Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Testing Context Dependent Behaviours with Design-Space Exploration

Scientific Students' Association Report

Author:

Dominik Frey

Advisor:

dr. Kristóf Marussy
dr. András Vörös

2023

Contents

Kivonat

A kritikus Kiber-Fizikai Rendszerek (CPS) biztonságos és helyes működése gyakran függ a működési környezettől, beleértve a környezeti hatásokat, az komponensek telepítését és a közöttük történő kommunikációt. A rendszer és a környezet bonyolult függőségének megjelenítésére időben változó gráf modelleket használnak a modellvezérelt rendszertervezésben. Így a magas szintű követelményeket futásidejű monitorokként formalizálják, amelyek figyelik a gráf időbeli változását. Például az autonóm vezetési alkalmazásokban egy futásidejű modell képviselheti a jármű állapotát és a szituáció résztvevői közötti kapcsolatokat, míg a futásidejű monitorok észlelhetik a közlekedési szabályok megsértését és biztosíthatják a biztonságos vezetést.

Az ilyen kontextusfüggő rendszerek helyességének ellenőrzése olyan tesztsorozatot igényel, amely lefedi a rendszer és a monitorok viselkedését. Az autonóm vezetésben különféle forgalmi helyzeteknek megfelelő tesztforgatókönyveket használnak a közlekedési szabályok betartásának ellenőrzésére. Azonban a viselkedések nagy mennyisége miatt a tesztsorozatok kézi létrehozása elegendő lefedettséggel nem kivitelezhető. Az kimerítő felsorolás szintén túl időigényes, a véletlenszerű felderítés pedig nem alkalmas érdekes és értékes tesztsorozatok generálására.

Ez a kutatás a Tervezési Tér Felderítés (DSE) heurisztikáját kívánja kihasználni a tesztsorozatok automatikus és hatékony levezetésére. Felhasználjuk a futásidejű monitorokat a felderítés irányítására és elegendő lefedettség elérésére. Különösen (i) kiterjesztjük a Parametrikus Időzített Automaták (PTE) formalizmusát gráf mintaillesztéssel, ami átfogó hátteret biztosít a futásidejű monitorokhoz. (ii) Bevezetünk egy specializált cél függvényt a DSE irányítására a monitorozó automatak alapján, biztosítva a releváns tesztsorozatok származtatását, amelyek elérnek elegendő viselkedési lefedettséget. (iii) Integráljuk a javasolt monitorokat az nyílt forráskódú *Refinery* gráf feldolgozó keretrendszerbe mind a futásidejű végrehajtás, mind a tervezési tér felderítés céljából. (iv) Értékeljük a javasolt megközelítés alkalmazhatóságát és skálázhatóságát több esettanulmányon, beleértve a teszteset generálást autonóm járművek számára.

Ennek eredményeként a mérnökök magas szintű specifikációkat és monitor automatákat használhatnak mind a komplex kontextusfüggő rendszerek tesztelésére, mind pedig azok futásidejű monitorozására.

Abstract

The safe and correct operation of critical Cyber-Physical Systems (CPS) often depends on their operating context, including environmental effects, the deployment of components, and the communication between them. To capture complex interdependence of the system and environment, time-evolving graph models describing the state of the system and its context are used in model-driven systems engineering. Thus, high-level requirements can be formalized as runtime monitors observing the time evolution of the graph. For example, in autonomous driving applications, a runtime model can represent the state and relationships of the vehicle and other actors in the scene, while runtime monitors can detect traffic rule violations and ensure safe driving.

Verifying the correctness of such context-dependent systems requires a test suite of system model trajectories (i.e., sequences of graphs) that cover the behaviour of the system and the monitors. In autonomous driving, test scenarios corresponding to various traffic situations are used to verify the adherence to traffic rules. However, the large amount of behaviours makes manual construction of such test sequences with sufficient coverage infeasible. Exhaustive enumeration is also prohibitively time-consuming, and random exploration lacks the focus to generate interesting and valuable test sequences.

This research aims to leverage Design-Space Exploration (DSE) heuristics to automatically and efficiently derive test sequences. We explicitly reuse the runtime monitors to guide the exploration and reach sufficient coverage.

In particular, (i) we extend the Parametric Timed Automata (PTE) formalism with graph pattern matching, which will serve as a comprehensive background formalism for runtime monitors. This extension incorporates Complex Event Processing (CEP) and temporal logic-based monitors, that can represent Metric Temporal Graph Logic (MTGL) specifications. We introduce a (ii) specialized objective function for guiding DSE based on the monitoring automata, ensuring the derivation of relevant test sequences reaching the sufficient coverage of bevaiours. We (iii) integrate the proposed monitors into the open-source *Refinery* graph processing framework for both runtime execution and design space exploration. We (iv) evaluate the applicability and scalability of the proposed approach on multiple case studies, including scenario generation for autonomous vehicles.

As a result, engineers can utilize high-level specifications with monitor automata both for testing complex context-dependent systems, as well as for their runtime monitoring.

Chapter 1

Introduction

1.1 Motivation

The growing use of Cyber-Physical Systems (CPS) such as autonomous vehicles (AV) has brought the focus on ensuring their safety-critical behaviour. Quality assurance of critical software-intensive systems often uses the automated synthesis of test data to reduce conceptual gaps in the test cases [26]. Filling these gaps is a hard task for automated tools, as these systems are highly context-dependent and have vast state space, so specific heuristics are needed to cover only the meaningful system states in the generated tests.

1.2 Related approaches

To decrease the complexity of these systems, engineers use qualitative abstraction [38] in Model-Based Systems testing. By representing systems using their qualitative attributes, we can simplify intricate systems into more understandable and manageable models. This allows for a holistic view of the system's key features without being overwhelmed by every small detail.

In the field of systems testing, among many verification techniques, runtime monitoring [20] of temporal specifications like Metric Temporal Graph Logic (MTGL) [21, 33], Metric First-Order Temporal Logic (MFOTL) [11, 12], Event Pattern Language (EPL) [16] is a widely used tool to reason about the behaviour of the system.

In terms of model-based test generation, the field uses Design-Space Exploration (DSE) [28, 15, 1, 18, 3, 36]. DSE is a process that seeks to discover optimal design options within a given domain, subject to various objectives and heuristics.

1.3 Contribution

We enhance the Parametric Timed Automata (PTE) formalism by integrating graph pattern matching capabilities.

We introduce a specialized fitness function to guide Design-Space Exploration (DSE). This function is calculated on the extended monitoring automata to drive the derivation test sequences, which can achieve sufficient coverage and generation time of the system behaviors.

Our proposed monitoring automata are integrated into the Refinery open-source graph processing framework. This integration supports both runtime execution and design-space exploration.

We undertake an experimental evaluation of the applicability and scalability of our approach. Multiple case studies are presented, including scenario generation for traffic situations, autonomous vehicles, transmit-receiver networks, and gesture recognition systems. The evaluation focuses on the diversity of test sequence generation and the time efficiency of our method.

We propose a technique for quantifying the diversity of test sequences. By deriving a single metric to measure the similarity of the generated sequences.

We conducted our research based on the principles of the state of the art of the research line [13].

1.4 Structure of this document

The remainder of this document is structured as follows.

- In Chapter 2, we briefly review the mathematical fundamentals of modeling, runtime monitoring, and design space explorations. Moreover, we introduce a system domain of traffic situations that will serve as our running example throughout the work.
- In Chapter 3, in introduce our proposed workflow for automatically and efficiently deriving test sequences using Design-Space Exploration guided by runtime monitors.
- In Chapter 4, we explain the components of the workflow, and how they are implemented, including
  - our proposed extension of Parametric Timed Automata (PTE) with graph pattern matching as a formal background for runtime monitors.
  - the DSE workflow, how the implemented components are used with existing DSE strategies [22, 35, 4, 28, 15].
  - our specialized objective function and acceptance criterion for guiding DSE according to the runtime monitors state configuration.
- In Chapter 6, we discuss similar technologies in the field of monitoring and Design-Space Exploration.
- In Chapter 7, we summarize our contributions and discuss future directions.

Chapter 2

Background

In this chapter, we briefly overview the use of graph models and runtime monitoring in systems engineering, as well as the corresponding mathematical background. As a running example, we will use the following case study about the generation of traffic situtations (TRAF), which will also serve as illustration in the remainder of the work.

2.1 Running example

A traffic scene, as defined by [34] and then further specified by [5] is a snapshot of both the environment's static and dynamic elements and their interactions. The static scenery includes the lane network, stationary items like traffic lights, road elevations, and environmental conditions. On the other hand, dynamic elements or actors encompass vehicles, pedestrians, and the ego vehicle, with details on their state (e.g., position, speed) and attributes (e.g., colour, car door status). Relations describe how these elements interact, such as the distance between two vehicles or their placement on lanes. When multiple traffic scenes are linked with their temporal changes, it forms a scenario. A scenario consists of an initial scene followed by actions and events driven by actors' goals, which can be either short-term (e.g., reaching a location) or long-term (e.g., safe driving).

