	> 0	✓	0.002s
$\mathbf{P}_{\geq 1}\big[\,\mathbf{G}\,\neg(G3_{\text{pursued}} = 1 \wedge$ $G4_{\text{pursued}} = 1)\big]$	OR-choice mutual exclusion between G3 and G4	invariant holds	✓	0.001s
$\mathbf{P}_{>0}\big[\,\mathbf{F}\,(G13_{\text{pursued}} = 1 \wedge$ $G3_{\text{pursued}} = 1)\big]$	Variant G3 is reachable from G13	> 0	✓	0.001s
$\mathbf{P}_{>0}\big[\,\mathbf{F}\,(G13_{\text{pursued}} = 1 \wedge$ $G4_{\text{pursued}} = 1)\big]$	Variant G4 is reachable from G13	> 0	✓	0.001s
$\text{filter}\big(\text{forall},\ \mathbf{P}_{\leq 0}[\,\mathbf{X}(T6_{\text{pursued}} = 1)\,],$ $T6_{\text{pursued}} = 0 \wedge T6_{\text{achieved}} = 0 \wedge$ $R1 < 2\big)$	T6 cannot be pursued when resource `R1` is below threshold	valid	✓	0.000s

Semantic Validation Query	Explanation	Condition	Satisfied?	Checking Time
filter$\big($forall, $\mathbf{P}_{\leq 0}[\,\mathbf{X}(T7_{\text{pursued}} = 1)\,]$, $T7_{\text{pursued}} = 0 \wedge T7_{\text{achieved}} = 0 \wedge \neg sampleLoaded\big)$	T7 is blocked under invalid context `!sampleLoaded`	valid	✓	0.000s
filter$\big($forall, $\mathbf{P}_{\leq 0}[\,\mathbf{X}(T9_{\text{pursued}} = 1)\,]$, $T9_{\text{pursued}} = 0 \wedge T9_{\text{achieved}} = 0 \wedge R1 < 2\big)$	T9 is blocked when resources are insufficient	valid	✓	0.000s

6.1.4 DroneDelivery validation

This subsection applies the GQM-based validation procedure to the *DroneDelivery* case study introduced in Chapter 5. The goal is to examine how the translation pipeline behaves in a scenario where autonomous package delivery must reconcile goal-driven mission logic, runtime adaptation semantics, and resource constraints. Following the structure defined earlier in this chapter, the validation is organized around the three GQM goals—G1 (traceability), G2 (expressiveness and semantic completeness), and G3 (behavioral correctness through PCTL). The corresponding metrics (M1–M17) are instantiated using evidence extracted from the translated PRISM controller and the results of Storm model checking.

The tables presented in this subsection summarize the raw measurements for each metric: coverage of structural elements (M1–M2), refinement and guard preservation (M3–M4), semantic annotation mapping (M5), line-of-code consistency (M6), formula completeness (M7), maintain-goal activation correctness (M8), compilation and model-size indicators (M9–M11), and verification outcomes for liveness, safety, and path correctness (M12–M17). In the following discussion, we analyze these results holistically without repeating the tabular content.

G1: Structural and Semantic Traceability

Under G1, we examine how faithfully the structural elements of the original *DroneDelivery* model are reflected in the generated PRISM specification. The associated questions—coverage, hierarchy preservation, link-type mapping, and annotation correctness—are evaluated through M1–M4.

M1–M2: Coverage Metrics The coverage metrics show full one-to-one correspondence between the goal model elements and their PRISM counterparts. All goals, tasks, and resources appear in the generated controller, and no omissions are detected. This indicates that the transformation pipeline not only traverses the entire goal tree but also correctly instantiates the associated variables and transitions. Such completeness is essential, as missing elements would compromise reachability reasoning and break refinement semantics in subsequent verification steps.

Table 6.27: M1–M2 Coverage Metrics for the DroneDelivery Model

Node Type	**Goal Model #**	**PRISM #**	**Missing #**
Goals (modules)	15	15	0
Tasks (variables)	12	24	0
Tasks (transitions)	12	36	0
Resources (variables)	1	1	0

M3–M4: Structural Consistency Metrics Structural alignment analysis reveals that all parent–child relations in the i* model are preserved in the PRISM pursue-graph. Every refinement—whether flat, nested, or mixed with tasks—yields the correct set of `pursue_X` labels. Similarly, the PRISM achievement conditions match the AND-refinement semantics: each composite goal becomes achievable only when all required subgoals are satisfied. The absence of mismatches across the entire structure suggests that the intermediate representation is correctly capturing dependency and refinement semantics, and that the template engine translates these semantics into deterministic control logic without erosion or reordering of relationships. Context and maintain-guard analysis further confirms that activation conditions are preserved and appropriately propagated, particularly for nodes whose behavior depends on runtime state (e.g., resource thresholds and communication constraints).

Table 6.28: M3 Parent/child mapping for the DroneDelivery model

Goal (i*)	**Children (i*)**	**Children label in PRISM**	**Mapped Correctly?**
G0	G1, G2, G3, G4, G5	pursue_G1, pursue_G2, pursue_G3, pursue_G4, pursue_G5	✓
G1	T1, T2	pursue_T1, pursue_T2	✓
G2	G6, G7	pursue_G6, pursue_G7	✓

Continues on next page

Goal (i*)	Children (i*)	Children label in PRISM	Mapped Correctly?
G3	G9, G10, G11	pursue_G9, pursue_G10, pursue_G11	✓
G4	G12, G13	pursue_G12, pursue_G13	✓
G5	G14, G15	pursue_G14, pursue_G15	✓
G6	T3	pursue_T3	✓
G7	T4	pursue_T4	✓
G9	T5	pursue_T5	✓
G10	T6	pursue_T6	✓
G11	T7	pursue_T7	✓
G12	T8	pursue_T8	✓
G13	T9	pursue_T9	✓
G14	T10, T11	pursue_T10, pursue_T11	✓
G15	T12	pursue_T12	✓

Table 6.29: M4 Synchronization consistency for the DroneDelivery model

Goal (i*)	i* Link	PRISM Achieve Joint	Mapped
G0	`AND(G1, G2, G3, G4, G5)`	`G1_achieved & G2_achieved & G3_achieved & G4_achieved & G5_achieved`	✓
G1	`AND(T1, T2)`	`T1_achieved & T2_achieved`	✓
G2	`AND(G6, G7)`	`G6_achieved & G7_achieved`	✓
G3	`AND(G9, G10, G11)`	`G9_achieved & G10_achieved & G11_achieved`	✓
G4	`AND(G12, G13)`	`G12_achieved & G13_achieved`	✓
G5	`AND(G14, G15)`	`G14_achieved & G15_achieved`	✓
G6	`AND(T3)`	`T3_achieved`	✓
G7	`AND(T4)`	`T4_achieved`	✓
G9	`AND(T5)`	`T5_achieved`	✓
G10	`AND(T6)`	`T6_achieved`	✓
G11	`AND(T7)`	`T7_achieved`	✓
G12	`AND(T8)`	`T8_achieved`	✓
G13	`AND(T9)`	`T9_achieved`	✓
G14	`AND(T10, T11)`	`T10_achieved & T11_achieved`	✓
G15	`AND(T12)`	`T12_achieved`	✓

Table 6.30: M4 Guard and context consistency for the DroneDelivery model

Node (i*)	Context Assertion	Maintain Condition	PRISM Label	PRISM Formula	Mapped
G14	`!hasMoreDelivery`	–	`pursue_G14`	–	✓
G2	`missionReady`	–	`pursue_G2`	–	✓
G7	`inFlight`	`commLink`	`pursue_G7`	`G7_achieved_maintain`	✓
T5	`R0<=3`	–	`pursue_T5`	–	✓
T6	`R0>=3`	–	`pursue_T6`	–	✓
T7	`R0>4`	–	`pursue_T7`	–	✓

G2: Expressiveness and Semantic Completeness

G2 evaluates whether the translation pipeline can express the full range of enriched runtime semantics required by *DroneDelivery*. The model uses all major constructs—choice, sequence, interleaving, and degradation—making it a suitable stress test for semantic completeness.

M5–M6: Construct Utilization Every annotated construct in the goal model results in a corresponding PRISM module structure. Choice goals produce mutually exclusive selection conditions; sequential refinements enforce ordering; interleaving refinements allow nondeterministic alternation between branches; and degradation goals introduce bounded retries and fallback transitions. The mapping between annotation counts and generated PRISM modules is exact. The diversity of constructs in the case study demonstrates that the extended syntax is expressive enough to capture complex mission logic, including non-uniform execution patterns and multi-modal adaptation behaviors. The consistency in lines-of-code per goal also reflects that the template instantiation process scales proportionally to refinement complexity and maintains predictable structure across goals.

```
// ID: G5
// Name: Standard Landing Delivery
// Type: alternative
// Relation to children: or
// Children: G14, G15
module G5
  G5_pursued : [0..1] init 0;
  G5_achieved : [0..1] init 0;

  [pursue_G5] G5_pursued=0 & G5_achieved=0 -> (G5_pursued'=1);
  [pursue_G14] G5_pursued=1 & G5_achieved=0 & G15_pursued=0 -> true;
  [pursue_G15] G5_pursued=1 & G5_achieved=0 & G14_pursued=0 -> true;

```

```
14   [achieved_G5] G5_pursued=1 & (G14_achieved=1 | G15_achieved=1) -> (G5_pursued'
       ↪ =0) & (G5_achieved'=1);
15
16   [skip_G5] G5_pursued=1 & G14_pursued=0 & G15_pursued=0 -> (G5_pursued'=0);
17 endmodule
18
19 formula G5_achievable = G14_achievable + G15_achievable - (G14_achievable *
       ↪ G15_achievable);
```

Listing 6.6: Example PRISM module for a DroneDelivery goal with auxiliary header

Table 6.31: M5 Construct-to-template coverage for DroneDelivery

Name	EDGE Annotation	Goal Model #	PRISM Modules #	Matches
choice	+	1	1	✓
degradation	->	1	1	✓
sequence	;	2	2	✓
interleaved	#	2	2	✓
alternative	|	1	1	✓
default	–	8	8	✓

Table 6.32: M6 Lines of PRISM code per goal for DroneDelivery

Goal	Children	Annotation	Runtime Type	LoC	Mapped
G0	5	#	interleaved	15	✓
G1	2	|	alternative	12	✓
G2	2	;	sequence	12	✓
G3	3	+	choice	14	✓
G4	2	->	degradation	13	✓
G5	2	#	interleaved	12	✓
G6	1	-	default	11	✓
G7	1	-	default	10	✓
G9	1	-	default	11	✓
G10	1	-	default	11	✓
G11	1	-	default	11	✓
G12	1	-	default	11	✓
G13	1	-	default	11	✓
G14	2	;	sequence	12	✓
G15	1	-	default	11	✓

M7: Achievability and Maintain Formulas All goals expose the correct number of formulas based on their type. Achieve-goals declare exactly one achievability formula, while maintain-goals declare both the maintain invariant and the achievability expression. No discrepancies are observed in formula content or naming. This confirms that refinement structure (e.g., AND, OR, single-child propagation) is faithfully encoded in the probability

expressions, and that maintain semantics are systematically exposed as reusable formulas outside the module. The uniform correctness of the formulas indicates that the expression generator handles structural recursion robustly and that no semantic branches of the original specification are lost.

Table 6.33: M7 Goal-level formula completeness

Goal	Type	Expected Formulas	Emitted Formulas	Matches
G1	maintain	1	1	✓
G2	achieve	1	1	✓
G3	achieve	1	1	✓
G4	achieve	1	1	✓
G5	achieve	1	1	✓
G6	achieve	1	1	✓
G7	achieve	2	2	✓
G9	achieve	1	1	✓
G10	achieve	1	1	✓
G11	maintain	1	1	✓
G12	achieve	1	1	✓
G13	achieve	1	1	✓
G14	achieve	1	1	✓
G15	achieve	1	1	✓

M8: Maintain-Goal Behavior Maintain-goal behavior is validated by inspecting the parent `pursue` guards and confirming that activation only occurs when the maintain condition is violated. The PRISM modules follow the expected structure: maintain goals are pursued only when their associated invariant evaluates to false, and parent achievement conditions require the maintain invariant to hold. The example involving communication maintenance (`commLink`) illustrates this interplay clearly: the maintain formula influences both pursuit and achievement transitions, yielding a stable, semantically coherent runtime loop. This demonstrates that the pipeline can encode persistent conditions that need continuous monitoring—an essential feature in autonomous systems with safety and communication constraints.

```
module G2
  G2_pursued : [0..1] init 0;
  G2_achieved : [0..1] init 0;

  [pursue_G2] G2_pursued=0 & G2_achieved=0 & missionReady -> (G2_pursued'=1);
  [pursue_G6] G2_pursued=1 & G2_achieved=0 -> true;
  [pursue_G7] G2_pursued=1 & G2_achieved=0 & G7_achieved_maintain=false -> true;

  [achieved_G2] G2_pursued=1 & (G6_achieved=1 & G7_achieved_maintain=true) -> (
    ↪ G2_pursued'=0) & (G2_achieved'=1);
```

```
10
11    [skip_G2] G2_pursued=1 & G6_pursued=0 & G7_pursued=0 -> (G2_pursued'=0);
12  endmodule
13  formula G2_achievable = G6_achievable * G7_achievable;
14  module G7
15    G7_pursued : [0..1] init 0;
16
17    [pursue_G7] G7_pursued=0 & G7_achieved_maintain=false & inFlight -> (
        ↪ G7_pursued'=1);
18    [pursue_T4] G7_pursued=1 & G7_achieved_maintain=false -> true;
19
20    [achieved_G7] G7_pursued=1 & (T4_achieved=1) -> (G7_pursued'=0);
21
22    [skip_G7] G7_pursued=1 & T4_pursued=0 -> (G7_pursued'=0);
23  endmodule
24
25  formula G7_achieved_maintain = commLink;
26  formula G7_achievable = T4_achievable;
```

Listing 6.7: Example maintain-goal interaction for DroneDelivery

M9–M11: Model Completeness and Compilation Feasibility The PRISM/Storm toolchain compiles the generated DTMC without errors, suggesting that the template engine does not introduce syntactic or structural inconsistencies. The model size—over 135k states and nearly one million transitions—reflects the combinatorial branching induced by interleaving and choice structures. Despite this scale, the absence of deadlocks and errors demonstrates that the produced controller is both semantically complete and operationally coherent. The successful construction of the DTMC is a strong indication that the transformation pipeline handles complex compositions without generating unreachable or ill-formed control paths.

Table 6.34: M9–M11 Completeness and Compilation Feasibility Metrics for DroneDelivery

Metric	Value	Pass?
M9: PRISM Compilation Feasibility	Parsed and constructed DTMC successfully	✓
M10: Generation Errors	0 errors, 0 warnings	✓
M11: Model Size Metrics	States: 135,201; Transitions: 963,598; Labels: 42	✓

G3: Verification via PCTL

This subsection evaluates the behavioural semantics of the *DroneDelivery* controller by analysing the PCTL properties defined in its verification file. Metrics M12–M17, as in the previous case study, group the properties into liveness, safety, and path-level correctness. All properties are checked over the DTMC induced jointly by the goal modules, the task-level `ChangeManager`, and the environment dynamics captured by the `System` module shown in Listing 6.8.

```
module System
  hasMoreDelivery: bool init false;
  missionReady: bool init false;
  commLink: bool init false;
  inFlight: bool init false;
  R0: [0..5] init 5;

  // Mission and delivery context
  [achieved_T1] true -> (missionReady' = true);
  [achieved_T4] true -> 0.3: (hasMoreDelivery' = true)
                        + 0.7: (hasMoreDelivery' = false);

  // Battery consumption
  [achieved_T3] true -> (inFlight' = true)
                        & (R0' = max(0, R0-1));
  [achieved_T7] true -> (R0' = max(0, R0-2));

  // Communication link dynamics
  [achieved_T4] true -> (commLink' = true);
  [achieved_T7] true -> 0.6: (commLink' = false)
                        + 0.4: (commLink' = commLink);
  [achieved_T6] true -> 0.3: (commLink' = false)
                        + 0.7: (commLink' = commLink);
endmodule
```

Listing 6.8: System module for DroneDelivery

A crucial aspect of this part of the evaluation is that the `System` module is *not* fully generated by the translation pipeline. While the translator does automatically emit the *structure* of the module—namely, the declarations of the system-level variables that appear in guards and context assertions—the *behavioural setup* of these variables is intentionally left to the PRISM specialist. This includes assigning initial values, defining

probabilistic effects, and specifying how tasks update environmental or resource conditions. These behavioural choices are domain- and analysis-dependent, meaning that the specialist must craft transition rules that provide meaningful dynamics for PCTL model checking. In other words, the translation ensures that all required variables exist and are properly typed, but their operational semantics must be authored and refined manually so that the DTMC exhibits the necessary variability to exercise and validate the semantics of the goal model.

The initial values and probabilistic effects chosen for variables such as `missionReady`, `commLink`, `inFlight`, `hasMoreDelivery`, and the battery resource `R0` are therefore not domain truths—they are modelling choices crafted to allow the exploration of the system's behavioural semantics. Their purpose is to ensure that all branches, guards, maintain conditions, and degradation paths in the goal model become reachable in the DTMC, enabling the verification process to evaluate the correctness of each semantic construct.

M12–M13: Liveness Properties All liveness properties hold with positive probability, confirming that the controller can always progress toward mission completion. The model allows transitions into all intended operational branches—including multiple landing strategies, degraded safety modes, and different energy profiles—demonstrating that no execution path has been inadvertently restricted by the translation. In addition, resource-dependent states such as battery conditions are reachable, validating the correct integration of contextual variables and runtime guards.

Table 6.35: Liveness property results for DroneDelivery (M12–M13)

Liveness Query	**Explanation**	**Result**	**Satisfied?**	**Checking Time**
P=? [F G0_achieved=1]	Probability that the entire delivery mission eventually completes	≈ 0.335	✓	0.018s
P=? [F (inFlight & commLink)]	Drone eventually reaches a stable in-flight state with communication link active	1.000	✓	0.015s
P>0 [F (R0<5)]	Battery consumption occurs along some execution path	true	✓	0.014s

Liveness Query	Explanation	Result	Satisfied?	Checking Time
P>0 [F (R0 ≤ 3)]	Low-battery states (enabling restricted profiles) are reachable	true	✓	0.015s
P>0 [F hasMoreDelivery]	Additional-delivery scenarios are reachable	true	✓	0.015s
P>0 [F (G3_pursued=1 & G9_pursued=1)]	Choice branch G9 (energy-saving profile) is reachable	true	✓	0.014s
P>0 [F (G3_pursued=1 & G10_pursued=1)]	Choice branch G10 (balanced profile) is reachable	true	✓	0.014s
P>0 [F (G3_pursued=1 & G11_pursued=1)]	Choice branch G11 (aggressive profile) is reachable	true	✓	0.014s
P>0 [F (G4_achieved=1 & G12_achieved=1 & G13_achieved=0)]	Safety envelope (G4) can succeed without degradation	true	✓	0.015s
P>0 [F (G4_achieved=1 & G13_achieved=1)]	Degraded mode (G13) can also achieve G4	true	✓	0.016s
P>0 [F (G5_achieved=1 & G14_achieved=1)]	Landing success via G14 is reachable	true	✓	0.018s
P>0 [F (G5_achieved=1 & G15_achieved=1)]	Landing success via G15 is reachable	true	✓	0.019s

M14–M15: Safety Properties Safety invariants are universally satisfied. No deadlocks are detected, and all mandatory preconditions for sequential or guarded behaviors hold across all reachable states. Of particular significance is the correctness of mutual exclusion in choice refinements: the model checker confirms that no two mutually exclusive profiles (e.g., G9, G10, G11) can be active simultaneously. This validates not only the guard logic but also the structural soundness of the pursue-conditions derived from choice semantics. Resource invariants behave as expected, ensuring that tasks bound to specific battery levels never activate outside their allowed domain.

Table 6.36: Safety property results for DroneDelivery (M14–M15)

Safety Query	**Explanation**	**Condition / Probability**	**Satisfied?**	**Checking Time**
P>=1 [G (R0>=0 & R0 ≤ 5)]	Battery resource always remains within bounds	invariant holds	✓	0.014s
P>=1 [G (G2_pursued=1 => G1_achieved=1)]	Sequential activation: G2 requires completion of G1	invariant holds	✓	0.014s
P>=1 [G (G3_pursued=1 => (G1_achieved=1 & G2_achieved=1))]	G3 only activates after G1, G2	invariant holds	✓	0.014s
P>=1 [G !((G9_pursued=1 & G10_pursued=1) | (G9_pursued=1 & G11_pursued=1) | (G10_pursued=1 & G11_pursued=1))]	Choice variants for G3 never run in parallel	invariant holds	✓	0.014s
P>=1 [G (T5_pursued=1 => R0 ≤ 3)]	Resource guard for T5 respected	invariant holds	✓	0.014s
P>=1 [G (T7_pursued=1 => R0 > 4)]	Guard for aggressive profile respected	invariant holds	✓	0.015s
P>=1 [G (G7_pursued=1 => inFlight)]	Maintain communication only while drone is airborne	invariant holds	✓	0.015s
Deadlock states	Storm reports no reachable deadlocks	0	✓	–

M16–M17: Semantic Validation at Path-level Properties These properties verify adherence to the runtime semantics of the enriched notation. All such properties are satisfied: sequential goals enforce ordering, interleaving goals permit nondeterministic alternation, choice goals remain mutually exclusive, and degradation paths require the correct number of failures before activation. Maintain-goal correctness also holds, confirming that maintain invariants, when violated, trigger appropriate corrective behavior. These results indicate that the translation pipeline successfully preserves the full operational semantics of the extended goal model, even when those semantics involve subtle runtime conditions and multi-branch control flows.

Table 6.37: Semantic validation properties for DroneDelivery (M16–M17)

Semantic Validation Query	**Explanation**	**Condition**	**Satisfied?**	**Checking Time**
P>=1 [G (G4_pursued=1 => (G1_achieved=1 & G2_achieved=1 & G3_achieved=1))]	Later subgoals require all prior ones	invariant holds	✓	0.014s
P>=1 [G (T11_pursued=1 => T10_achieved=1)]	Sequential safety for G14 (T10 ; T11)	invariant holds	✓	0.014s
P>0 [F (G1_pursued=1 & T1_pursued=1 & T2_pursued=0)]	Interleaving: T1 may run before T2	> 0	✓	0.014s
P>0 [F (G1_pursued=1 & T2_pursued=1 & T1_pursued=0)]	Interleaving: T2 may run before T1	> 0	✓	0.014s
P>0 [F (G2_pursued=1 & G6_pursued=1 & G7_pursued=0)]	G6 may run before G7	> 0	✓	0.013s
P>0 [F (G2_pursued=1 & G7_pursued=1 & G6_pursued=0)]	G7 may run before G6	> 0	✓	0.013s
P>=1 [G !((G9_pursued=1 & G10_pursued=1) | (G9_pursued=1 & G11_pursued=1) | (G10_pursued=1 & G11_pursued=1))]	Choice mutual exclusion enforced	invariant holds	✓	0.014s
P>=1 [G (G13_pursued=1 => G12_failed>=2)]	G4 degrades only after two failures	invariant holds	✓	0.015s
P>0 [F (G4_achieved=1 & G12_achieved=1 & G13_achieved=0)]	Non-degraded success path reachable	> 0	✓	0.015s
P>0 [F (G4_achieved=1 & G13_achieved=1)]	Degraded success path reachable	> 0	✓	0.016s
P>=1 [G ((G7_achieved_maintain & commLink) | (!G7_achieved_maintain & !commLink))]	Maintain-goal state aligns with commLink	invariant holds	✓	0.015s
filter(forall, P>0 [F (commLink | !inFlight)], inFlight & !commLink)	If communication fails in flight, recovery or exit is possible	valid	✓	0.020s

Overall, the GQM-based validation for *DroneDelivery* demonstrates that the proposed translation methodology achieves structural fidelity, semantic validation, and behavioral correctness across a diverse and operationally rich autonomous system model. The combined evidence from structural metrics, template mapping, and PCTL verification confirms that the generated PRISM controller accurately reflects the intent and requirements encoded in the enriched goal model.

6.2 Performance metrics

This section evaluates the performance of the proposed translation and verification workflow. Our analysis focuses on how the size of a goal model—measured in terms of its total number of nodes—influences the computational resources required at each stage of the toolchain. Specifically, we study (i) the time and memory consumed during translation from EDGE to PRISM, (ii) the cost of parsing, constructing, and model checking the resulting DTMCs, and (iii) the size of the generated PRISM artifacts. Together, these metrics provide a quantitative assessment of the responsiveness, and practical feasibility of applying automated controller generation and probabilistic verification in iterative modelling workflows.

6.2.1 EDGE to PRISM Controller Translation

The first part of the evaluation examines the performance of the translation pipeline that converts an EDGE goal model into a PRISM goal-controller specification. Since translation is invoked repeatedly during model refinement, responsiveness is critical: designers must be able to iteratively modify goal structures, regenerate controllers, and re-evaluate behavioral properties with minimal delay.

In the evaluation presented in the next subsections, a variety of goal models with different sizes and structural characteristics were analyzed. In Table 6.38, the labels **S** and **Effect** correspond to the TAS and LSL case-study systems, respectively, while all remaining entries represent systematic variations of the drone-delivery toy model used to assess specific semantic constructs. For transparency and reproducibility, all models used in the evaluation are publicly available at `https://github.com/vieirin/goal-controller/tree/1080c5c/examples`. Detailed instructions for running the experiments, including prerequisites and execution steps, are provided in the project README: `https://github.com/vieirin/goal-controller?tab=readme-ov-file#running-experiments`.

Table 6.38: Model Labels, Rationales, and Node Counts

Original Name	**Label**	**Nodes**	**Model Rationale**
1-minimal	minimal	6	Variation of model 10: Basic minimal model with only essential goals and tasks, no advanced features.
2-OrVariation	Variation	11	Variation of model 10: Adds OR/alternative goal variations to demonstrate choice semantics.
3-interleavedPaltPseq	Pseq	16	Variation of model 10: Adds interleaved, parallel, and sequential goal composition constructs.
4-interleavedChoicePDegradation	Degradation	22	Variation of model 10: Adds interleaved, choice, and degradation goal types.
5-allAnnotations	Annotations	27	Variation of model 10: Includes all annotation types (assertions, QoS properties, etc.).
6-allnotationsReduced	Reduced	27	Variation of model 10: All annotations included but in a reduced/simplified form.
7-minimalAll	All	24	Variation of model 10: Minimal structure with all basic goal composition features.
8-minimalMaintain	Maintain	27	Variation of model 10: Adds maintain goal types for goal preservation semantics.
9-minimalMaintainContext	Context	27	Variation of model 10: Adds maintain goals and context variables for conditional goal execution.

Original Name	Label	Nodes	Model Rationale
10-minimalMaintainResource	Resource	31	Full delivery drone model with maintain goals, context variables, and resource management (battery resource with consumption/production).
goalModel_TAS_3	S	40	TAS (TeleAssistance System) model: Healthcare support system with emergency services, privacy features, and life support automation.
labSamplesWithSideEffect	Effect	31	LSL (Lab Sample Logistics) system: Sample delivery system with side effects, monitoring goals, and sample tracking.

For each of these models we measure three key aspects of translation behavior: (i) translation time, capturing how long it takes for EDGE2PRISM to emit a complete PRISM model; (ii) memory usage during translation, reflecting the runtime footprint of the TypeScript/Node.js-based translator; and (iii) the size of the generated PRISM file, which indicates how model structure scales into concrete controller code.

The subsections that follow present the results for each of these metrics. These findings illuminate how structural properties of the goal model—including node count, decomposition patterns, and the introduction of system-level variables—shape the cost of translation and the characteristics of the resulting controller.

6.2.1.1 Translation time

Figure 6.1 plots the total number of nodes in each goal model against the time taken by *EDGE2PRISM* to generate the corresponding PRISM controller. Across all experiments, translation completes within **milliseconds**, even for the larger models. While the fitted trend line shows a mild upward tendency, this dataset is too small to draw strong conclusions about scalability; larger models would be needed for a clearer picture.

It is also evident that translation time is influenced not only by node count but by model structure and the number of variables introduced. For example, the clusters *[Context, Reduced, Maintain]* and *[Resource, Effect]* contain models with similar sizes but different generation times due to their structural and variable complexity. These variations suggest that structural richness can affect translation cost, though overall the process remains fast enough to support highly interactive modelling workflows.