Definition 1 (Dangerous scene). We call a traffic scene *dangerous scene* when traffic rules are about to be violated and/or some actors of a traffic scene are about to crash. ▪

Definition 2 (Dangerous scenario). We call a traffic scenario *dangerous scenario* or *dangerous situation* when the traffic scenario results in a *dangerous scene.* ▪

2.2 Graph modeling

Model-driven systems Engineering (MDSE) is an approach to systems engineering that emphasizes the use of models to abstract and understand complex systems. We use graph models and graph theory concepts to represent systems, their components, and relationships.

A graph consists of two main elements:

- **Nodes**: Represent entities, like actors, products, or events.
- **Relationships**: Represent connections between nodes. They have a direction, and type, and can have properties.

The type of nodes and relationships are often referred to as class or label.

2.2.1 Metamodeling

Definition 3 (Metamodel). The structural description of a graph model is called metamodel. In a metamodel, one can describe domain-specific constraints for the structure of the model. We formalize the metamodel of our target domain $\mathcal{M} = \langle \Sigma, \alpha, dom \rangle \in M$ using an algebraic representation with a signature Σ, an arity function $\alpha : \Sigma \to \mathbb{N}$ and a domain function $dom : \Sigma \to \Omega$.

dom specifies what values can be associated with a specific symbol. By default, and in the vast majority of cases $dom(s) = \{0, 1\}$. In the rest of the work, we leave *dom* out of the signature, if it does not differ from the default.

$$\Sigma = \{C_1, \ldots, C_n, R_1, \ldots, R_m\}$$

with:

- Unary predicate symbols $\{C_1, \ldots, C_n\}$ defined for each object *Class*, with the arity function $\alpha(C_i) = 1$.
- n-ary predicate symbols $\{R_1, \ldots, R_m\}$ representing *Relationships* between n objects, with the arity function $\alpha(R_i) = n$.

A metamodel also imposes several structural constraints to enforce syntactic consistency for model manipulation or model persistence operations:

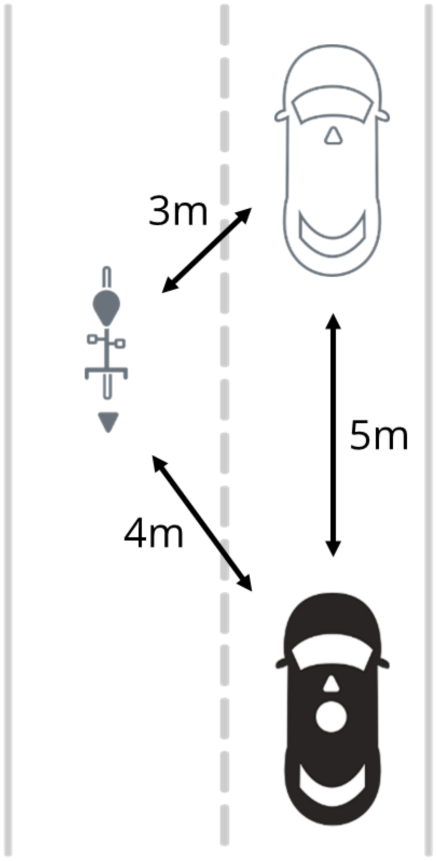
- Type hierarchy, which requires that $C_1(o) \implies C_2(o)$ if C_1 is a child of C_2.
- Type compliance, which requires that for any relation $R(o_1, o_2)$, o_1 and o_2 must have compliant types.

The enforcement of these constraints is not explicitly discussed in this work. ▪

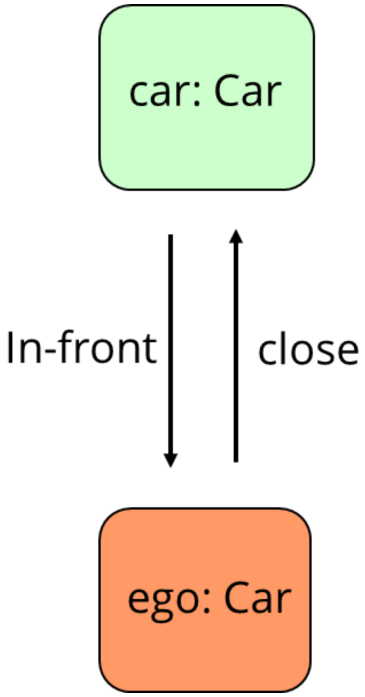
Definition 4 (Qualitative abstraction). Qualitative abstraction is a Model-Based Systems Engineering (MBSE) technique to manage complexity.

Abstraction, in general, is the process of reducing the complexity of a system by focusing on a higher level of detail and ignoring specific nuances or lower-level details. Qualitative abstraction, specifically, focuses on the abstract representation of systems based on their qualities or characteristics rather than their quantitative details. It allows engineers to make decisions and analyze systems based on broad characteristics without getting lost in the smaller details [5]. ▪

Example 1. *Figure 2.1 shows an example of how we leverage qualitative abstraction when trying to model real-life traffic scenes from domain* Traf. *Figure 2.1(a) shows distances between objects calculated from their position vectors, e.g.,* $d(\text{ego}, \text{car}) = \sqrt{(x_{\text{ego}} - x_{\text{car}})^2 + (y_{\text{ego}} - y_{\text{car}})^2}$. *In Figure 2.1(b), we derive the abstract relations* close *and* in-front *from the position vectors, e.g.,* $\text{close}(\text{ego}, \text{car}) \Leftrightarrow d(\text{ego}, \text{car}) \leq 10\,m$, $\text{in-front}(\text{ego}, \text{car}) \Leftrightarrow x_{\text{ego}} < x_{\text{car}}$.



(a) A concrete traffic scene where a car is 5 meters in front of another and a cyclist is passing by in the opposite lane.



(b) The qualitative abstraction of Figure 2.1a. We are only interested in the relationship of the two cars.

Figure 2.1: Example for a qualitative abstraction using our Traf running example as context.

Example 2. *We introduce how we model a system domain using our running example and metamodeling technique introduced in Definition 3. By applying qualitative abstraction as in Definition 4, we map real-life traffic scenarios to a simple model representation sufficient to capture and simulate abstract positions and maneuvers.*

Lanes and positions will be dealt with using lanelet representation [14], where a lanelet represents a segment of the road. Lanelets are attached directly and for simplicity, we only represent straight roads.

- *The width of a lanelet is 3.8 meters, following the width of a freeway lane by the Hungarian standard.*
- *The length of a lanelet is 10 meters.*
- *The length of a car is 5 meters.*
- *A car is considered placed on a lanelet, when the full length of the car is inside the lanelet.*
- *the positions of the cars are determined by their placement on the lanelets:*

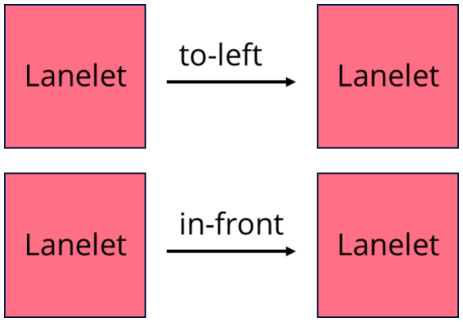
abstraction	$d(\mathrm{a},\mathrm{b}) \Leftrightarrow \sqrt{(x_\mathrm{a}-x_\mathrm{b})^2+(y_\mathrm{a}-y_\mathrm{b})^2}$
close	< 10
mid-distance	< 20
far	< 30

In our example, we use a metamodel an extended signature $\mathcal{M}_{Traf} = \langle \Sigma_{Traf}, \alpha_{Traf}, TR_{Traf} \rangle$ *where* $TR_{Traf} \subseteq TR$ *is a set of transformation rules that later can be executed on a model defined over this metamodel. Figure 2.2 demonstrates the type of nodes and relations we use, which are:*

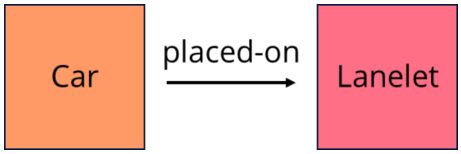
$\Sigma_{Traf} = \{Car, Lanelet, in\text{-}front, to\text{-}left\}$

α_{Traf} :

$\alpha_{Traf}(Car) = 1$
$\alpha_{Traf}(Lanelet) = 1$
$\alpha_{Traf}(to\text{-}left) = 2$
$\alpha_{Traf}(in\text{-}front) = 2$



(a) A lanelet can be in front, or to the left of another lanelet.