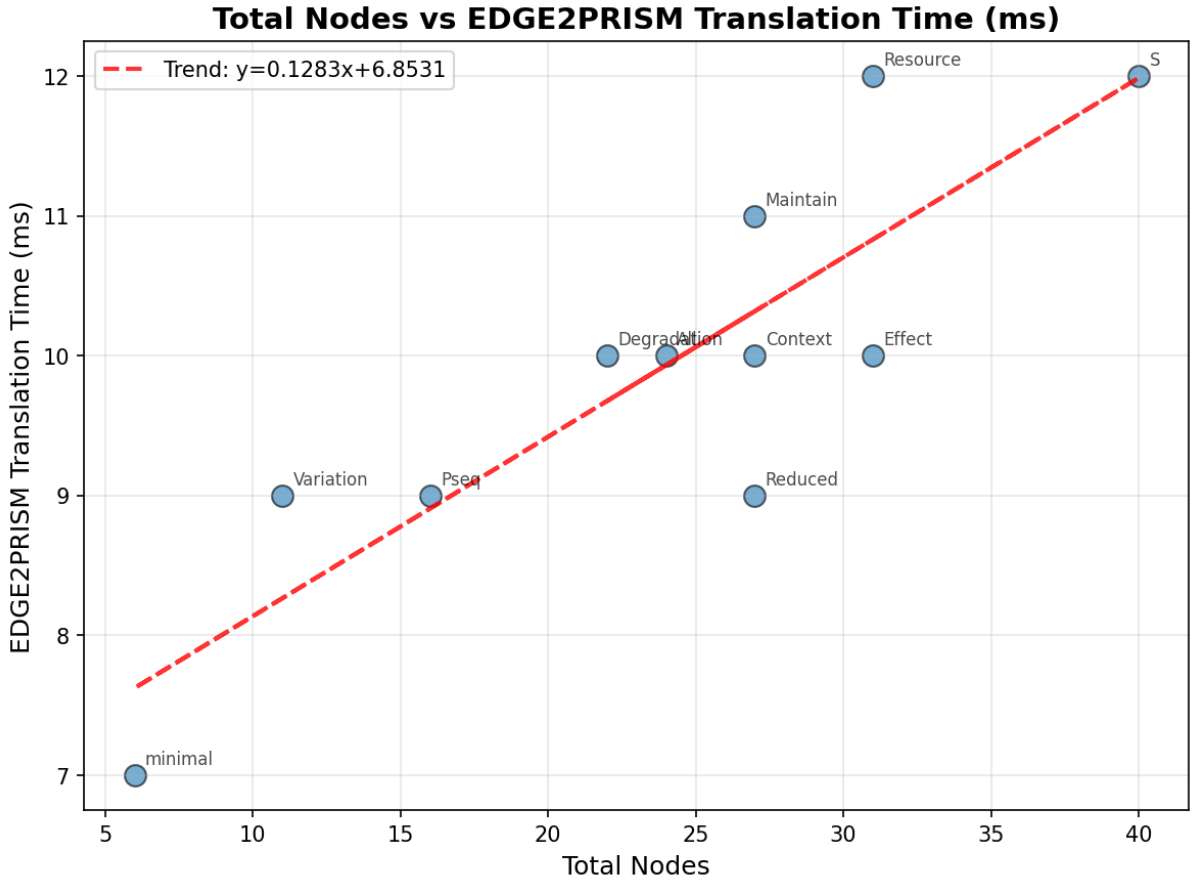


Figure 6.1: Total Nodes vs. EDGE2PRISM Translation Time (ms)

Within the scope of this validation, however, no clear tendency for translation time to increase with model size was observed. Instead, translation times remain tightly clustered across all configurations, indicating that the translation behaves efficiently and appears relatively insensitive to the structural complexity of the model. This responsiveness makes the process suitable for **interactive modelling workflows**, where designers iteratively adjust goal structures, regenerate controllers, and re-run analyses with minimal delay.

6.2.1.2 Memory Usage

Figure 6.2 reports the peak memory consumption observed during the translation of each goal model. Across the ten evaluated models, no clear tendency toward increased memory usage is visible. Instead, the values fluctuate within a narrow band of approximately 360–380 MB. This behavior is consistent with expectations for a TypeScript/Node.js development environment, where process-level memory allocation is dominated by the runtime and associated libraries rather than by per-model scaling. Since the tested implementation is a non-production version and memory overhead is largely constant, these results indicate that *EDGE2PRISM* operates comfortably within typical development-time resource budgets, with no observable memory growth attributable to model size in this evaluation.

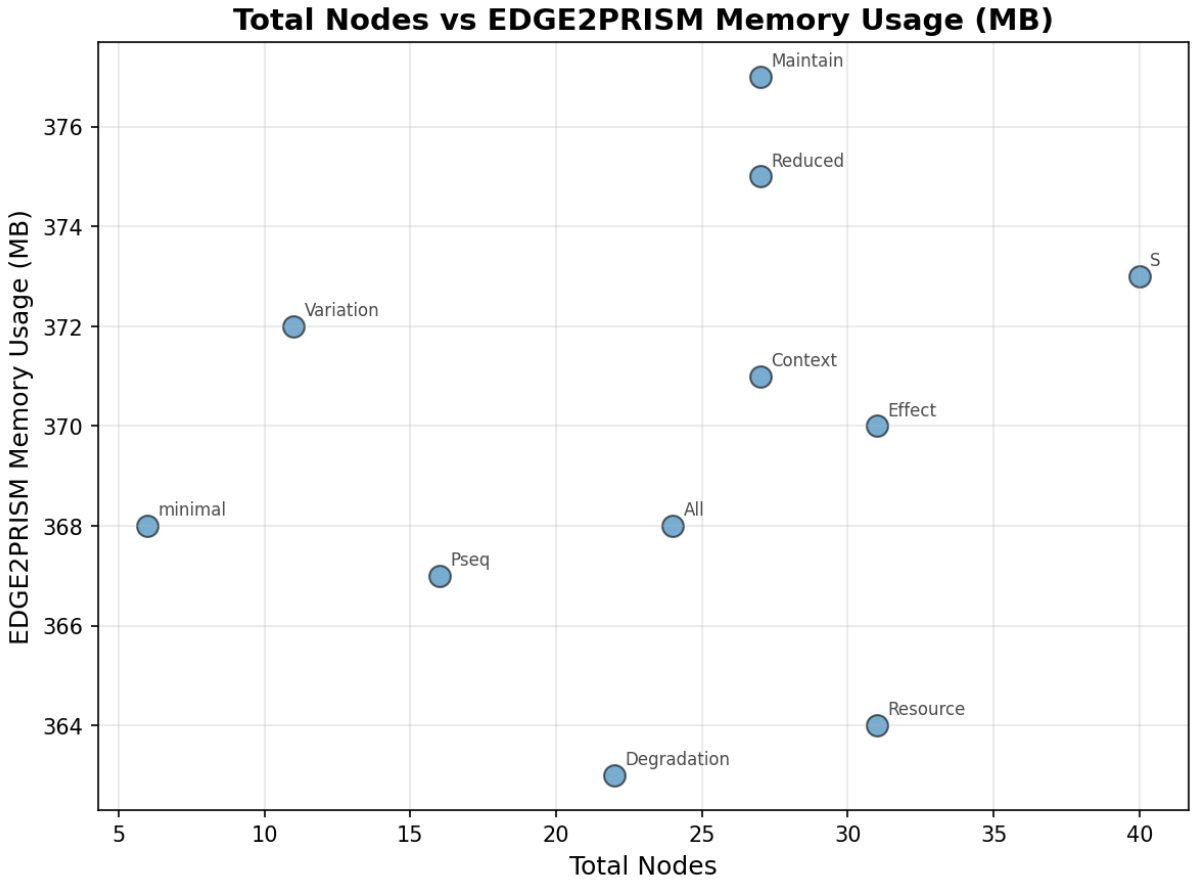


Figure 6.2: Total Nodes vs. *EDGE2PRISM* Memory Usage (MB).

6.2.1.3 Model Size

Figure 6.3 examines the relationship between the number of nodes in each goal model and the resulting size of the generated PRISM file, measured in lines of code. The plot reveals a clear near-linear trend: as the number of goals and tasks increases, the emitted PRISM

model grows proportionally. This behavior is expected, since each node contributes a relatively fixed set of module declarations, guards, and update commands, and therefore expands the file size by a consistent amount.

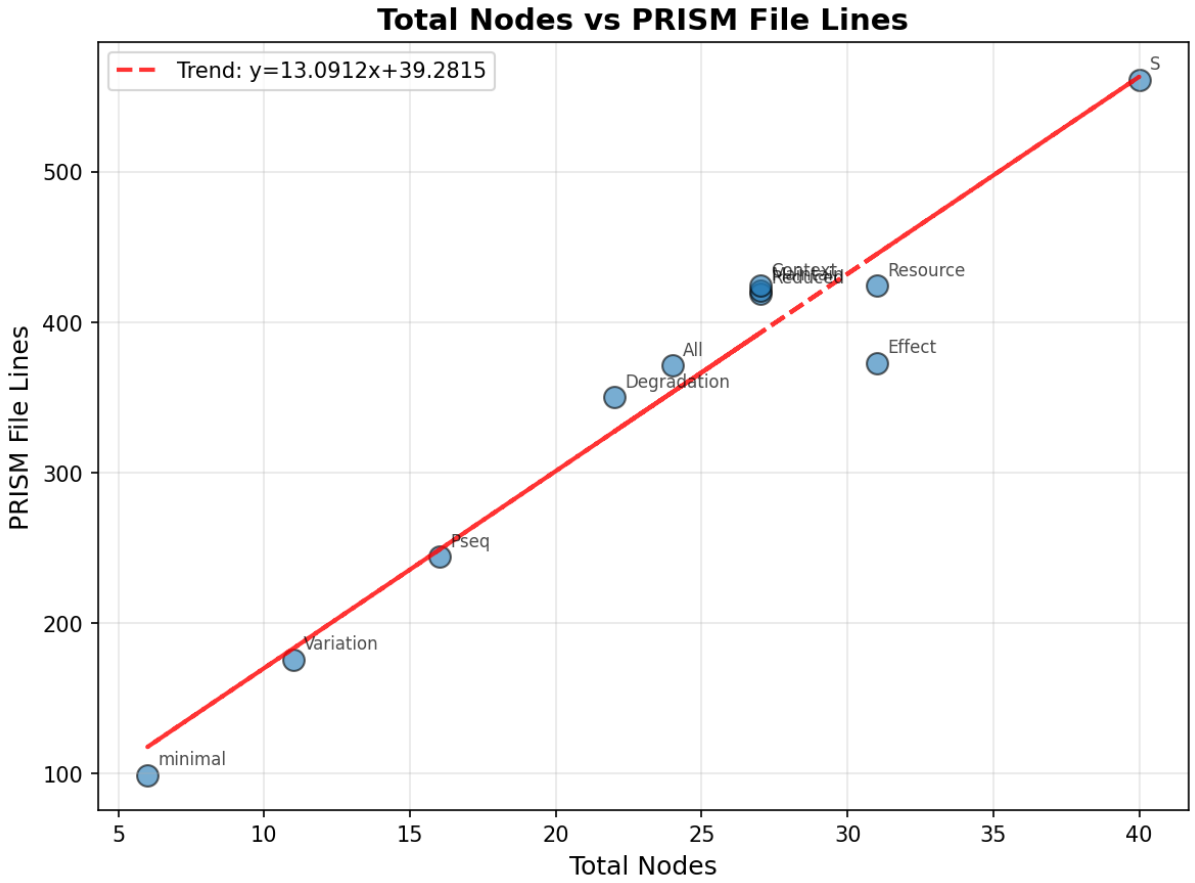


Figure 6.3: Total Nodes vs. PRISM File Lines.

While the fitted trend line closely matches the empirical data, it is important to note that the available dataset is relatively small. The current experiments cover models of modest size, and although the linear relation is strong within this range, a more comprehensive assessment would require additional models with higher node counts and more varied structures. Nonetheless, the results so far indicate that EDGE2PRISM generates PRISM controllers whose file size scales predictably with the conceptual model, reinforcing the traceability and structural regularity of the translation process.

6.2.2 PRISM Model Checking metrics

In this section, we analyse the computational resources required to model check the generated PRISM models for each experiment. During validation, the PRISM model checker encountered difficulties when computing the state space for several of the larger models.

To ensure completeness, all quantitative PRISM metrics were therefore evaluated using Storm, a modern PRISM-compatible model checker that successfully handled the full set of experiments.

The subsections that follow examine how the number of nodes in each goal model influences three key resource metrics: PRISM parsing time, memory consumption, total CPU time for model check and model-construction time. Together, these results provide insight into the computational profile of executing probabilistic analysis over automatically generated goal-controller DTMCs.

6.2.2.1 PRISM Parsing Time

Figure 6.4 relates the total number of nodes in each goal model to the time that PRISM spends parsing the generated model before construction. Parsing times range from roughly 2 to 10 ms, remaining in the millisecond range even for the largest models in this dataset. The fitted trend line indicates a moderate positive tendency, but the number of samples is too small to claim a general scalability result; a more robust assessment would require models with many more nodes. As with translation time, parsing cost does not depend only on node count: structural complexity and the number of declared variables also play a role. This is visible, for example, in the cluster *[Reduced, Maintain, Context]*, where models with comparable node counts exhibit slightly different parsing times because they introduce different variable sets and module structures.

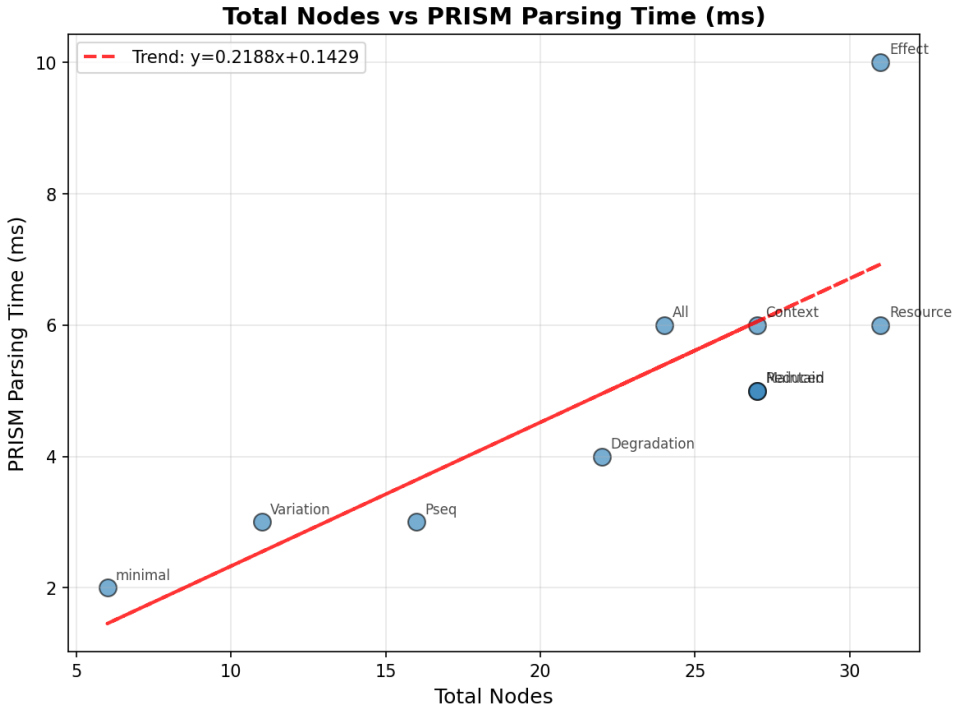


Figure 6.4: Total nodes vs. PRISM parsing time (ms)

6.2.2.2 PRISM Peak Memory

Figure 6.5 reports the peak memory consumption observed during model checking for each translated PRISM model. For most goal models, the peak usage remains relatively stable—typically near 80–200 MB—and does not show a systematic growth trend across the tested range of model sizes. This is consistent with the behaviour of modern symbolic model checkers, where memory usage often depends more on the structure of the state space than on the raw number of nodes in the goal model.