(b) A car can be placed on a lanelet.

Figure 2.2: The resented metamodel $\mathcal{M}_{Traf}$ to represent the domain of the TRAF system.

2.2.2 Instance models

Definition 5 (Instance model). An instance model consists of concrete objects and the relationships between them. The objects always have a unique identifier that they can be referred to as.

The formal representation of a graph model:

Given a metamodel $\mathcal{M} = \langle \Sigma, \alpha \rangle$, an instance model is a logic structure $\mathcal{G} = \langle O_\mathcal{G}, I_\mathcal{G} \rangle \in G$ where:

- $O_\mathcal{G}$ is the finite set of objects in the model.
- $I_\mathcal{G}$ gives an interpretation of values from a specific domain for each symbol $s \in \Sigma$ as:

$$I_\mathcal{G}(s) : O_\mathcal{G}^{\alpha(s)} \to \mathcal{D}.$$

  Where the default and most commonly used scenario is $\mathcal{D} = \{0, 1\}$, however, in some cases, we specify a different domain for $\mathcal{D}$, for instance, $\mathbb{N}$. This way, tuples of objects interpreted on $s \in \Sigma$ can store any values or structures, that can be considered as the property or additional information of an object or vector of objects. ▪

Example 3. *Now that we have a metamodel to provide our domain, we define an initial model as the root model state of the trajectory generation. Let us have* $\mathcal{G}_{Traf} = \langle O_{Traf}, I_{Traf} \rangle$ *defined over* $\mathcal{M}_{Traf}$ *where:*

- $O_{Traf} = \{c_1, c_2, ll_1, \ldots, ll_n, rl_1, \ldots, rl_n\}$ c_1, c_2 *are objects representing two cars, and* ll_i *and* rl_i *are left and right lanelets, representing two lanes that consist of* n *segments.*
- I_{Traf}*: We explicitly add interpretations to these objects to build a model structure that later can be manipulated through transformations:*

$I_{Traf}(Car)(c_1)$	$= 1$	c_1 *is a Car.*
$I_{Traf}(Car)(c_2)$	$= 1$	c_2 *is a Car.*
$I_{Traf}(\text{placed-on})(c_1, ll_1)$	$= 1$	c_1 *is placed on the first left lanelet* ll_1*.*
$I_{Traf}(\text{placed-on})(c_2, rl_1)$	$= 1$	c_2 *is placed on the first right lanelet* rl_1*.*
$I_{Traf}(Lanelet)(ll_i)$	$= 1$	*Every* $ll_i \in \{ll_1, \ldots, ll_n\}$ *is a Lanelet.*
$I_{Traf}(Lanelet)(rl_i)$	$= 1$	*Every* $rl_i \in \{rl_1, \ldots, rl_n\}$ *is a Lanelet.*
$I_{Traf}(\text{to-left})(ll_i, rl_i)$	$= 1$	*Every left lanelet is to the left of the right lanelet with the same index.*
$I_{Traf}(\text{in-front})(ll_{i+1}, ll_i)$	$= 1$	*Every left lanelet is in front of the one with the previous index.*
$I_{Traf}(\text{in-front})(rl_{i+1}, rl_i)$	$= 1$	*Every right lanelet is in front of the one with the previous index.*

The intuition behind this model structure is shown in Figure 2.3. c_1 *and* c_2 *are two cars placed on* ll_1 *and* rl_1 *the two starting lanelets of a road that consists of two neighbouring lanes, a left lane, which we represent with a series of lanelets* $ll_i \in \{ll_1, \ldots, ll_n\}$ *and a right lane* $rl_i \in \{rl_1, \ldots, rl_n\}$*.*

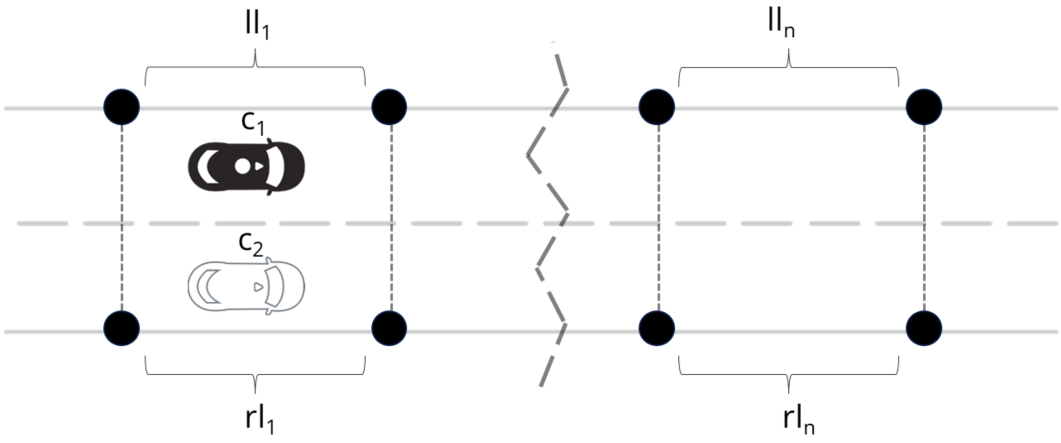


Figure 2.3: Real-world intuition for the given interpretation of $\mathcal{G}_{Traf}$.

Definition 6 (Graph query). A graph query ϕ is defined over a metamodel $\mathcal{M} = \langle \Sigma, \alpha \rangle$ and an infinite vector of (object) parameters $\hat{p} = (p_1, p_2, \ldots)$, using grammar rules described below:

$\phi :=$		
	$C(p) \mid R(p_1, p_2, \dots)$	global value, class, and relationship query
	$p_1 = p_2$	equivalence
	$\neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$	logic connectives
	$\exists p : \phi \mid \forall p : \phi$	quantified expression
	$\phi^+(p_1, p_2, \dots)$	transitive closure

Given a graph query ϕ, it can be evaluated on a model $\mathcal{G}$ along with a variable binding $\beta : \hat{p} \to O_{\mathcal{G}}$ (denoted as $[\![\phi]\!]^{\mathcal{G}}_{\beta}$), which can result in either 0 or 1.

$$s(\hat{p}) \Leftrightarrow I(s)(\beta(\hat{p}))$$

.

For an object o, $o \in O_{\mathcal{G}} \Leftrightarrow \exists s \in \Sigma, \hat{o} : s(\hat{o}) = 1, o \in \hat{o}$, which means that we only consider an object part of a model $\mathcal{G}$, if there exists an interpretation for a symbol in Σ on object vector $\hat{o}$ in the model, where o is in $\hat{o}$ and the following condition holds: $s(o) = 1$.

Example 4. *We provide some basic graph query formulae in the* Traf *domain extended with an Actor symbol with an arity of 1:*

1. ***Type checking**: Whether 'car' is a Car.*

   $\phi_1(car) :=$ *Car(car)*

2. ***Exploring Relationships**: Whether if 'car' has any Actors in front of it.*

   $\phi_2(car) :=$ $\exists$ *actor: Actor(actor)*
   $\wedge$ *Car(car)*
   $\wedge$ *in-front(actor, car)*

3. ***Pattern Matching**: Whether 'actor' is in front of 'car', but not placed on the same lane.*

   $\phi_3(actor, car) :=$ $\forall$ *lane: Lane(lane)*
   $\wedge$ *Actor(actor)*
   $\wedge$ *Car(car)*
   $\wedge$ *in-front(actor, car)*
   $\wedge$ *placed-on(actor, lane)*
   $\wedge \neg$ *placed-on(car, lane)*

4. ***Traversing Multiple Levels**: Whether 'actor1' is in front of another Actor that is in front of a Car (2-hop relationships).*

   $\phi_4(actor_1) :=$ $\exists$ $actor_2$, *car: Actor*($actor_1$)
   $\wedge$ *Actor*($actor_2$)
   $\wedge$ *Car(car)*
   $\wedge$ *in-front*($actor_2$, *car*)
   $\wedge$ *in-front*($actor_1$, $actor_2$)

Definition 7 (Graph pattern matching). The main strength of graph databases lies in their ability to efficiently answer relationship-driven questions. To retrieve or manipulate data in graph databases, one uses graph pattern matching.

While matching graph patterns, rather than operating over rows or records as in SQL, operations are performed over nodes and relationships with the help of graph queries introduced in Definition 6.

Assuming that a graph query $\phi(\hat{p})$ is defined for a metamodel $\mathcal{M} = \langle \Sigma, \alpha \rangle$, with object parameters $\hat{p} = (p_1, \ldots, p_n)$, we define a graph matching function as:

$$\Psi_{\mathcal{G}}(\phi(\hat{p})) \to 2^{\Omega_{\hat{p}}}$$

which takes all possible $\Omega_{\hat{p}} = \{\beta_1, \beta_2, \ldots\}$ parameter bindings, for parameter vector $\hat{p}$ and returns a set of bindings $\{\beta'_1, \ldots, \beta'_k\} \subseteq \Omega_{\hat{p}}$, that were evaluated to true on the graph predicate $[\![\phi^{\mathcal{G}}_{\beta_i}]\!]$.