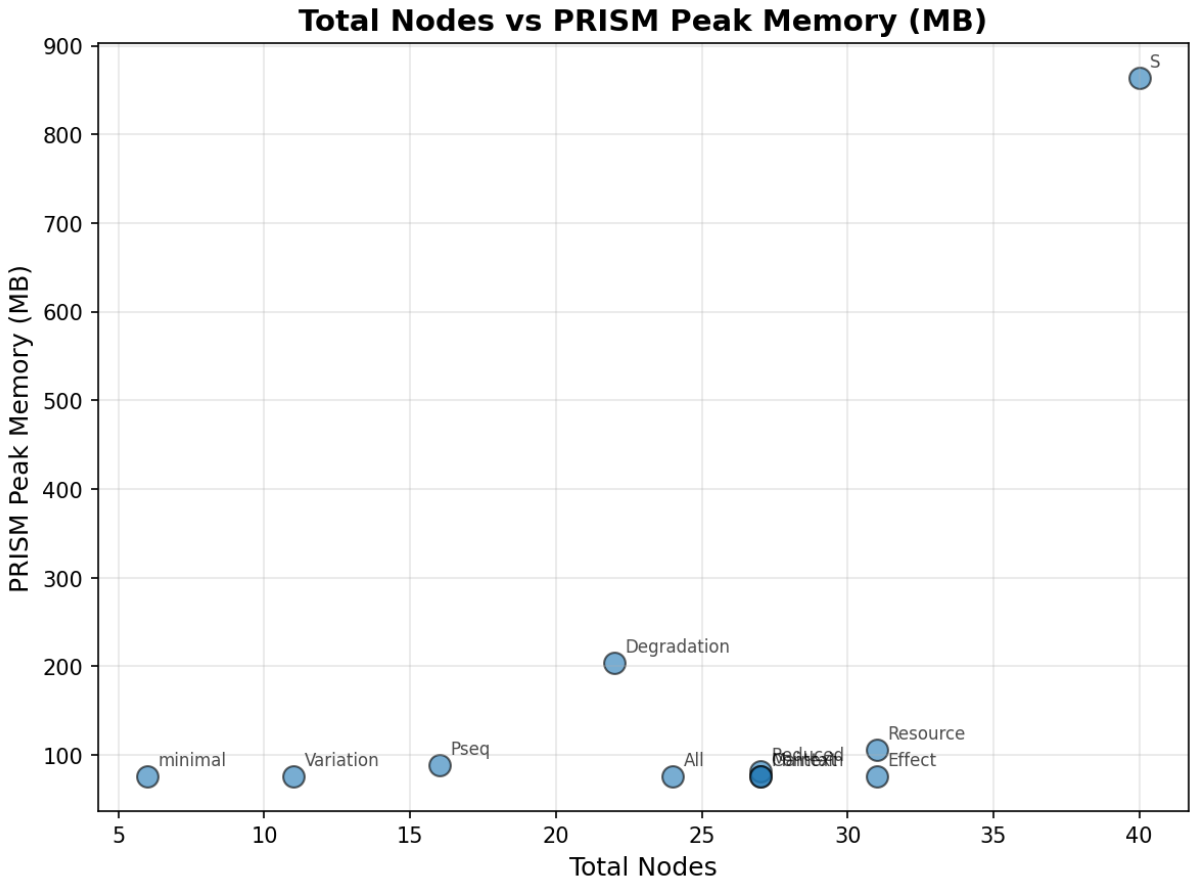


Figure 6.5: Total Nodes vs. PRISM Peak Memory Usage (MB).

A notable outlier is the *TAS* case study, which consumes over 800 MB. This spike arises from the significantly more complex `System` module used in that model: it declares a larger set of variables and introduces richer environment dynamics, both of which increase the size of the resulting DTMC and therefore the memory footprint of the checker. Apart from this structurally complex model, the memory usage across the remaining examples remains moderate and well within the expected bounds for PRISM-compatible workflows.

6.2.2.3 PRISM CPU Time

Figure 6.6 presents the CPU time required by PRISM to verify each translated model, shown on a logarithmic scale to accommodate the large variation observed across the dataset. The results reveal a clear separation between structurally simple models and those with richer environmental dynamics: lightweight models with few variables and minimal interaction between modules complete in just a few milliseconds, while more elaborate configurations can take several orders of magnitude longer.

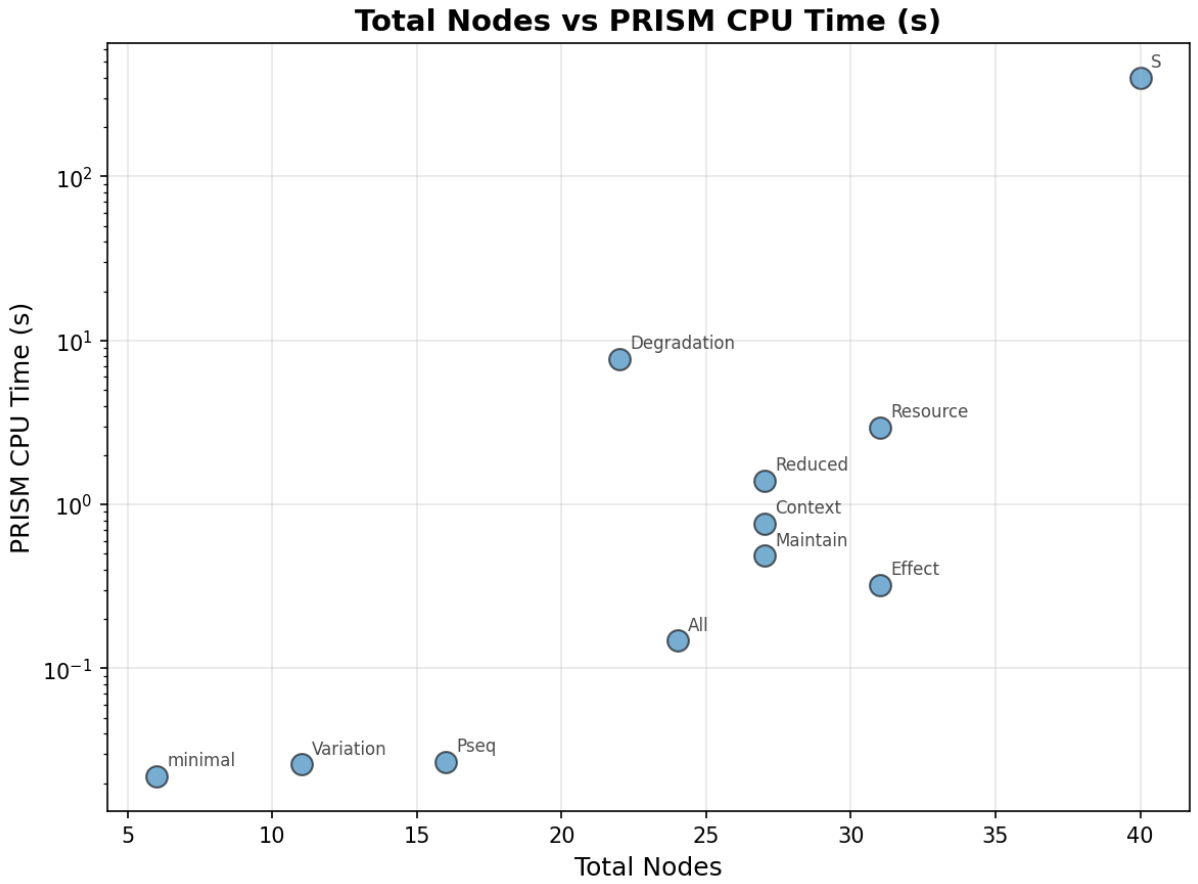


Figure 6.6: Total Nodes vs. PRISM CPU Time (s). Logarithmic scale.

This difference is particularly evident in models such as *Degradation*, *Resource*, and especially the *TAS* case study, the latter exceeding 200 s of CPU time due to its complex `System` module and large set of state-dependent variables. In contrast, examples like *minimal*, *Variation*, or *Pseq* finish almost instantaneously despite their similar node counts. This again illustrates that CPU time is driven less by the number of goals and tasks, and more by the semantics encoded in the system-level state space, allowing structurally compact but behaviourally rich models to dominate verification cost.

6.2.2.4 PRISM Model Construction Time

Figure 6.7 reports the time required by PRISM to *construct* the underlying DTMC before model checking takes place, plotted on a logarithmic scale. The pattern closely mirrors the CPU-time results discussed in the previous subsection: lightweight models with few variables and simple system dynamics are constructed almost instantly (on the order of 10^{-2} s), while models with richer behavioural structure require progressively more time.

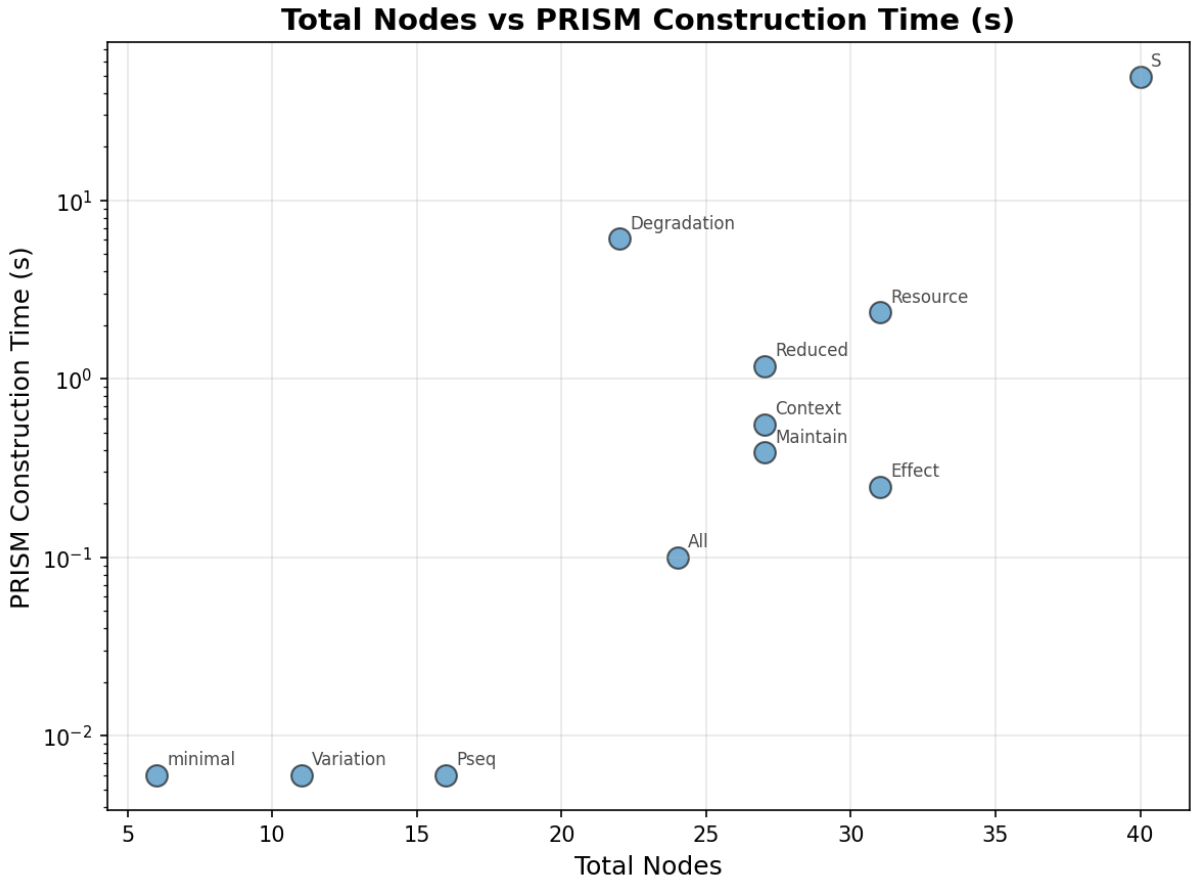


Figure 6.7: Total Nodes vs. PRISM Model Construction Time (s). Logarithmic scale.

A clear correspondence emerges when comparing this chart with the CPU-time results: models that take longer to construct also take significantly longer to query. This near 1:1 relation suggests that construction time is dominated by the same factor that dominates overall CPU cost—namely, the size and complexity of the state space induced by the `System` module and its variables. As before, the *TAS* case study is an outlier, requiring over 30 s to construct due to its rich environment dynamics and high variable count, whereas compact examples like *minimal*, *Variation*, and *Pseq* complete almost immediately.

Overall, these results reinforce the conclusion that verification cost is driven less by the number of goals or tasks, and more by the semantic richness encoded in the system-level state space.

PRISM Results Across all four performance dimensions—parsing time, peak memory usage, CPU time, and model-construction time—the results demonstrate that the computational cost of model checking the automatically generated DTMCs is influenced far more by the behavioral richness encoded in the `System` module than by the number of goals or tasks in the conceptual model. Models with simple context dynamics and few variables remain lightweight throughout the toolchain, completing parsing, construction, and verification in milliseconds. Conversely, models whose `System` module introduces complex state evolution or large variable sets exhibit substantially higher memory demand and longer verification times, even when their goal-structure size is comparable.

Taken together, these results highlight an important distinction: while the EDGE2PRISM translation process itself scales smoothly with model size, the subsequent model-checking cost reflects the semantic complexity of the underlying domain assumptions rather than the size of the goal model alone. This underscores the role of the `System` module as the primary driver of verification complexity and provides a clear roadmap for future optimisation efforts—particularly for highly expressive or variable-rich environments such as the *TAS* case study. Despite these variations, all models remained analyzable, and the overall workflow proved robust when paired with a modern PRISM-compatible checker such as Storm.

6.3 Threats to Validity

The evaluation of the EDGE2PRISM translation pipeline is based on a combination of structural analysis, semantic inspection, and probabilistic model checking. As with any empirical study, the conclusions drawn from these results are subject to several limitations. To clarify the scope of the findings, this section discusses potential threats to validity across four dimensions: construct validity, internal validity, external validity, and reliability. Each dimension reflects a different way in which the evidence gathered during validation may fall short of fully capturing the behavior or generality of the proposed approach.

6.3.1 Construct Validity

In this work, the constructs of interest include structural preservation, semantic validation, behavioral correctness, and computational performance. Although the GQM framework

provides a disciplined way of operationalizing these constructs, several limitations remain.

A first limitation stems from the fact that structural metrics such as coverage (M1–M2) and refinement alignment (M3–M4) verify the presence and shape of elements in the PRISM controller but cannot fully guarantee semantic equivalence. They confirm, for example, that an AND-refinement is translated into a conjunction of `achieved` predicates, but they do not validate deeper behavioural properties unless reinforced by PCTL queries.

A second issue arises from the role of the `System` module. While the translator generates all variable declarations needed to support guards and annotations, the dynamics of these variables—initial values, probabilistic effects, and side-effect rules—are manually authored. As a result, the behaviour observed in the DTMC reflects both the translation and the modelling choices made by the specialist, making it difficult to attribute effects solely to the translation pipeline.

6.3.2 Internal Validity

The most significant threat arises from the manual definition of system-level behaviour. Even small changes in the `System` module—such as different initial values, resource bounds, or transition probabilities—can dramatically alter the reachability of tasks, the activation of maintain or degradation loops, and the outcomes of PCTL queries. This makes it difficult to disentangle the semantics introduced by the translator from those introduced by the environment configuration.