For example, graph matching $\Psi(\phi_1(car))$ (Example 4) on instance model, depicted in Example 3, would return $\{(c_1), (C_2)\}$, where c_1 and c_2 are the only Car objects in the model.

2.2.3 Graph databases

Graph databases are a type of NoSQL database optimized for handling complex relationships in data, in contrast to relational databases which primarily manage tabular data.

Graph databases fill a gap that relational databases don't cover effectively: the intricate, interconnected relationships in datasets. For domains like social networking, recommendation engines, fraud detection, and more, the relations between entities are as crucial as the entities themselves. Graph databases excel in:

- **Expressiveness**: Representing complex relationships directly as first-class entities.
- **Flexibility**: Adapting to evolving data models without the need for schema migrations.
- **Performance**: Traversing relationships is often faster than joining tables, especially for deep relationships.

Challenges:

- **Complexity**: As data grows, the graph can become complex. Regularly reassessing and refining the model is essential.
- **Data Consistency**: Ensuring data integrity and consistency across the graph can be challenging, especially with distributed systems.

Some notable graph databases are Neo4j [19] or Refinery [39].

2.3 Runtime verification

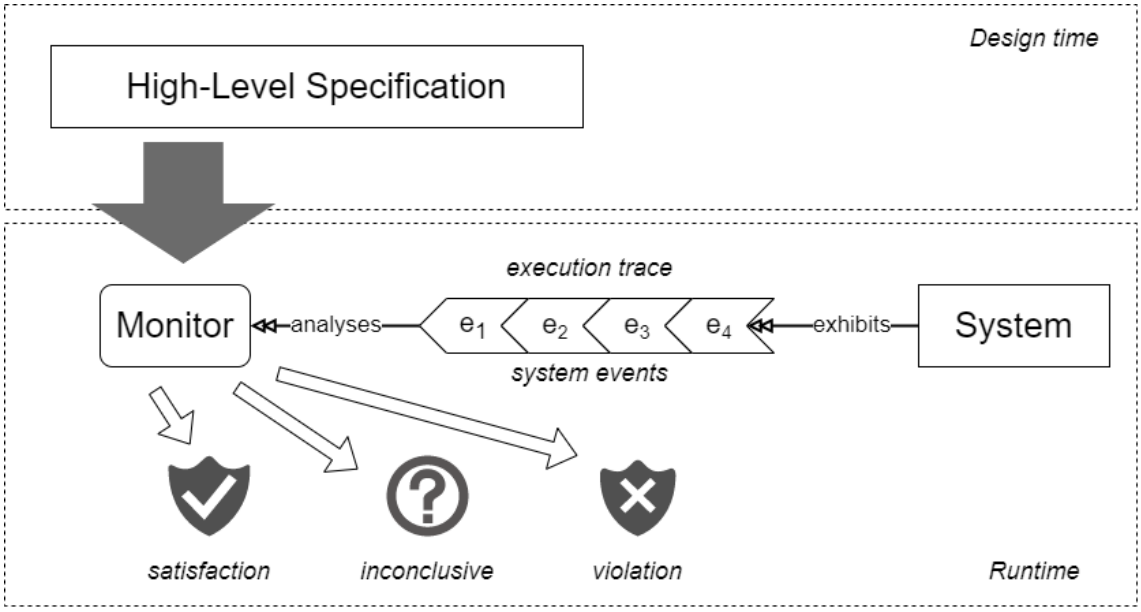


Figure 2.4: Runtime monitor synthesis and operational set-up [20].

Runtime Verification (RV) is a formal method that is both lightweight and thorough, offering a practical supplement to traditional comprehensive verification methods such as model checking and theorem proving by examining a single execution trace of a system. Despite the trade-off of having limited execution coverage, RV can provide highly accurate information about the system's behaviour during operation [10].

The System Under Scrutiny (SUS) could be anything from a software system, hardware, a cyber-physical system, or a sensor network, to any other system whose dynamic behaviour can be observed.

Fundamentally, RV operates under the presumption of a logic outlining the correctness criteria that the SUS should comply with. Based on these specifications, monitoring programs are developed and integrated to run alongside the SUS. Their purpose is to analyze the ongoing execution of the SUS, represented as a sequence of events, and deduce any instances of compliance or breaches concerning these specifications (Figure 2.4).

2.3.1 Parametric Event Automata

Event Automata (EA) and its parameterized variant, Parametric Event Automata (PEA) introduced in papers [9, 2], provide a mechanism to model and reason about system behaviours based on events. A PEA extends the standard EA by introducing parameters to events, enabling a more expressive representation of system behaviours.

Definition 8 (Parametric Event Automaton). The algebraic representation of a PEA, $\mathcal{A} \in A$ is the following:

$$\mathcal{A} = \langle Q, q_0, F, P, E, \Delta \rangle$$

where:

- Q: A finite set of states.
- q_0: The initial state $q_0 \in Q$.
- F: The set of accept states $F \subseteq Q$.
- P: A finite set of parameters, where we denote V as a set of possible values for parameters.
- E: A finite set of events, where $e(p_1, \ldots, p_k)$ is an event with parameters and each $p_i \in P$.
- $\Delta \subseteq Q \times E \times 2^P \times Q$: A transition relation. An element $(q, e, B, q') \in \Delta$ indicates a transition from state q to state q' on an event e with a binding set $B \subseteq P$. ▪

Semantics For an event $e(p_1, \ldots, p_k)$ and a transition (q, e, B, q'), the transition can be taken if there is a binding of parameters in B, to values in V such that the event can be matched. The binding of parameters to values is represented as a function:

$$\beta : P \to V$$

This function β maps parameters to values. For a transition to be taken, there should be a suitable β for which:

$$e(\beta(p_1), \ldots, \beta(p_k))$$

is the occurring event.

Given a PEA $\mathcal{A} = \langle Q, q_0, F, P, E, \Delta \rangle$, the system can move to a new state q' upon the occurrence of an event $e(v_1, \ldots, v_k)$ if and only if there exists a transition $(q_0, e, B, q') \in \Delta$ and a binding β such that for all parameters $p_i \in B$:

$$\beta(p_i) = v_i$$

that are the values of the parameters of the event match the binding for the transition.

Parameter Binding Mechanism The core idea behind parameter binding in PEA is to allow flexibility in specifying which occurrences of events can trigger transitions. By defining a binding set B for a transition, the PEA specifies which parameters of an event need to be matched (or bound) to values for that transition to be taken. If a parameter is not in B, its value doesn't affect the transition.

For example, for an event $e(a, b)$ and a transition $(q, e, \{a\}, q')$, only the value of the parameter a must match the binding for the transition to be taken. The value of parameter b is irrelevant for this transition.

Acceptance When the automaton reaches any state $q \in F$, it is considered to have accepted the event sequence $e_1, \ldots, e_n$ that led to that state.

Example 5. *Consider a PEA that models a takeover from the left in our* Traf *running example with events that represent changes in the system model, like in-front(car_1, car_2) which signals that 'car_1' got in front of 'car_2' and to-left(car_1, car_2) which signals that 'car_1' got to the left of 'car_2':*

1. *Parameters:* $P = \{car_1, car_2\}$
2. *Events:* $E = \{$*in-front, to-left*$\}$
3. *States:* $Q = \{$*initial, overtakeStarted, overtakeCanceled, takeOverSucceded*$\}$
4. *Initial state:* $q_0 =$ *initial*
5. *Accept states:* $F = \{$*takeOverSucceded*$\}$
6. *Transitions* (Δ)*:*

initial	*in-front*	$\{car_2, car_1\}$	*initial,*
initial	*to-left*	$\{car_1, car_2\}$	*overtakeStarted,*
overtakeStarted	*in-front*	$\{car_2, car_1\}$	*overtakeCanceled,*
overtakeStarted	*in-front*	$\{car_1, car_2\}$	*overtakeSucceeded*

The visual representation [27] of this automaton can be seen in Figure 2.5.