Another source of threat is the use of two different verification engines. Several of the larger models cannot be constructed by PRISM but are successfully analysed by Storm. Storm's symbolic methods, memory strategies, and solver heuristics may behave differently from PRISM's engines.

A further internal validity concern is that the validator consumes auxiliary annotations generated by the translation pipeline itself. This improves traceability but reduces independence: errors in template generation could in principle propagate to validation without being detected unless contradicted by PCTL behaviour or manual inspection.

6.3.3 External Validity

The primary limitation is the small number of case studies. Although *LabSamples*, *DroneDelivery*, and *TAS* cover different styles of modelling, they cannot represent the full diversity of goal-oriented specifications found in real systems.

Scalability results also generalise only cautiously. The performance experiments include models up to several dozen nodes, but it remains unclear how the translation or

verification costs behave for models with hundreds of goals, multiple interacting maintain loops, or extensive degradation chains.

The behavior of the translated controllers also depends on the way domain dynamics are encoded in the `System` module. The transition rules used in the case studies are intentionally designed to keep semantic constructs observable for validation. Real-world system behavior may be more restrictive or nonlinear, potentially leading to different reachability patterns and different validation outcomes.

6.3.4 Reliability

The core transformation from EDGE to PRISM is deterministic: given the same goal model, the translator produces identical PRISM modules, and all intermediate steps are logged. The GQM metrics and validator outputs are also automatically generated, which supports reproducibility.

However, reliability is weakened by the manually authored parts of the system-level behavior. Reproducing the exact DTMC requires replicating all initial values, probabilistic effects, and state updates used in each case study. Even small deviations in these parameters can yield different probabilities, reachability conditions, or semantic activation patterns, especially for rare behaviors such as degradation transitions or context-dependent maintain loops.

Performance measures such as parsing time, peak memory, or CPU usage also depend strongly on hardware, PRISM/Storm implementation details, and runtime conditions. Replicating the exact numerical results therefore requires controlled execution environments or containerization.

Despite these limitations, all artifacts—goal models, generated PRISM code, property files, and logs—are sufficient for independent replication of the structural and semantic aspects of the evaluation.

Chapter 7

Conclusion and Future Work

This thesis presented a translation methodology for the automated generation of *EDGE*-compliant goal controllers in PRISM from enriched ExtenDed Goal Models. The central contribution lies in establishing a precise and operational mapping between the modelling constructs of EDGE and their behavioural counterparts in PRISM, enabling controllers to be generated in a systematic, verifiable, and semantically faithful manner. The work combines conceptual foundations, an intermediate representation, a full translation pipeline, and a comprehensive validation strategy grounded in the Goal–Question–Metric (GQM) framework.

Across this dissertation, three main contributions were developed—each grounded in the research questions introduced in Section 1.2

- **Contribution 1 — Structural fidelity (RQ1).** Formal mapping of EDGE constructs to PRISM modules and variables (developed in Chapter 3; instantiated through the case models in Chapter 4).
- **Contribution 2 — Semantic expressiveness (RQ2).** A language-agnostic Intermediate Representation capturing enriched EDGE semantics (specified in Chapter 3; implemented in Chapter 5).
- **Contribution 3 — Behavioural verification under PCTL (RQ3).** Validation of synthesized controllers using liveness, safety, and reachability properties (evaluated in Chapter 6).

Overall, this methodology has been implemented as a modular toolchain that has successfully produced EDGE goal controllers in study, supporting both academic experimentation and internal research workflows. By reducing the manual effort previously required for PRISM model construction and ensuring a documented and traceable mapping between constructs, the approach enhances reproducibility, scalability, and methodological clarity in the design of goal-based probabilistic controllers.

7.1 Future Work

Several avenues offer promising extensions to the work presented in this thesis:

Scalability Analysis. Although the translation pipeline does not impose inherent size limitations, probabilistic models may grow rapidly as new modules, guards, and context variables are included. A systematic exploration of scalability—including memory pressure, state-space explosion, and computational constraints—would provide deeper insights into the practical limits of EDGE-based controller synthesis.

Formalisation of the EDGE–PRISM Model. A formal specification of the EDGE–PRISM semantics would allow correctness to be proven at the model level, ensuring that every generated controller satisfies well-formedness, type consistency, and behavioural soundness.

Decision Variables and Parametric Analysis. Introducing explicit decision variables could enable richer classes of DTMC and MDP models, allowing analysts to explore parameter ranges, optimise behaviour, or perform sensitivity analysis beyond Boolean contexts alone.

Enhanced and Formal Validator. A more powerful validator—capable of reconstructing the PRISM model in memory and performing deeper static analysis—would support more rigorous semantic validation and reduce reliance on manual inspection.

Achievability, Robustness, and Sensitivity Evaluation. Future studies could analyse how structural patterns, refinement strategies, or context assumptions influence goal achievability and robustness. This would connect EDGE translation with design-time reasoning about system reliability and performance.

Appendices

Appendix A

Pursue Lines Generation Algorithm

Algorithm 7 `pursueStatements(goal)`: Generate PRISM pursue transitions through a multi-stage pipeline

```
1:  procedure pursueStatements(goal)
2:      goalsToPursue ← [goal] ∪ childrenIncludingTasks(goal)
3:      statements ← [ ]
                                  ▷ Stage 1: Initialize base left and right statements
4:      for all child in goalsToPursue do
5:          isItself ← (child.id == goal.id)
6:          left ← [pursue_child.id] & pursued(goal.id) = (0 if isItself else 1)
7:          if goal.execCondition.maintain then
8:              left ← left & achievedMaintain(goal.id) = false
9:          else
10:             left ← left & achieved(goal.id) = 0
11:         end if
12:         if isItself then
13:             left ← left & goalDependencyStatement(goal)
14:             right ← (pursued(goal.id)'=1)
15:         else
16:             right ← true
17:         end if
18:         statements ← statements ∪ [[child, {left, right}]]
19:     end for
                                  ▷ Stage 2: Add execution detail conditions
20:     for all [child, {left, right}] in statements do
21:         if not (child.id == goal.id) then
22:             if goal.relationToChildren == 'or' then
```

```
23:                 type ← goal.executionDetail.type
24:                 if type == 'choice' then
25:                     children ← childrenIncludingTasks(goal)
26:                     pursueCondition ← pursueChoiceGoal(goal, children, child.id)
27:                     left ← left & pursueCondition
28:                     if child.index > 0 then
29:                         right ← (chosenVariable(goal.id)'=child.index)
30:                     end if
31:                 else if type == 'degradation' then
32:                     pursueCondition ← pursueDegradationGoal(goal,
    goal.executionDetail.degradationList, child.id)
33:                     if pursueCondition ≠ "" then
34:                         left ← left & pursueCondition
35:                     end if
36:                 else if type == 'alternative' then
37:                     pursueCondition ← pursueAlternativeGoal(goal, child.id)
38:                     left ← left & pursueCondition
39:                 end if
40:             else if goal.relationToChildren == 'and' then
41:                 if goal.executionDetail.type == 'sequence' then
42:                     children ← childrenIncludingTasks(goal)
43:                     pursueCondition ← pursueAndSequentialGoal(goal,
    goal.executionDetail.sequence, child.id, children)
44:                     left ← left & pursueCondition
45:                 end if
46:             end if
47:         end if
48:     end for
                                    ▷ Stage 3: Add activation and maintain context guards
49:     for all [child, {left, right}] in statements do
50:         activationContext ← ""
51:         if (child.id == goal.id or child.type == 'task') and child.execCondition then
52:             if child.execCondition.assertion.sentence then
53:                 activationContext ← child.execCondition.assertion.sentence
54:             end if
55:         end if
56:         maintainContext ← ""
```

```
57:         if not (child.id == goal.id) and child.execCondition.maintain then
58:             if child.execCondition.maintain.sentence then
59:                 maintainContext ← achievedMaintain(child.id) = false
60:             end if
61:         end if
62:         if activationContext ≠ "" or maintainContext ≠ "" then
63:             guards ← [activationContext, maintainContext]
64:             guards ← FILTEREMPTY(guards)
65:             left ← left & JOIN(guards, " & ")
66:         end if
67:     end for
                                                ▷ Stage 4: Add failed counter updates
68:     for all [child, {left, right}] in statements do
69:         if not (child.id == goal.id) and child.properties.maxRetries then
70:             updateFailed ← (failed(child.id)'=min(maxRetries,
    failed(child.id)+1))
71:             if right == "true" then
72:                 right ← updateFailed
73:             else
74:                 right ← right & updateFailed
75:             end if
76:         end if
77:     end for
                                                ▷ Stage 5: Remove repeated conditions
78:     for all [child, {left, right}] in statements do
79:         left ← REMOVEREPEATEDCONDITIONS(left)
80:         right ← REMOVEREPEATEDCONDITIONS(right)
81:     end for
                                                ▷ Stage 6: Format as PRISM transition strings
82:     pursueLines ← [ ]
83:     for all [child, {left, right}] in statements do
84:         transition ← left -> right;
85:         pursueLines ← pursueLines ∪ [transition]
86:     end for
87:     return pursueLines
88: end procedure
89: procedure GOALDEPENDENCYSTATEMENT(goal)
```

```
90:     if goal.dependsOn == null or LENGTH(goal.dependsOn) == 0 then
91:         return ""
92:     end if
93:     conditions ← [ ]
94:     for all dep in goal.dependsOn do
95:         condition ← HASBEENACHIEVED(dep, {condition: true})
96:         conditions ← conditions ∪ [condition]
97:     end for
98:     joinedConditions ← JOIN(conditions, " & ")
99:     return " & (" + joinedConditions + ")"
100: end procedure
101: procedure REMOVEREPEATEDCONDITIONS(condition)
102:     parts ← SPLIT(condition, " & ")
103:     uniqueParts ← [ ]
104:     for all part in parts do
105:         if part ∉ uniqueParts then
106:             uniqueParts ← uniqueParts ∪ [part]
107:         end if
108:     end for
109:     return JOIN(uniqueParts, " & ")
110: end procedure
```

Appendix B

PRISM Goal Controller for TAS

```
dtmc
// ID: G0
// Name: Provide Health Support
// Type: interleaved
// Relation to children: and
// Children: G1, G2
module G0
  G0_pursued : [0..1] init 0;
  G0_achieved : [0..1] init 0;

  [pursue_G0] G0_pursued=0 & G0_achieved=0 -> (G0_pursued'=1);
  [pursue_G1] G0_pursued=1 & G0_achieved=0 -> true;
  [pursue_G2] G0_pursued=1 & G0_achieved=0 & G2_achieved_maintain=false -> true;

  [achieved_G0] G0_pursued=1 & (G1_achieved=1 & G2_achieved_maintain=true) -> (
    ↪ G0_pursued'=0) & (G0_achieved'=1);

  [skip_G0] G0_pursued=1 & G1_pursued=0 & G2_pursued=0 -> (G0_pursued'=0);
endmodule

formula G0_achievable = G1_achievable * G2_achievable;

// ID: G1
// Name: Support in emergency
// Type: degradation
// Relation to children: or
// Children: G3, G4
module G1
```

```
  G1_pursued : [0..1] init 0;
  G1_achieved : [0..1] init 0;
  G4_failed : [0..10] init 0;

  [pursue_G1] G1_pursued=0 & G1_achieved=0 -> (G1_pursued'=1);
  [pursue_G3] G1_pursued=1 & G1_achieved=0 & G4_failed >= 2 -> true;
  [pursue_G4] G1_pursued=1 & G1_achieved=0 & G4_achieved_maintain=false -> (
    ↪ G4_failed'=min(10, G4_failed+1));

  [achieved_G1] G1_pursued=1 & (G3_achieved=1 | G4_achieved_maintain=true) -> (
    ↪ G1_pursued'=0) & (G1_achieved'=1);

  [skip_G1] G1_pursued=1 & G3_pursued=0 & G4_pursued=0 -> (G1_pursued'=0);
endmodule

formula G1_achievable = G3_achievable + G4_achievable - (G3_achievable *
    ↪ G4_achievable);

// ID: G2
// Name: Track patient location
// Type: alternative
// Relation to children: or
// Children: G5, G6
module G2
  G2_pursued : [0..1] init 0;

  [pursue_G2] G2_pursued=0 & G2_achieved_maintain=false & patientTracking -> (
    ↪ G2_pursued'=1);
  [pursue_G5] G2_pursued=1 & G2_achieved_maintain=false & G6_pursued=0 -> true;
  [pursue_G6] G2_pursued=1 & G2_achieved_maintain=false & G5_pursued=0 -> true;

  [achieved_G2] G2_pursued=1 & (G5_achieved=1 | G6_achieved=1) -> (G2_pursued'
    ↪ =0);

  [skip_G2] G2_pursued=1 & G5_pursued=0 & G6_pursued=0 -> (G2_pursued'=0);
endmodule

formula G2_achieved_maintain = highPrecision;
formula G2_achievable = G5_achievable + G6_achievable - (G5_achievable *
    ↪ G6_achievable);
```

```
62
63 // ID: G3
64 // Name: Self-diagnosed emergency support service
65 // Type: basic
66 // Relation to children: and
67 // Children: G7
68 module G3
69   G3_pursued : [0..1] init 0;
70   G3_achieved : [0..1] init 0;
71
72   [pursue_G3] G3_pursued=0 & G3_achieved=0 & privacyEnabled -> (G3_pursued'=1);
73   [pursue_G7] G3_pursued=1 & G3_achieved=0 -> true;
74
75   [achieved_G3] G3_pursued=1 & (G7_achieved=1) -> (G3_pursued'=0) & (
       ↪ G3_achieved'=1);
76
77   [skip_G3] G3_pursued=1 & G7_pursued=0 -> (G3_pursued'=0);
78 endmodule
79
80 formula G3_achievable = G7_achievable;
81
82 // ID: G4
83 // Name: Automated life support service
84 // Type: sequence
85 // Relation to children: and
86 // Children: G8, G9
87 module G4
88   G4_pursued : [0..1] init 0;
89
90   [pursue_G4] G4_pursued=0 & G4_achieved_maintain=false & enoughBattery &
       ↪ highReliability -> (G4_pursued'=1);
91   [pursue_G8] G4_pursued=1 & G4_achieved_maintain=false & G8_achieved_maintain=
       ↪ false -> true;
92   [pursue_G9] G4_pursued=1 & G4_achieved_maintain=false & G8_achieved_maintain=
       ↪ true -> true;
93
94   [achieved_G4] G4_pursued=1 & (G8_achieved_maintain=true & G9_achieved=1) -> (
       ↪ G4_pursued'=0);
95
96   [skip_G4] G4_pursued=1 & G8_pursued=0 & G9_pursued=0 -> (G4_pursued'=0);
```

```
97 endmodule
98
99 formula G4_achieved_maintain = inEmergency;
100 formula G4_achievable = G8_achievable * G9_achievable;
101
102 // ID: G5
103 // Name: Manual Tracking
104 // Type: basic
105 // Relation to children: and
106 // Children: T2
107 module G5
108   G5_pursued : [0..1] init 0;
109   G5_achieved : [0..1] init 0;
110
111   [pursue_G5] G5_pursued=0 & G5_achieved=0 & privacyEnabled -> (G5_pursued'=1);
112   [pursue_T2] G5_pursued=1 & G5_achieved=0 & privacyEnabled -> true;
113
114   [achieved_G5] G5_pursued=1 & (T2_achieved=1) -> (G5_pursued'=0) & (
      ↪ G5_achieved'=1);
115
116   [skip_G5] G5_pursued=1 & T2_pursued=0 -> (G5_pursued'=0);
117 endmodule
118
119 formula G5_achievable = T2_achievable;
120
121 // ID: G6
122 // Name: Automatic Tracking
123 // Type: basic
124 // Relation to children: and
125 // Children: T3
126 module G6
127   G6_pursued : [0..1] init 0;
128   G6_achieved : [0..1] init 0;
129
130   [pursue_G6] G6_pursued=0 & G6_achieved=0 & privacyEnabled=false -> (
      ↪ G6_pursued'=1);
131   [pursue_T3] G6_pursued=1 & G6_achieved=0 & R1>50 -> true;
132
133   [achieved_G6] G6_pursued=1 & (T3_achieved=1) -> (G6_pursued'=0) & (
      ↪ G6_achieved'=1);
```

```
134
135   [skip_G6] G6_pursued=1 & T3_pursued=0 -> (G6_pursued'=0);
136 endmodule
137
138 formula G6_achievable = T3_achievable;
139
140 // ID: G7
141 // Name: Allow push button
142 // Type: basic
143 // Relation to children: and
144 // Children: T1
145 module G7
146   G7_pursued : [0..1] init 0;
147   G7_achieved : [0..1] init 0;
148
149   [pursue_G7] G7_pursued=0 & G7_achieved=0 -> (G7_pursued'=1);
150   [pursue_T1] G7_pursued=1 & G7_achieved=0 -> true;
151
152   [achieved_G7] G7_pursued=1 & (T1_achieved=1) -> (G7_pursued'=0) & (
      ↪ G7_achieved'=1);
153
154   [skip_G7] G7_pursued=1 & T1_pursued=0 -> (G7_pursued'=0);
155 endmodule
156
157 formula G7_achievable = T1_achievable;
158
159 // ID: G8
160 // Name: Detect patient's health status
161 // Type: sequence
162 // Relation to children: and
163 // Children: G10, G11
164 module G8
165   G8_pursued : [0..1] init 0;
166
167   [pursue_G8] G8_pursued=0 & G8_achieved_maintain=false & sensorAvailable -> (
      ↪ G8_pursued'=1);
168   [pursue_G10] G8_pursued=1 & G8_achieved_maintain=false & G10_achieved=0 ->
      ↪ true;
169   [pursue_G11] G8_pursued=1 & G8_achieved_maintain=false & G10_achieved=1 ->
      ↪ true;
```

```
170
171   [achieved_G8] G8_pursued=1 & (G10_achieved=1 & G11_achieved=1) -> (G8_pursued'
      ↪ =0);
172
173   [skip_G8] G8_pursued=1 & G10_pursued=0 & G11_pursued=0 -> (G8_pursued'=0);
174 endmodule
175
176 formula G8_achieved_maintain = enoughBattery & highReliability;
177 formula G8_achievable = G10_achievable * G11_achievable;
178
179 // ID: G9
180 // Name: Enact treatment
181 // Type: sequence
182 // Relation to children: and
183 // Children: G13, G12
184 module G9
185   G9_pursued : [0..1] init 0;
186   G9_achieved : [0..1] init 0;
187
188   [pursue_G9] G9_pursued=0 & G9_achieved=0 -> (G9_pursued'=1);
189   [pursue_G13] G9_pursued=1 & G9_achieved=0 & G12_achieved=1 -> true;
190   [pursue_G12] G9_pursued=1 & G9_achieved=0 & G12_achieved=0 -> true;
191
192   [achieved_G9] G9_pursued=1 & (G13_achieved=1 & G12_achieved=1) -> (G9_pursued'
      ↪ =0) & (G9_achieved'=1);
193
194   [skip_G9] G9_pursued=1 & G13_pursued=0 & G12_pursued=0 -> (G9_pursued'=0);
195 endmodule
196
197 formula G9_achievable = G13_achievable * G12_achievable;
198
199 // ID: G10
200 // Name: Read vital signs
201 // Type: basic
202 // Relation to children: and
203 // Children: T4
204 module G10
205   G10_pursued : [0..1] init 0;
206   G10_achieved : [0..1] init 0;
207
```

```
208   [pursue_G10] G10_pursued=0 & G10_achieved=0 -> (G10_pursued'=1);
209   [pursue_T4] G10_pursued=1 & G10_achieved=0 & R0=true -> true;
210
211   [achieved_G10] G10_pursued=1 & (T4_achieved=1) -> (G10_pursued'=0) & (
      ↪ G10_achieved'=1);
212
213   [skip_G10] G10_pursued=1 & T4_pursued=0 -> (G10_pursued'=0);
214 endmodule
215
216 formula G10_achievable = T4_achievable;
217
218 // ID: G11
219 // Name: Analyse data
220 // Type: choice
221 // Relation to children: or
222 // Children: G14, G15
223 module G11
224   G11_pursued : [0..1] init 0;
225   G11_achieved : [0..1] init 0;
226   G11_chosen : [0..2] init 0;
227
228   [pursue_G11] G11_pursued=0 & G11_achieved=0 -> (G11_pursued'=1);
229   [pursue_G14] G11_pursued=1 & G11_achieved=0 & G11_chosen!=2 -> (G11_chosen'=1)
      ↪ ;
230   [pursue_G15] G11_pursued=1 & G11_achieved=0 & G11_chosen!=1 -> (G11_chosen'=2)
      ↪ ;
231
232   [achieved_G11] G11_pursued=1 & (G14_achieved=1 | G15_achieved=1) -> (
      ↪ G11_pursued'=0) & (G11_achieved'=1);
233
234   [skip_G11] G11_pursued=1 & G14_pursued=0 & G15_pursued=0 -> (G11_pursued'=0);
235 endmodule
236
237 formula G11_achievable = G14_achievable + G15_achievable - (G14_achievable *
      ↪ G15_achievable);
238
239 // ID: G12
240 // Name: Administer medicine
241 // Type: alternative
242 // Relation to children: or
```

```
243 // Children: G16, G17
244 module G12
245   G12_pursued : [0..1] init 0;
246   G12_achieved : [0..1] init 0;
247 
248   [pursue_G12] G12_pursued=0 & G12_achieved=0 & pharmacyAvailable&atHome -> (
      ↪ G12_pursued'=1);
249   [pursue_G16] G12_pursued=1 & G12_achieved=0 & G17_pursued=0 -> true;
250   [pursue_G17] G12_pursued=1 & G12_achieved=0 & G16_pursued=0 -> true;
251 
252   [achieved_G12] G12_pursued=1 & (G16_achieved=1 | G17_achieved=1) -> (
      ↪ G12_pursued'=0) & (G12_achieved'=1);
253 
254   [skip_G12] G12_pursued=1 & G16_pursued=0 & G17_pursued=0 -> (G12_pursued'=0);
255 endmodule
256 
257 formula G12_achievable = G16_achievable + G17_achievable - (G16_achievable *
      ↪ G17_achievable);
258 
259 // ID: G13
260 // Name: Analyse side effect
261 // Type: sequence
262 // Relation to children: and
263 // Children: G18, G19
264 module G13
265   G13_pursued : [0..1] init 0;
266   G13_achieved : [0..1] init 0;
267 
268   [pursue_G13] G13_pursued=0 & G13_achieved=0 -> (G13_pursued'=1);
269   [pursue_G18] G13_pursued=1 & G13_achieved=0 & G18_achieved_maintain=false ->
      ↪ true;
270   [pursue_G19] G13_pursued=1 & G13_achieved=0 & G18_achieved_maintain=true ->
      ↪ true;
271 
272   [achieved_G13] G13_pursued=1 & (G18_achieved_maintain=true & G19_achieved=1)
      ↪ -> (G13_pursued'=0) & (G13_achieved'=1);
273 
274   [skip_G13] G13_pursued=1 & G18_pursued=0 & G19_pursued=0 -> (G13_pursued'=0);
275 endmodule
276 
```

```
277 formula G13_achievable = G18_achievable * G19_achievable;
278
279 // ID: G14
280 // Name: Remote analysis
281 // Type: basic
282 // Relation to children: and
283 // Children: T5
284 module G14
285   G14_pursued : [0..1] init 0;
286   G14_achieved : [0..1] init 0;
287   T5_failed : [0..8] init 0;
288
289   [pursue_G14] G14_pursued=0 & G14_achieved=0 -> (G14_pursued'=1);
290   [pursue_T5] G14_pursued=1 & G14_achieved=0 -> (T5_failed'=min(8, T5_failed+1))
      ↪ ;
291
292   [achieved_G14] G14_pursued=1 & (T5_achieved=1) -> (G14_pursued'=0) & (
      ↪ G14_achieved'=1);
293
294   [skip_G14] G14_pursued=1 & T5_pursued=0 -> (G14_pursued'=0);
295 endmodule
296
297 formula G14_achievable = T5_achievable;
298
299 // ID: G15
300 // Name: Local analysis
301 // Type: basic
302 // Relation to children: and
303 // Children: T6
304 module G15
305   G15_pursued : [0..1] init 0;
306   G15_achieved : [0..1] init 0;
307
308   [pursue_G15] G15_pursued=0 & G15_achieved=0 -> (G15_pursued'=1);
309   [pursue_T6] G15_pursued=1 & G15_achieved=0 -> true;
310
311   [achieved_G15] G15_pursued=1 & (T6_achieved=1) -> (G15_pursued'=0) & (
      ↪ G15_achieved'=1);
312
313   [skip_G15] G15_pursued=1 & T6_pursued=0 -> (G15_pursued'=0);
```

```
endmodule

formula G15_achievable = T6_achievable;

// ID: G16
// Name: Change drug
// Type: basic
// Relation to children: and
// Children: T7
module G16
  G16_pursued : [0..1] init 0;
  G16_achieved : [0..1] init 0;

  [pursue_G16] G16_pursued=0 & G16_achieved=0 -> (G16_pursued'=1);
  [pursue_T7] G16_pursued=1 & G16_achieved=0 -> true;

  [achieved_G16] G16_pursued=1 & (T7_achieved=1) -> (G16_pursued'=0) & (
    ↪ G16_achieved'=1);

  [skip_G16] G16_pursued=1 & T7_pursued=0 -> (G16_pursued'=0);
endmodule

formula G16_achievable = T7_achievable;

// ID: G17
// Name: Change dose
// Type: basic
// Relation to children: and
// Children: T8
module G17
  G17_pursued : [0..1] init 0;
  G17_achieved : [0..1] init 0;

  [pursue_G17] G17_pursued=0 & G17_achieved=0 -> (G17_pursued'=1);
  [pursue_T8] G17_pursued=1 & G17_achieved=0 -> true;

  [achieved_G17] G17_pursued=1 & (T8_achieved=1) -> (G17_pursued'=0) & (
    ↪ G17_achieved'=1);

  [skip_G17] G17_pursued=1 & T8_pursued=0 -> (G17_pursued'=0);
```

```
endmodule

formula G17_achievable = T8_achievable;

// ID: G18
// Name: Monitor side effects
// Type: basic
// Relation to children: and
// Children: T9
module G18
  G18_pursued : [0..1] init 0;

  [pursue_G18] G18_pursued=0 & G18_achieved_maintain=false & medicationApplied
    ↪ -> (G18_pursued'=1);
  [pursue_T9] G18_pursued=1 & G18_achieved_maintain=false & R3=true -> true;

  [achieved_G18] G18_pursued=1 & (T9_achieved=1) -> (G18_pursued'=0);

  [skip_G18] G18_pursued=1 & T9_pursued=0 -> (G18_pursued'=0);
endmodule

formula G18_achieved_maintain = inSideEffectWindow;
formula G18_achievable = T9_achievable;

// ID: G19
// Name: Trigger Alarm Service
// Type: choice
// Relation to children: or
// Children: G20, G21
module G19
  G19_pursued : [0..1] init 0;
  G19_achieved : [0..1] init 0;
  G19_chosen : [0..2] init 0;

  [pursue_G19] G19_pursued=0 & G19_achieved=0 & inEmergency -> (G19_pursued'=1);
  [pursue_G20] G19_pursued=1 & G19_achieved=0 & G19_chosen!=2 -> (G19_chosen'=1)
    ↪ ;
  [pursue_G21] G19_pursued=1 & G19_achieved=0 & G19_chosen!=1 -> (G19_chosen'=2)
    ↪ ;