The automaton starts in the `initial` *state and moves to the* `overtakeStarted` *state when a* `to-left`(car_1, car_2) *event occurs,* car_1 *and* car_2 *parameters are then bound to the provided values. Finally, as an effect of an incoming event sequence* `to-left`(car_1, car_2), `in-front`(car_1, car_2)*, the automaton takes the acceptance state* `overtakeSucceeded`*.*

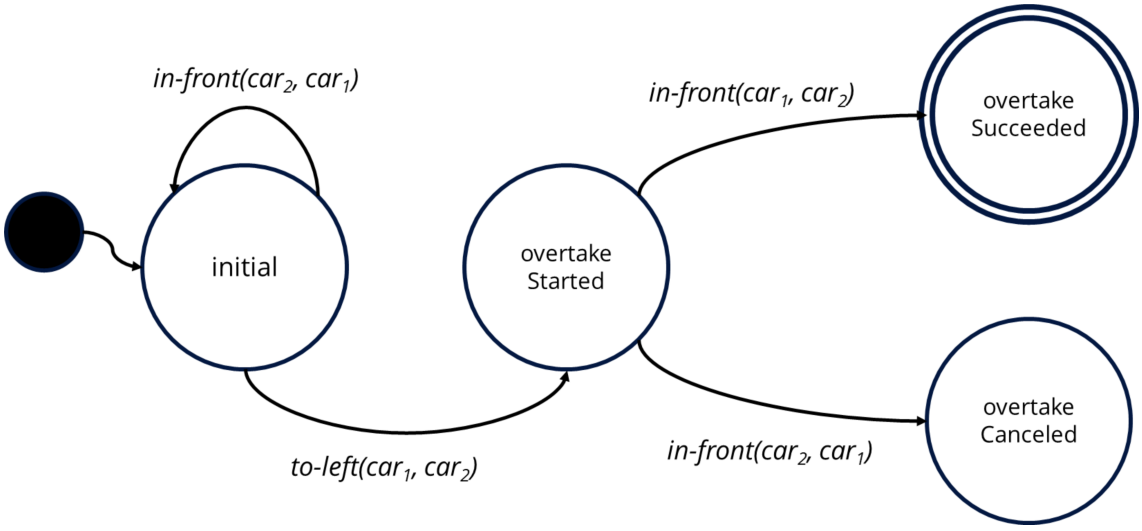


Figure 2.5: Visualization of the PEA for Example 5.

2.3.2 Monitor definition languages

In order to enhance the creation of monitors, temporal languages can be employed to define the time-dependent behaviour to be found in the test trace. The system's specification can be formalized using these languages. Subsequent subsections provide details on some current specification languages and algorithms, the majority of which are derived from Linear Temporal Logic.

2.3.2.1 Metric Temporal Graph Logic

Metric Temporal Graph Logic (MTGL) is a logic introduced to reason about timed graph sequences. It is used to express properties on the structure and attributes of states, as well

as on the occurrence of states over time that are related by their inner structure. MTGL was introduces in [21, 33].

Definition 9 describes the basic syntax for an MTGL condition.

Definition 9. (Metric Temporal Graph Conditions (MTGCs)). The class of metric temporal graph conditions Θ_{MTGC_H} for the graph H contains ψ if one of the following cases applies.

1. $\psi = \bigwedge S$ and $S = \{\theta_1, ..., \theta_n\} \subseteq \Theta_{MTGC_H}$.
   - Meaning that ψ is a conjunction of a set of conditions S. Each condition θ_i in the set S is a member of the set of all MTGCs for the graph H, denoted as Θ_{MTGC_H}.
2. $\psi = \neg\theta$ and $\theta \in \Theta_{MTGC_H}$.
   - This means that ψ is the negation of a condition θ. The condition θ is a member of the set of all MTGCs for the graph H.
3. $\psi = \exists(a, \theta)$, $a : H \to H'$, and $\theta \in \Theta_{MTGC_{H'}}$.
   - ψ is an existential quantification, stating that there exists a morphism a from graph H to graph H' such that the condition θ holds. The condition θ is a member of the set of all MTGCs for the graph H'.
4. $\psi = \theta_1 U_I \theta_2$, I is an interval over R_0, and $\{\theta_1, \theta_2\} \subseteq \Theta_{MTGC_H}$.
   - This means that ψ is a temporal condition stating that condition θ_1 holds until the condition θ_2 becomes true within the time interval I. Both conditions θ_1 and θ_2 are members of the set of all MTGCs for the graph H. ▪

2.3.2.2 Metric First Order Temporal Logic

The Metric First-Order Temporal Logic (MFOTL) is an extension of first-order logic (FOL) enriched with temporal operators and quantifiers that allow the expression of properties spanning across discrete time intervals. [11, 12]

Syntax and semantics Let I be the set of nonempty intervals over $\mathbb{N}$. A signature S is a tuple (C, R, a), where C is a finite set of constant symbols, R is a finite set of predicates disjoint from C, and the function $a : R \to \mathbb{N}$ associates each predicate $r \in R$ with an arity $a(r) \in \mathbb{N}$. V denotes a countably infinite set of variables, where we assume that

$$V \cap (C \cup R) = \emptyset$$

for every signature $S = (C, R, a)$.

Definition 10. The formulae over S are inductively defined:

1. For $t, t' \in V \cup C$, $t \approx t'$ and $t \prec t'$ are formulae.
2. For $r \in R$ and $t_1, \ldots, t_{a(r)} \in V \cup C$, $r(t_1, \ldots, t_{a(r)})$ is a formula.

3. For $x \in V$, if θ and θ' are formulae then $(\neg\theta)$, $(\theta \wedge \theta')$, and $(\exists x.\theta)$ are formulae.

4. For $I \in \mathcal{I}$, if θ and θ' are formulae then $(\Diamond_I \theta)$, $(\Box_I \theta)$, $(\theta \mathcal{S}_I \theta')$, and $(\theta \mathcal{U}_I \theta')$ are formulae.

To define the semantics of MFOTL, we need the following notions:

A (first-order) structure D over S consists of a domain $|D| \neq \emptyset$ and interpretations $c^D \in |D|$ and $r^D \subseteq |D|^{a(r)}$, for each $c \in C$ and $r \in R$.

A temporal (first-order) structure over S is a pair (D, τ), where $D = (D_0, D_1, \dots)$ is a sequence of structures over S and $\tau = (\tau_0, \tau_1, \dots)$ is a sequence of natural numbers (time stamps), where:

1. The sequence τ is monotonically increasing (i.e., $\tau_i \leq \tau_{i+1}$, for all $i \geq 0$) and makes progress (i.e., for every $i \geq 0$, there is some $j > i$ such that $\tau_j > \tau_i$).

2. D has constant domains, i.e., $|D_i| = |D_{i+1}|$, for all $i \geq 0$. We denote the domain by $|D|$ and require that $|D|$ is linearly ordered by the relation $<$.

3. Each constant symbol $c \in C$ has a rigid interpretation, i.e., $c^{D_i} = c^{D_{i+1}}$, for all $i \geq 0$. We denote the interpretation of c by c^D.

A valuation is a mapping $v : V \to |D|$. ▪

Definition 11. Let (D, τ) be a temporal structure over S, with $D = (D_0, D_1, \dots)$ and $\tau = (\tau_0, \tau_1, \dots)$, θ a formula over S, v a valuation, and $i \in \mathbb{N}$. We define $(D, \tau, v, i) \models \theta$ as follows:

- $(D, \bar{\tau}, v, i) \models t \approx t' \iff v(t) = v(t')$
- $(D, \bar{\tau}, v, i) \models t \prec t' \iff v(t) < v(t')$
- $(D, \bar{\tau}, v, i) \models r(t_1, ..., t_{\iota(r)}) \iff \langle v(t_1), ..., v(t_{\iota(r)}) \rangle \in r^{D_i}$
- $(D, \bar{\tau}, v, i) \models \neg\theta \iff (D, \bar{\tau}, v, i) \not\models \theta$
- $(D, \bar{\tau}, v, i) \models \theta \wedge \theta' \iff (D, \bar{\tau}, v, i) \models \theta$ and $(D, \bar{\tau}, v, i) \models \theta'$
- $(D, \bar{\tau}, v, i) \models \exists x.\theta \iff (D, \bar{\tau}, v[x/d], i) \models \theta,$ for some $d \in |D|$
- $(D, \bar{\tau}, v, i) \models \Diamond_I \theta \iff i > 0, \tau_i - \tau_{i-1} \in I,$ and $(D, \bar{\tau}, v, i-1) \models \theta$
- $(D, \bar{\tau}, v, i) \models \Box_I \theta \iff \tau_{i+1} - \tau_i \in I$ and $(D, \bar{\tau}, v, i+1) \models \theta$
- $(D, \bar{\tau}, v, i) \models \theta \mathcal{S}_I \theta' \iff$ for some $j \leq i, \tau_i - \tau_j \in I$,
  $(D, \bar{\tau}, v, j) \models \theta',$ and $(D, \bar{\tau}, v, k) \models \theta,$ for all $k \in [j+1, i+1)$
- $(D, \bar{\tau}, v, i) \models \theta \mathcal{U}_I \theta' \iff$ for some $j \geq i, \tau_j - \tau_i \in I$,
  $(D, \bar{\tau}, v, j) \models \theta',$ and $(D, \bar{\tau}, v, k) \models \theta,$ for all $k \in (i, j)$ ▪

Example 6. *Let us formulate some examples of a dangerous scenario in the* Traf *domain:*

1. *Within the next 2 time units, Car c_1 overtakes Car c_2 while they both are close to Car c_3:*

$$\theta = \Diamond_{[0,2]}(\textit{to-left}(c_1, c_2) \wedge \textit{close}(c_1, c_3) \wedge \textit{close}(c_2, c_3))$$

2. *Car c_1 is consistently close to Car c_2 for 3 time units, and at the same time, Car c_2 is close to Car c_3. Then, within the next 2 time units, if Car c_1 overtakes Car c_2, it becomes a dangerous situation:*

$$\theta = (\Box_{[0,3]} \mathit{close}(c_1, c_2) \wedge \Box_{[0,3]} \mathit{close}(c_2, c_3)) \rightarrow \Diamond_{[0,2]} \mathit{to\text{-}left}(c_1, c_2)$$

Monitoring algorithms for MFOTL have been used to monitor different policies on synthetic data streams. The efficiency of the algorithm is such that the monitors can also be used online to detect policy violations. Furthermore, in some cases, MFOTL monitors can be used for policy enforcement. This involves changing future actions or predicting when existing actions have consistent extensions.