```

```
389   [achieved_G19] G19_pursued=1 & (G20_achieved=1 | G21_achieved=1) -> (
        ↪ G19_pursued'=0) & (G19_achieved'=1);
390
391   [skip_G19] G19_pursued=1 & G20_pursued=0 & G21_pursued=0 -> (G19_pursued'=0);
392 endmodule
393
394 formula G19_achievable = G20_achievable + G21_achievable - (G20_achievable *
        ↪ G21_achievable);
395
396 // ID: G20
397 // Name: Trigger SMS Service
398 // Type: basic
399 // Relation to children: and
400 // Children: T10
401 module G20
402   G20_pursued : [0..1] init 0;
403   G20_achieved : [0..1] init 0;
404
405   [pursue_G20] G20_pursued=0 & G20_achieved=0 -> (G20_pursued'=1);
406   [pursue_T10] G20_pursued=1 & G20_achieved=0 & R4>35 -> true;
407
408   [achieved_G20] G20_pursued=1 & (T10_achieved=1) -> (G20_pursued'=0) & (
        ↪ G20_achieved'=1);
409
410   [skip_G20] G20_pursued=1 & T10_pursued=0 -> (G20_pursued'=0);
411 endmodule
412
413 formula G20_achievable = T10_achievable;
414
415 // ID: G21
416 // Name: Trigger alarm service
417 // Type: basic
418 // Relation to children: and
419 // Children: T11
420 module G21
421   G21_pursued : [0..1] init 0;
422   G21_achieved : [0..1] init 0;
423
424   [pursue_G21] G21_pursued=0 & G21_achieved=0 -> (G21_pursued'=1);
425   [pursue_T11] G21_pursued=1 & G21_achieved=0 & networkAvailable -> true;
```

```
426
427   [achieved_G21] G21_pursued=1 & (T11_achieved=1) -> (G21_pursued'=0) & (
      ↪ G21_achieved'=1);
428
429   [skip_G21] G21_pursued=1 & T11_pursued=0 -> (G21_pursued'=0);
430 endmodule
431
432 formula G21_achievable = T11_achievable;
433
434
435 const double T1_achievable = 0.2;
436 const double T10_achievable = 0.2;
437 const double T11_achievable = 0.5;
438 const double T2_achievable = 0.6;
439 const double T3_achievable = 0.8;
440 const double T4_achievable = 0.3;
441 const double T5_achievable = 0.5;
442 const double T6_achievable = 0.4;
443 const double T7_achievable = 0.3;
444 const double T8_achievable = 0.6;
445 const double T9_achievable = 0.8;
446
447 module ChangeManager
448   T1_pursued: [0..1] init 0;
449   T1_achieved: [0..1] init 0;
450   T10_pursued: [0..1] init 0;
451   T10_achieved: [0..1] init 0;
452   T11_pursued: [0..1] init 0;
453   T11_achieved: [0..1] init 0;
454   T2_pursued: [0..1] init 0;
455   T2_achieved: [0..1] init 0;
456   T3_pursued: [0..1] init 0;
457   T3_achieved: [0..1] init 0;
458   T4_pursued: [0..1] init 0;
459   T4_achieved: [0..1] init 0;
460   T5_pursued: [0..1] init 0;
461   T5_achieved: [0..1] init 0;
462   T6_pursued: [0..1] init 0;
463   T6_achieved: [0..1] init 0;
464   T7_pursued: [0..1] init 0;
```

```
465 T7_achieved: [0..1] init 0;
466 T8_pursued: [0..1] init 0;
467 T8_achieved: [0..1] init 0;
468 T9_pursued: [0..1] init 0;
469 T9_achieved: [0..1] init 0;
470
471 // Task T1: Trigger alarm by pushing emergency button
472 [pursue_T1] T1_pursued=0 & T1_achieved=0 -> (T1_pursued'=1);
473 [try_T1] T1_pursued=1 & T1_achieved=0 -> T1_achievable: (T1_achieved'=1) + 1-
      ↪ T1_achievable: (T1_pursued'=0);
474 [achieved_T1] T1_pursued=1 & T1_achieved=1 -> true;
475
476
477 // Task T10: Enact SMS Service
478 [pursue_T10] T10_pursued=0 & T10_achieved=0 -> (T10_pursued'=1);
479 [try_T10] T10_pursued=1 & T10_achieved=0 -> T10_achievable: (T10_achieved'=1)
      ↪ + 1-T10_achievable: (T10_pursued'=0);
480 [achieved_T10] T10_pursued=1 & T10_achieved=1 -> true;
481
482
483 // Task T11: Enact alarm service
484 [pursue_T11] T11_pursued=0 & T11_achieved=0 -> (T11_pursued'=1);
485 [try_T11] T11_pursued=1 & T11_achieved=0 -> T11_achievable: (T11_achieved'=1)
      ↪ + 1-T11_achievable: (T11_pursued'=0);
486 [achieved_T11] T11_pursued=1 & T11_achieved=1 -> true;
487
488
489 // Task T2: Perform Manual Tracking
490 [pursue_T2] T2_pursued=0 & T2_achieved=0 -> (T2_pursued'=1);
491 [try_T2] T2_pursued=1 & T2_achieved=0 -> T2_achievable: (T2_achieved'=1) + 1-
      ↪ T2_achievable: (T2_pursued'=0);
492 [achieved_T2] T2_pursued=1 & T2_achieved=1 -> true;
493
494
495 // Task T3: Perform Automatic Tracking
496 [pursue_T3] T3_pursued=0 & T3_achieved=0 -> (T3_pursued'=1);
497 [try_T3] T3_pursued=1 & T3_achieved=0 -> T3_achievable: (T3_achieved'=1) + 1-
      ↪ T3_achievable: (T3_pursued'=0);
498 [achieved_T3] T3_pursued=1 & T3_achieved=1 -> true;
499
```

```
500
501 // Task T4: Send sensed data
502 [pursue_T4] T4_pursued=0 & T4_achieved=0 -> (T4_pursued'=1);
503 [try_T4] T4_pursued=1 & T4_achieved=0 -> T4_achievable: (T4_achieved'=1) + 1-
     ↪ T4_achievable: (T4_pursued'=0);
504 [achieved_T4] T4_pursued=1 & T4_achieved=1 -> true;
505
506
507 // Task T5: Perform remote analysis
508 [pursue_T5] T5_pursued=0 & T5_achieved=0 -> (T5_pursued'=1);
509 [try_T5] T5_pursued=1 & T5_achieved=0 -> T5_achievable: (T5_achieved'=1) + 1-
     ↪ T5_achievable: (T5_pursued'=0);
510 [achieved_T5] T5_pursued=1 & T5_achieved=1 -> true;
511
512
513 // Task T6: Perform local analysis
514 [pursue_T6] T6_pursued=0 & T6_achieved=0 -> (T6_pursued'=1);
515 [try_T6] T6_pursued=1 & T6_achieved=0 -> T6_achievable: (T6_achieved'=1) + 1-
     ↪ T6_achievable: (T6_pursued'=0);
516 [achieved_T6] T6_pursued=1 & T6_achieved=1 -> true;
517
518
519 // Task T7: Apply drug change
520 [pursue_T7] T7_pursued=0 & T7_achieved=0 -> (T7_pursued'=1);
521 [try_T7] T7_pursued=1 & T7_achieved=0 -> T7_achievable: (T7_achieved'=1) + 1-
     ↪ T7_achievable: (T7_pursued'=0);
522 [achieved_T7] T7_pursued=1 & T7_achieved=1 -> true;
523
524
525 // Task T8: Apply dose change
526 [pursue_T8] T8_pursued=0 & T8_achieved=0 -> (T8_pursued'=1);
527 [try_T8] T8_pursued=1 & T8_achieved=0 -> T8_achievable: (T8_achieved'=1) + 1-
     ↪ T8_achievable: (T8_pursued'=0);
528 [achieved_T8] T8_pursued=1 & T8_achieved=1 -> true;
529
530
531 // Task T9: Apply side effect monitor
532 [pursue_T9] T9_pursued=0 & T9_achieved=0 -> (T9_pursued'=1);
533 [try_T9] T9_pursued=1 & T9_achieved=0 -> T9_achievable: (T9_achieved'=1) + 1-
     ↪ T9_achievable: (T9_pursued'=0);
```

```
534   [achieved_T9] T9_pursued=1 & T9_achieved=1 -> true;
535
536 endmodule
537
538
539 module System
540   pharmacyAvailable: bool init true;
541   atHome: bool init true;
542   inSideEffectWindow: bool init false;
543   medicationApplied: bool init true;
544   inEmergency: bool init false;
545   highPrecision: bool init false;
546   patientTracking: bool init true;
547   privacyEnabled: bool init true;
548   enoughBattery: bool init true;
549   highReliability: bool init true;
550   sensorAvailable: bool init true;
551   networkAvailable: bool init true;
552
553   R0: bool init true;
554   R1: [0..5] init 5;
555   R2: [0..5] init 5;
556   R3: bool init true;
557   R4: [1..4] init 4;
558   R5: [0..10] init 10;
559
560 endmodule
```

Listing B.1: TAS Full PRISM goal controller

Appendix C

PRISM Goal Controller for LSL

```
dtmc
// ID: G0
// Name: Provide Sample Delivery
// Type: interleaved
// Relation to children: and
// Children: G1, G2
module G0
  G0_pursued : [0..1] init 0;
  G0_achieved : [0..1] init 0;

  [pursue_G0] G0_pursued=0 & G0_achieved=0 -> (G0_pursued'=1);
  [pursue_G1] G0_pursued=1 & G0_achieved=0 & G1_achieved_maintain=false -> true;
  [pursue_G2] G0_pursued=1 & G0_achieved=0 -> true;

  [achieved_G0] G0_pursued=1 & (G1_achieved_maintain=true & G2_achieved=1) -> (
    ↪ G0_pursued'=0) & (G0_achieved'=1);

  [skip_G0] G0_pursued=1 & G1_pursued=0 & G2_pursued=0 -> (G0_pursued'=0);
endmodule

formula G0_achievable = G1_achievable * G2_achievable;

// ID: G1
// Name: Monitor Sample Track System
// Type: basic
// Relation to children: and
// Children: T1
module G1
```

```
  G1_pursued : [0..1] init 0;

  [pursue_G1] G1_pursued=0 & G1_achieved_maintain=false & sampleLoaded -> (
    ↪ G1_pursued'=1);
  [pursue_T1] G1_pursued=1 & G1_achieved_maintain=false & R0=true -> true;

  [achieved_G1] G1_pursued=1 & (T1_achieved=1) -> (G1_pursued'=0);

  [skip_G1] G1_pursued=1 & T1_pursued=0 -> (G1_pursued'=0);
endmodule

formula G1_achieved_maintain = !sampleLoaded;
formula G1_achievable = T1_achievable;

// ID: G2
// Name: Deliver Sample and Pickup Sample if Needed
// Type: sequence
// Relation to children: and
// Children: G13, G6
module G2
  G2_pursued : [0..1] init 0;
  G2_achieved : [0..1] init 0;

  [pursue_G2] G2_pursued=0 & G2_achieved=0 -> (G2_pursued'=1);
  [pursue_G13] G2_pursued=1 & G2_achieved=0 & G13_achieved=0 -> true;
  [pursue_G6] G2_pursued=1 & G2_achieved=0 & G13_achieved=1 -> true;

  [achieved_G2] G2_pursued=1 & (G13_achieved=1 & G6_achieved=1) -> (G2_pursued'
    ↪ =0) & (G2_achieved'=1);

  [skip_G2] G2_pursued=1 & G13_pursued=0 & G6_pursued=0 -> (G2_pursued'=0);
endmodule

formula G2_achievable = G13_achievable * G6_achievable;

// ID: G3
// Name: Pickup Sample
// Type: sequence
// Relation to children: and
// Children: G5
```

```
module G3
  G3_pursued : [0..1] init 0;
  G3_achieved : [0..1] init 0;

  [pursue_G3] G3_pursued=0 & G3_achieved=0 -> (G3_pursued'=1);
  [pursue_G5] G3_pursued=1 & G3_achieved=0 & G5_achieved=0 -> true;

  [achieved_G3] G3_pursued=1 & (G5_achieved=1) -> (G3_pursued'=0) & (
    ↪ G3_achieved'=1);

  [skip_G3] G3_pursued=1 & G5_pursued=0 -> (G3_pursued'=0);
endmodule

formula G3_achievable = G5_achievable;

// ID: G4
// Name: Load sample to deliver
// Type: basic
// Relation to children: and
// Children: T5
module G4
  G4_pursued : [0..1] init 0;
  G4_achieved : [0..1] init 0;

  [pursue_G4] G4_pursued=0 & G4_achieved=0 -> (G4_pursued'=1);
  [pursue_T5] G4_pursued=1 & G4_achieved=0 & R2=true -> true;

  [achieved_G4] G4_pursued=1 & (T5_achieved=1) -> (G4_pursued'=0) & (
    ↪ G4_achieved'=1);

  [skip_G4] G4_pursued=1 & T5_pursued=0 -> (G4_pursued'=0);
endmodule

formula G4_achievable = T5_achievable;

// ID: G5
// Name: Approach Nurse and Pickup Sample
// Type: sequence
// Relation to children: and
// Children: G7, G8, G9
```

```
104 module G5
105   G5_pursued : [0..1] init 0;
106   G5_achieved : [0..1] init 0;
107 
108   [pursue_G5] G5_pursued=0 & G5_achieved=0 -> (G5_pursued'=1);
109   [pursue_G7] G5_pursued=1 & G5_achieved=0 & G7_achieved=0 -> true;
110   [pursue_G8] G5_pursued=1 & G5_achieved=0 & G7_achieved=1 & G9_achieved=0 ->
        ↪ true;
111   [pursue_G9] G5_pursued=1 & G5_achieved=0 & G7_achieved=1 & G8_achieved=1 ->
        ↪ true;
112 
113   [achieved_G5] G5_pursued=1 & (G7_achieved=1 & G8_achieved=1 & G9_achieved=1)
        ↪ -> (G5_pursued'=0) & (G5_achieved'=1);
114 
115   [skip_G5] G5_pursued=1 & G7_pursued=0 & G8_pursued=0 & G9_pursued=0 -> (
        ↪ G5_pursued'=0);
116 endmodule
117 
118 formula G5_achievable = G7_achievable * G8_achievable * G9_achievable;
119 
120 // ID: G6
121 // Name: Deliver Sample to Lab
122 // Type: sequence
123 // Relation to children: and
124 // Children: G10, G11, G12
125 module G6
126   G6_pursued : [0..1] init 0;
127   G6_achieved : [0..1] init 0;
128 
129   [pursue_G6] G6_pursued=0 & G6_achieved=0 -> (G6_pursued'=1);
130   [pursue_G10] G6_pursued=1 & G6_achieved=0 & G10_achieved=0 -> true;
131   [pursue_G11] G6_pursued=1 & G6_achieved=0 & G10_achieved=1 & G12_achieved=0 &
        ↪ G11_achieved_maintain=false -> true;
132   [pursue_G12] G6_pursued=1 & G6_achieved=0 & G10_achieved=1 &
        ↪ G11_achieved_maintain=true -> true;
133 
134   [achieved_G6] G6_pursued=1 & (G10_achieved=1 & G11_achieved_maintain=true &
        ↪ G12_achieved=1) -> (G6_pursued'=0) & (G6_achieved'=1);
135 
```

```
136   [skip_G6] G6_pursued=1 & G10_pursued=0 & G11_pursued=0 & G12_pursued=0 -> (
      ↪ G6_pursued'=0);
137 endmodule
138
139 formula G6_achievable = G10_achievable * G11_achievable * G12_achievable;
140
141 // ID: G7
142 // Name: Approach the Nurse
143 // Type: basic
144 // Relation to children: and
145 // Children: T2
146 module G7
147   G7_pursued : [0..1] init 0;
148   G7_achieved : [0..1] init 0;
149
150   [pursue_G7] G7_pursued=0 & G7_achieved=0 -> (G7_pursued'=1);
151   [pursue_T2] G7_pursued=1 & G7_achieved=0 & R1>=2 -> true;
152
153   [achieved_G7] G7_pursued=1 & (T2_achieved=1) -> (G7_pursued'=0) & (
      ↪ G7_achieved'=1);
154
155   [skip_G7] G7_pursued=1 & T2_pursued=0 -> (G7_pursued'=0);
156 endmodule
157
158 formula G7_achievable = T2_achievable;
159
160 // ID: G8
161 // Name: Scan Sample Barcode
162 // Type: basic
163 // Relation to children: and
164 // Children: T3
165 module G8
166   G8_pursued : [0..1] init 0;
167   G8_achieved : [0..1] init 0;
168
169   [pursue_G8] G8_pursued=0 & G8_achieved=0 -> (G8_pursued'=1);
170   [pursue_T3] G8_pursued=1 & G8_achieved=0 & R1>=1 -> true;
171
172   [achieved_G8] G8_pursued=1 & (T3_achieved=1) -> (G8_pursued'=0) & (
      ↪ G8_achieved'=1);
```

```
173
174   [skip_G8] G8_pursued=1 & T3_pursued=0 -> (G8_pursued'=0);
175 endmodule
176
177 formula G8_achievable = T3_achievable;
178
179 // ID: G9
180 // Name: Pickup Sample
181 // Type: basic
182 // Relation to children: and
183 // Children: T4
184 module G9
185   G9_pursued : [0..1] init 0;
186   G9_achieved : [0..1] init 0;
187
188   [pursue_G9] G9_pursued=0 & G9_achieved=0 -> (G9_pursued'=1);
189   [pursue_T4] G9_pursued=1 & G9_achieved=0 & R1>=2&R2=true -> true;
190
191   [achieved_G9] G9_pursued=1 & (T4_achieved=1) -> (G9_pursued'=0) & (
      ↪ G9_achieved'=1);
192
193   [skip_G9] G9_pursued=1 & T4_pursued=0 -> (G9_pursued'=0);
194 endmodule
195
196 formula G9_achievable = T4_achievable;
197
198 // ID: G10
199 // Name: Approach the Lab
200 // Type: basic
201 // Relation to children: and
202 // Children: T6
203 module G10
204   G10_pursued : [0..1] init 0;
205   G10_achieved : [0..1] init 0;
206
207   [pursue_G10] G10_pursued=0 & G10_achieved=0 -> (G10_pursued'=1);
208   [pursue_T6] G10_pursued=1 & G10_achieved=0 & R1>=2 -> true;
209
210   [achieved_G10] G10_pursued=1 & (T6_achieved=1) -> (G10_pursued'=0) & (
      ↪ G10_achieved'=1);
```

```
211
212   [skip_G10] G10_pursued=1 & T6_pursued=0 -> (G10_pursued'=0);
213 endmodule
214
215 formula G10_achievable = T6_achievable;
216
217 // ID: G11
218 // Name: Monitor Sample Data
219 // Type: sequence
220 // Relation to children: and
221 // Children: T7, T8
222 module G11
223   G11_pursued : [0..1] init 0;
224
225   [pursue_G11] G11_pursued=0 & G11_achieved_maintain=false & sampleLoaded -> (
      ↪ G11_pursued'=1);
226   [pursue_T7] G11_pursued=1 & G11_achieved_maintain=false & T7_achieved=0 &
      ↪ sampleLoaded -> true;
227   [pursue_T8] G11_pursued=1 & G11_achieved_maintain=false & T7_achieved=1 ->
      ↪ true;
228
229   [achieved_G11] G11_pursued=1 & (T7_achieved=1 & T8_achieved=1) -> (
      ↪ G11_pursued'=0);
230
231   [skip_G11] G11_pursued=1 & T7_pursued=0 & T8_pursued=0 -> (G11_pursued'=0);
232 endmodule
233
234 formula G11_achieved_maintain = analysisEnded;
235 formula G11_achievable = T7_achievable * T8_achievable;
236
237 // ID: G12
238 // Name: Give sample to Robotic Arm
239 // Type: basic
240 // Relation to children: and
241 // Children: T9
242 module G12
243   G12_pursued : [0..1] init 0;
244   G12_achieved : [0..1] init 0;
245
246   [pursue_G12] G12_pursued=0 & G12_achieved=0 -> (G12_pursued'=1);
```

```
247   [pursue_T9] G12_pursued=1 & G12_achieved=0 & R1>=2 -> true;
248
249   [achieved_G12] G12_pursued=1 & (T9_achieved=1) -> (G12_pursued'=0) & (
        ↪ G12_achieved'=1);
250
251   [skip_G12] G12_pursued=1 & T9_pursued=0 -> (G12_pursued'=0);
252 endmodule
253
254 formula G12_achievable = T9_achievable;
255
256 // ID: G13
257 // Name: Prepare sample
258 // Type: choice
259 // Relation to children: or
260 // Children: G3, G4
261 module G13
262   G13_pursued : [0..1] init 0;
263   G13_achieved : [0..1] init 0;
264   G13_chosen : [0..2] init 0;
265
266   [pursue_G13] G13_pursued=0 & G13_achieved=0 -> (G13_pursued'=1);
267   [pursue_G3] G13_pursued=1 & G13_achieved=0 & G13_chosen!=2 -> (G13_chosen'=1);
268   [pursue_G4] G13_pursued=1 & G13_achieved=0 & G13_chosen!=1 -> (G13_chosen'=2);
269
270   [achieved_G13] G13_pursued=1 & (G3_achieved=1 | G4_achieved=1) -> (
        ↪ G13_pursued'=0) & (G13_achieved'=1);
271
272   [skip_G13] G13_pursued=1 & G3_pursued=0 & G4_pursued=0 -> (G13_pursued'=0);
273 endmodule
274
275 formula G13_achievable = G3_achievable + G4_achievable - (G3_achievable *
        ↪ G4_achievable);
276
277
278 const double T1_achievable = 0.8;
279 const double T2_achievable = 0.8;
280 const double T3_achievable = 0.8;
281 const double T4_achievable = 0.8;
282 const double T5_achievable = 0.8;
283 const double T6_achievable = 0.8;
```

```
284 const double T7_achievable = 0.8;
285 const double T8_achievable = 0.8;
286 const double T9_achievable = 0.8;
287
288 module ChangeManager
289   T1_pursued: [0..1] init 0;
290   T1_achieved: [0..1] init 0;
291   T2_pursued: [0..1] init 0;
292   T2_achieved: [0..1] init 0;
293   T3_pursued: [0..1] init 0;
294   T3_achieved: [0..1] init 0;
295   T4_pursued: [0..1] init 0;
296   T4_achieved: [0..1] init 0;
297   T5_pursued: [0..1] init 0;
298   T5_achieved: [0..1] init 0;
299   T6_pursued: [0..1] init 0;
300   T6_achieved: [0..1] init 0;
301   T7_pursued: [0..1] init 0;
302   T7_achieved: [0..1] init 0;
303   T8_pursued: [0..1] init 0;
304   T8_achieved: [0..1] init 0;
305   T9_pursued: [0..1] init 0;
306   T9_achieved: [0..1] init 0;
307
308   // Task T1: Process Lab Sample Delivery Request
309   [pursue_T1] T1_pursued=0 & T1_achieved=0 -> (T1_pursued'=1);
310   [try_T1] T1_pursued=1 & T1_achieved=0 -> T1_achievable: (T1_achieved'=1) + 1-
        ↪ T1_achievable: (T1_pursued'=0);
311   [achieved_T1] T1_pursued=1 & T1_achieved=1 -> true;
312
313
314   // Task T2: Navigate to Nurse Position
315   [pursue_T2] T2_pursued=0 & T2_achieved=0 -> (T2_pursued'=1);
316   [try_T2] T2_pursued=1 & T2_achieved=0 -> T2_achievable: (T2_achieved'=1) + 1-
        ↪ T2_achievable: (T2_pursued'=0);
317   [achieved_T2] T2_pursued=1 & T2_achieved=1 -> true;
318
319
320   // Task T3: Barcode Scanner Service
321   [pursue_T3] T3_pursued=0 & T3_achieved=0 -> (T3_pursued'=1);
```

```
322 [try_T3] T3_pursued=1 & T3_achieved=0 -> T3_achievable: (T3_achieved'=1) + 1-
      ↪ T3_achievable: (T3_pursued'=0);
323 [achieved_T3] T3_pursued=1 & T3_achieved=1 -> true;
324
325
326 // Task T4: Pickup Task
327 [pursue_T4] T4_pursued=0 & T4_achieved=0 -> (T4_pursued'=1);
328 [try_T4] T4_pursued=1 & T4_achieved=0 -> T4_achievable: (T4_achieved'=1) + 1-
      ↪ T4_achievable: (T4_pursued'=0);
329 [achieved_T4] T4_pursued=1 & T4_achieved=1 -> true;
330
331
332 // Task T5: Load Sample Manually
333 [pursue_T5] T5_pursued=0 & T5_achieved=0 -> (T5_pursued'=1);
334 [try_T5] T5_pursued=1 & T5_achieved=0 -> T5_achievable: (T5_achieved'=1) + 1-
      ↪ T5_achievable: (T5_pursued'=0);
335 [achieved_T5] T5_pursued=1 & T5_achieved=1 -> true;
336
337
338 // Task T6: Navigate to Lab Position
339 [pursue_T6] T6_pursued=0 & T6_achieved=0 -> (T6_pursued'=1);
340 [try_T6] T6_pursued=1 & T6_achieved=0 -> T6_achievable: (T6_achieved'=1) + 1-
      ↪ T6_achievable: (T6_pursued'=0);
341 [achieved_T6] T6_pursued=1 & T6_achieved=1 -> true;
342
343
344 // Task T7: Track Sample Location
345 [pursue_T7] T7_pursued=0 & T7_achieved=0 -> (T7_pursued'=1);
346 [try_T7] T7_pursued=1 & T7_achieved=0 -> T7_achievable: (T7_achieved'=1) + 1-
      ↪ T7_achievable: (T7_pursued'=0);
347 [achieved_T7] T7_pursued=1 & T7_achieved=1 -> true;
348
349
350 // Task T8: Monitor Sample Temperature
351 [pursue_T8] T8_pursued=0 & T8_achieved=0 -> (T8_pursued'=1);
352 [try_T8] T8_pursued=1 & T8_achieved=0 -> T8_achievable: (T8_achieved'=1) + 1-
      ↪ T8_achievable: (T8_pursued'=0);
353 [achieved_T8] T8_pursued=1 & T8_achieved=1 -> true;
354
355
```

```
356   // Task T9: Delivery Task
357   [pursue_T9] T9_pursued=0 & T9_achieved=0 -> (T9_pursued'=1);
358   [try_T9] T9_pursued=1 & T9_achieved=0 -> T9_achievable: (T9_achieved'=1) + 1-
        ↪ T9_achievable: (T9_pursued'=0);
359   [achieved_T9] T9_pursued=1 & T9_achieved=1 -> true;
360
361 endmodule
362
363 module System
364   sampleLoaded: bool init true;
365   analysisEnded: bool init true;
366   R0: bool init true;
367   R1: [0..5] init 5;
368   R2: bool init true;
369 endmodule
```

Listing C.1: LSL Full PRISM goal controller

References

[1] F. Dalpiaz, A. Borgida, J. Horkoff, and J. Mylopoulos, "Runtime goal models: Keynote," in *IEEE 7th International Conference on Research Challenges in Information Science (RCIS)*, 2013, pp. 1–11. 5, 9, 10, 17, 18, 19

[2] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From goals to components: a combined approach to self-management," in *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1–8. [Online]. Available: https://doi.org/10.1145/1370018.1370020 5, 10, 22, 23, 24, 28

[3] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: A reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, 2015, pp. 9–16. 5, 10, 13, 22, 23, 24, 28, 35, 53, 54

[4] D. F. Mendonça, G. Nunes Rodrigues, R. Ali, V. Alves, and L. Baresi, "Goda: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information and Software Technology*, vol. 80, pp. 245–264, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916301471 5, 7, 29

[5] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, vol. 17, pp. 184–206, 2015, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157411921400162X 9

[6] D. Weyns, *An introduction to self-adaptive systems: A contemporary software engineering perspective.* John Wiley & Sons, 2020. 9

[7] Z. Yang, Z. Li, Z. Jin, and Y. Chen, "A systematic literature review of requirements modeling and analysis for self-adaptive systems," in *International working conference on requirements engineering: foundation for software quality.* Springer, 2014, pp. 55–71. 9

[8] D. F. Mendonça, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi, "Goda: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information and Software Technology*, vol. 80, 2016. 9, 17, 19, 21, 22, 28

[9] R. Calinescu and G. N. Rodrigues, "Goal controller synthesis for self-adaptive systems," in *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 2023, pp. 1–6. 10, 14, 15, 20, 26, 46, 48, 63

[10] A. Dezfouli and B. W. Balleine, "Actions, action sequences and habits: evidence that goal-directed and habitual action control are hierarchically organized," *PLoS computational biology*, vol. 9, no. 12, p. e1003364, 2013. 11

[11] F. Cushman and A. Morris, "Habitual control of goal selection in humans," *Proceedings of the National Academy of Sciences*, vol. 112, no. 45, pp. 13 817–13 822, 2015. 11

[12] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," in *Computer Performance Evaluation: Modelling Techniques and Tools*, T. Field, P. G. Harrison, J. Bradley, and U. Harder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 200–204. 14, 20, 21

[13] E. Syriani, L. Luhunu, and H. Sahraoui, "Systematic mapping study of template-based code generation," *Computer Languages, Systems & Structures*, vol. 52, pp. 43–62, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842417301239 14, 35, 61

[14] D. Weyns and R. Calinescu, "Tele assistance: A self-adaptive service-based system exemplar," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2015, pp. 88–92. 14, 85, 86

[15] M. Askarpour, C. Tsigkanos, C. Menghi, R. Calinescu, P. Pelliccione, S. García, R. Caldas, T. J. Von Oertzen, M. Wimmer, L. Berardinelli *et al.*, "Robomax: Robotic mission adaptation exemplars," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2021, pp. 245–251. 14, 56, 85, 99

[16] A. van Lamsweerde, "Goal-oriented requirements engineering: a guided tour," in *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, 2001, pp. 249–262. 17

[17] R. Ali, F. Dalpiaz, and P. Giorgini, "A goal-based framework for contextual requirements modeling and analysis," *Requirements engineering*, vol. 15, pp. 439–458, 2010. 17, 18, 19

[18] E. Gil, "Mutrose: A goal-oriented framework for mission specification and decomposition of multi-robot systems," Ph.D. dissertation, Master's thesis, UnB, 2021. ix, 3, 13, 14, 20, 26, 2021. 19

[19] M. A. Langford, K. H. Chan, J. E. Fleck, P. K. McKinley, and B. H. Cheng, "Modalas: addressing assurance for learning-enabled autonomous systems in the face of uncertainty," *Software and Systems Modeling*, vol. 22, no. 5, pp. 1543–1563, 2023. 19

[20] H. Araujo, M. R. Mousavi, and M. Varshosaz, "Testing, validation, and verification of robotic and autonomous systems: A systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3542945 20

[21] C. Baier, L. de Alfaro, V. Forejt, and M. Kwiatkowska, *Model Checking Probabilistic Systems*. Cham: Springer International Publishing, 2018, pp. 963–999. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_28 20

[22] T. Sekizawa, F. Otsuki, K. Ito, and K. Okano, "Behavior verification of autonomous robot vehicle in consideration of errors and disturbances," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3, 2015, pp. 550–555. 22

[23] G. N. Rodrigues, "Edge case study – prism model," https://github.com/Genaina/Formalise23/blob/main/EDGE-CaseStudy/EDGE_MDP.pm, 2023, accessed: June 2025. [Online]. Available: https://github.com/Genaina/Formalise23/blob/main/EDGE-CaseStudy/EDGE_MDP.pm 28, 30

[24] J. Pimentel and J. Castro, "pistar tool–a pluggable online tool for goal modeling," in *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 2018, pp. 498–499. 76, 79

[25] V. R. B. G. Caldiera and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, pp. 528–532, 1994. 112

[26] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 4, pp. 589–610, 2022. 115