2.3.2.3 Complex Event Processing

Complex Event Processing (CEP) [16] is a highly effective technology in real-time distributed environments, offering a quick and effective method for drawing inferences and correlating about events as they happen. This technology finds its applications in a wide range of domains including logistics, critical infrastructure monitoring, finance applications, business processes, the Internet of Things, autonomous unmanned aerial vehicles, and intelligent transportation.

A key attribute of CEP is the facility to define event patterns through rules. These rules can be set up using various Event Processing Languages (EPLs) like Esper EPL (EsperTech 2022) and SiddhiQL (WSO2 2022). In the CEP context, simple events are unique, occurring at a distinct point in time. The correlation of several such simple events can lead to the formation of a complex event, which offers significant and valuable data. In essence, a CEP engine automatically generates these complex events when certain conditions set in an event pattern are fulfilled. A CEP engine is a software element that enables developers to establish event patterns with the assistance of EPLs. On detecting an event pattern, the CEP engine can react instantaneously.

CEP technology's functionality can be broadly divided into three phases [32]. The different phases are demonstrated by Figure 2.6.

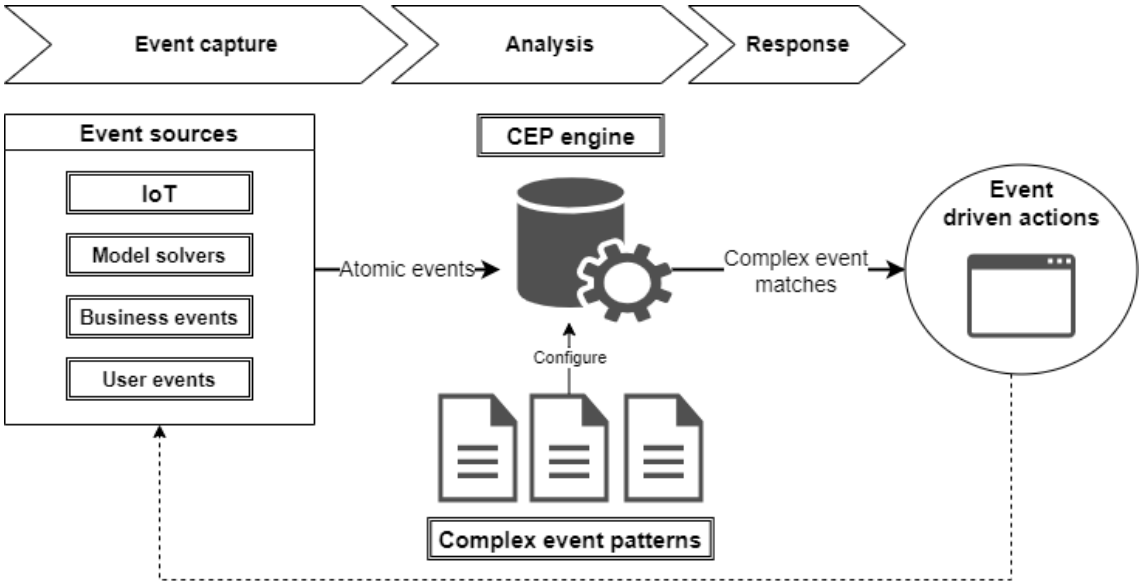


Figure 2.6: High-level overview of the stages of complex event processing.

Event Capture: This stage involves receiving and correlating simple events in real-time.

Analysis: This stage involves identifying situations of interest when the conditions pre-set in an event pattern are met.

Response: This stage involves reacting to the detected situations and informing the concerned parties. The application of CEP technology comes with numerous benefits, such as quick and automated responses, a decrease in human workload, enhancement in decision-making quality, and prevention of information overload. Compared to traditional event analysis techniques, the decision-making process becomes significantly quicker as these situations of interest can be identified and reported in real-time.

2.4 Rule-Based Design-Space Exploration

Design Space Exploration (DSE) is a process that seeks to discover optimal design options within a given domain, subject to various objectives. These design options are bounded by intricate structural and numerical limitations. The goal of rule-based DSE is to locate these options that can be obtained from an initial model by executing a series of exploration rules. Tackling a rule-based DSE problem presents a significant challenge due to the fundamentally dynamic characteristics of the problem.

2.4.1 Graph transformations

DSE applies graph transformations defined in transformation rules to derive model instances from the initial model and the already explored model instances.

Definition 12 (Graph transformation rule). We define two different kinds of transformation rules, one for creating new objects with interpretations for a model $\mathcal{G}$, and one for modifying interpretations for existing objects:

1. Changing the interpreted value of a tuple of objects for a specific symbol, in a model $\mathcal{G}$, with a metamodel $\mathcal{M} = \langle \Sigma, \alpha \rangle$:

$$PUT = \langle \phi(\hat{p}), \hat{p}', s, d \rangle$$

where:

- $\phi(\hat{p})$ is a graph query that will be matched on a given model, bounding parameters $\hat{p}$ to objects that identify the interpretation to be modified.
- $\hat{p}'$ is a parameter vector $p'_i \in \hat{p}$ providing the parameters that determine the interpretation to modify. $|\hat{p}'| = \alpha(s)$.
- s is a symbol, $s \in \Sigma$, to identify the interpretation for the matched objects.
- $d \in \mathcal{D}$ is a value to be assigned to the identified interpretations.

2. Adding a new object with interpretation for a specific symbol to a model $\mathcal{G}$ with a metamodel $\mathcal{M} = \langle \Sigma, \alpha \rangle$:

$$CREATE = \langle s, d \rangle$$

where:

- s is a symbol, $s \in \Sigma$, to identify the interpretation for the new object. $alpha(s) = 1$ is required in this case.
- $d \in \mathcal{D}$ is a value to be assigned to the identified interpretation.

From now on, we denote the domain set of transformation rules with $\mathcal{TR}$ where

$$\mathcal{TR} = \{PUT_1, \ldots, PUT_n, CREATE_1, \ldots, CREATE_m\}$$

.

Example 7. *We define a set of transformation rules for G_{Traf} that can be executed on demand, thus we can generate new scenes from the initial model. In the context of our example, we model cars moving on the road, passing through lanelets in any direction, but only if those lanelets are directly connected.*

In that regard, take the following graph query based on Definition 6:

$$\begin{aligned} \phi_{canMove}(c, l_1, l_2) := \quad & \textit{Lanelet}(l_1) \\ & \wedge\ \textit{Lanelet}(l_2) \\ & \wedge\ \textit{Car}(c) \\ & \wedge\ \textit{placed-on}(c, l_1) \\ & \wedge\ (\quad \textit{in-front}(l_1, l_2) \\ & \qquad \vee\ \textit{in-front}(l_2, l_1) \\ & \qquad \vee\ \textit{to-left}(l_1, l_2) \\ & \qquad \vee\ \textit{to-left}(l_2, l_1)\ ) \end{aligned}$$

Figure 2.7 Illustrates graph patterns that are accepted by $\phi_{canMove}(c, l_1, l_2)$. The lighter Lanelet nodes represent the $\vee$ relationship, meaning either of them could be a possible destination for c, but only one of them.

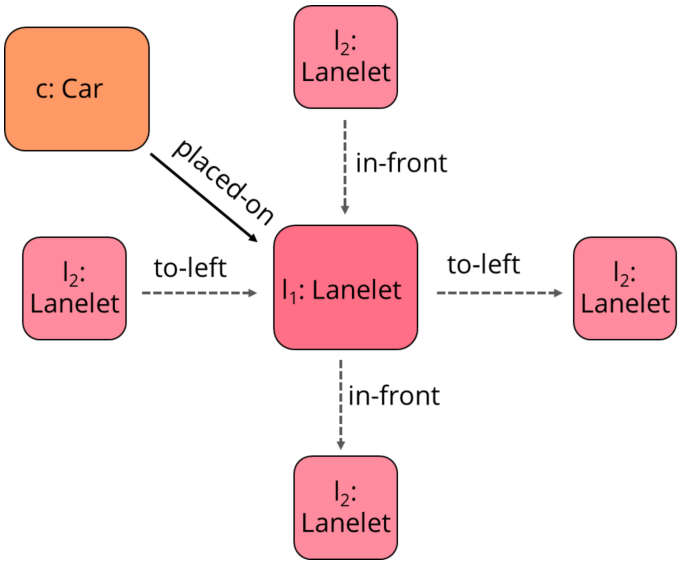


Figure 2.7: Graph patterns matched by $\phi_{canMove}(c, l_1, l_2)$.

It checks for type compliance and a graph pattern where a car c is placed on a lane l_1 that is next to another lane l_2. In our case, we want to move the car from l_1 to l_2. Based on Definition 12, let us assign a transformation rule to do that:

$$PUT_{take} := \langle \phi_{canMove}(c, l_1, l_2), (c, l_1), \textit{placed-on}, 0 \rangle$$

First, we take the car c_1 from lanelet l_1.

$$PUT_{place} := \langle \phi_{canMove}(c, l_1, l_2), (c, l_2), \textit{placed-on}, 1 \rangle$$

Then, we place it on lanelet l_2. Figure 2.8 illustrates how the two transformations above are applied by removing the placed-on(c_1, l_1) edge from the graph and inserting a new placed-on(c_1, l_2) edge. Note that l_1 and l_2 are neighbouring lanelets.

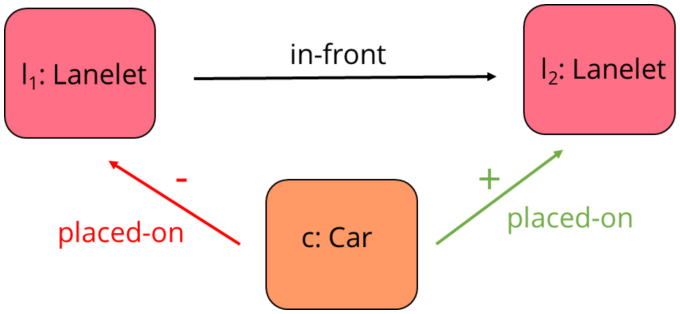


Figure 2.8: The execution of PUT_{place} and PUT_{take} transformations.

Now we can assign the last element of our metamodel $\mathcal{M}_{Traf}$:

$$TR_{Traf} = \{PUT_{take}, PUT_{place}\}$$

Definition 13 (Graph transformations). A graph transformation $\mathrm{X} : G \times 2^{\mathcal{TR}} \to G$ is a function that maps a model to another modified model by executing a set of model transformation rules.

When calling, $\mathrm{X}(\mathcal{G}, TR)$ runs through all the transformation rules and applies them the following way:

If $tr \in TR$ is a:

1. $CREATE = \langle s, d \rangle$

   A new object o_{new} is created and added to the model $\mathcal{G}$:

   $$O_{\mathcal{G}} = O_{\mathcal{G}} \cup \{o_{new}\}$$

   The function sets the interpretations of o_{new} for symbols $s \in S$ to the value of d:

   $$I_{\mathcal{G}}(s)(o_{new}) = d$$

2. $PUT = \langle \phi(\hat{p}), \hat{p}', s, d \rangle$

   $B = \Psi_{\mathcal{G}}(\phi(\hat{p}))$, where $\Psi_{\mathcal{G}}$ is the graph pattern matcher defined on $\mathcal{G}$ returning the binding set satisfying ϕ.

   For every, $\beta \in B$, an object vector $\hat{o}$ is constructed:

   $$\hat{o} = (\beta(p'_1), \ldots, \beta(p'_{\alpha(s)}))$$

   that only contains objects that are bound to the elements of the parameter list $\hat{p}'$ specified in the rule. This also enforces $|\hat{o}| = \alpha(s)$, which is required to set the interpretation for s:

   $$I_{\mathcal{G}}(s)(\hat{o}) = d$$

$\mathcal{G}'$ is produced by first evaluating all the graph queries defined in the transformation rules, and only after the object vectors are constructed, we apply the transformations on $\mathcal{G}$. This ensures that a set of transformations TR given as a parameter is evaluated only on the snapshot of $\mathcal{G}$ before any modifications are made. ▪

Additionally, graph transformations can be annotated with a logical timespan value that tells how much logical time it takes for the transformation to finish. Every time a transformation is applied, a global logical clock is incremented by the timespan of the transformation.

2.4.2 Guided design-space exploration

Definition 14 (Trajectory). We consider a vector of instance graph models $t = (\mathcal{G}_1, \ldots, \mathcal{G}_m)$, a graph *trajectory*. (In the sense of our Traf running example, a traffic scenario can be represented by a graph *trajectory*). ▪

Definition 15 (Dangerous trajectory). A dangerous scenario, in the sense of our Traf running example, is called a *dangerous trajectory* in the graph modeling domain. ▪

Existing DSE approaches usually apply model checking with exhaustive state space exploration or solve finite domain constraint satisfaction problems (CSP) [28, 15]. To explore alternative system designs efficiently, designers use guided model-driven DSE by making use of advanced model-driven techniques (e.g. incremental model transformations) and hints (obtained by analysis tools or provided by the designer)

The guided design space exploration approach is based on a general search process, which traverses the design space starting from the initial state. This general process includes a step (Evaluate criteria), which relies on the guidance and hints provided by system analysis to the different exploration strategies (identify decisions challenge). [22, 1, 18, 3, 36].

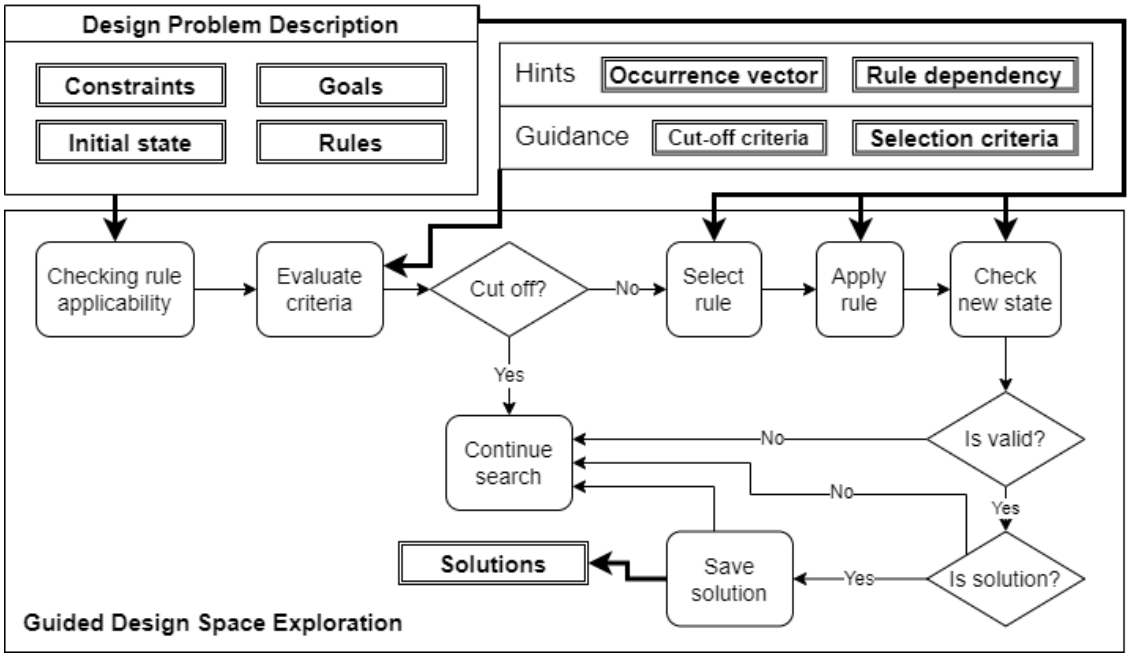


Figure 2.9: Workflow of the guided design space exploration [22].

The search process, shown in Figure 2.9, consists of the following steps:

1. *Check operation applicability* First, labeling rules (of the design problem description) are checked for executability (i.e. whether they can be executed in the current state of the model) and this information is passed to the criteria evaluation.

2. *Evaluate criteria* The cut-off and selection criteria are evaluated using the hints (the rule dependencies and the occurrence vector) and the results are stored.

3. *Cut-off?* If at least one of the cut-off criteria were satisfied during the evaluation, or there are no applicable rules, the state is a dead end and the branch is cut.

4. *Select rule* The DSE engine then selects the next applicable rule based on the evaluation results.

5. *Apply rule* The selected rule is applied to the model, resulting in a new model state.

6. *Check new state* The global constraints and goals are checked on the new state to decide whether it is an invalid or solution state.

   (a) *Is valid state?* If any of the constraint are violated, the state is invalid and the exploration continues from the previous state. Note, that a state is also considered invalid if the exploration has visited it earlier, since in this case the reachable states are already explored from this state.

   (b) *Is solution found?* If all the goals are satisfied, the state is a solution.

7. *Save solution* When a solution model is found, the trajectory (with the executed rules and corresponding model state information) is saved to a solution list.

8. *Continue search* Once the new model state is checked, the next applicable rule is selected from a valid new state, otherwise from the previous state.

Chapter 3

Overview of the approach

In this section, we present our monitor-based guided DSE approach. We first introduce a high-level overview from the perspective of the user of the technique. Secondly, we present the approach componentwise, how different parts are in relation to each other. Then, we go into detail about the three main steps of the approach which are (i) the high-level specification of a problem domain, (ii) the specification of the desired system behaviour and monitoring, and lastly, (iii) the previous two steps end products used by a specific DSE technique to generate test cases in a guided manner.

3.1 Test generation workflow

As our main goal is to effectively generate multiple diverse test sequences for model-based testing, the output of our process is a set of trajectories, each leading to a dangerous scene. In this subsection, we walk through the test generation from the workflow perspective.

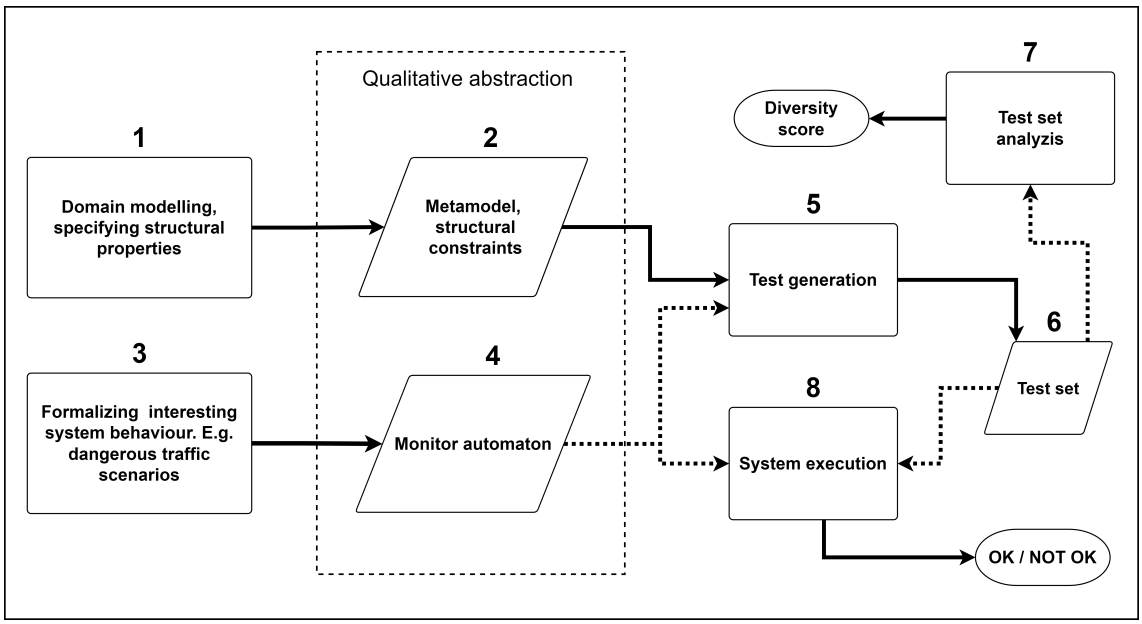


Figure 3.1: High-level overview of our workflow of testing a system.

Figure 3.1 describes the different steps of our process required to generate test scenarios:

1. The engineers model the system under study and define how and under what constraints it can change over time.
2. The abstract metamodel of the system (introduced in Definition 4), an initial instance model, and the transformation rules, will be presented in the next section.
3. Concurrently, engineers formalize the desired abstract behaviour of the system, identifying a system state that is considered undesirable in a production setting, thus, it is necessary to generate test scenarios to evaluate this behaviour.
4. The formalization which can be either written in MFOTL, EPL, or MTGL then gets converted into an executable monitor automaton, suitable to represent this system behaviour.
5. From the structural specification of the system and the monitor, we can generate new model instances that following the state of the monitor can be chained together into trajectories that represent dangerous scenarios.
6. These trajectories are considered as our generated test sequence set representing scenarios leading to dangerous scenes.
7. The generated test cases are analyzed in terms of diversity and state coverage, which results in a diversity score.
8. The system is then tested against these configurations, while monitored to see if the desired behaviour is met during the tests. [26]

> It is important to note that to run the generated test sets in a simulation or other test environment, abstract scenarios need to be concretized. Although model concretization is not a trivial task, one can use existing methodologies to derive concrete executable test scenarios: [36, 5, 28].

3.2 High-level overview

In this section, we present the main components of the approach and how they depend on one another. Figure 3.2 shows the leading figure where the three logical parts are differentiated with colours.

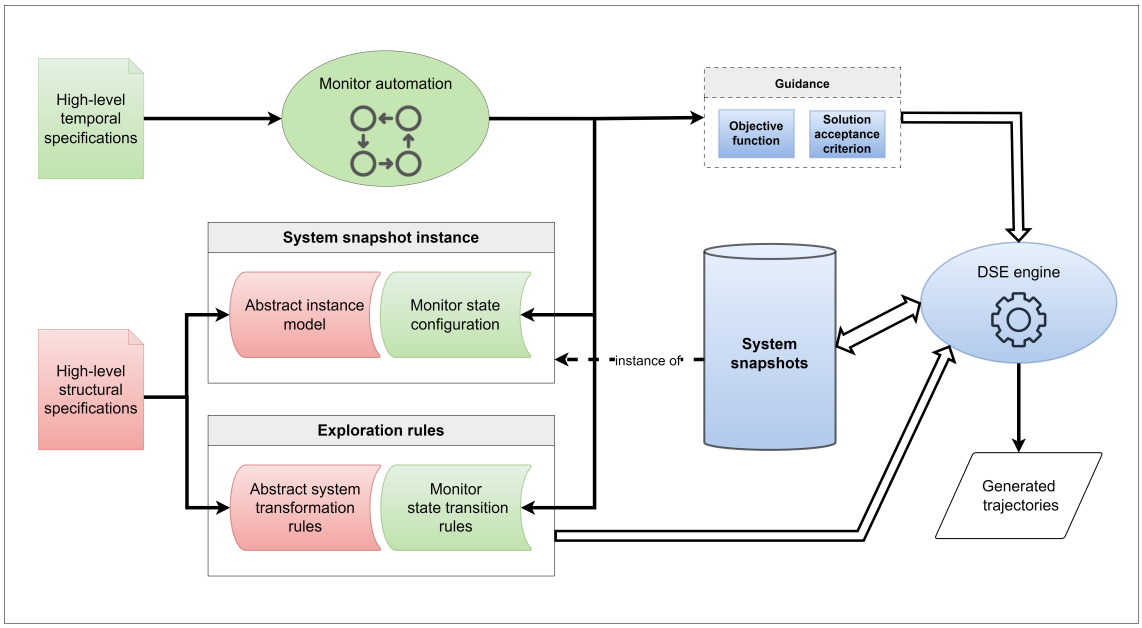


Figure 3.2: High-level overview of the approach.

In the following, we introduce the structural and behavioural specifications of the system domain. This is the part where engineers produce an abstract representation of the system and its behaviour to be tested. The components are coloured with **red** in Figure 3.2.

- *High-level structural specifications:* Abstract metamodel of the system, an initial instance model, and the transformation rules for new instance model generation.
- *Abstract instance model:* Several generated models originated from an initial model and are instantiated from the metamodel of the system.
- *Abstract system transformation rules:* They contain model transformations to generate the next instance model by altering the previous one if the conditions are met.

In the following, we introduce the specification and execution of the monitor automaton. In this part, engineers define the abstract behaviour patterns that lead to dangerous scenarios, which then are converted into an automaton to be executed in generation time and system run-time. The components are coloured with **green** in Figure 3.2.

- *High-level temporal specifications:* User-defined MFOTL, EPL, MTGL, etc. expressions formalizing dangerous scenarios.
- *Monitor automation:* Executable state machine representation of the formalization to detect dangerous scenarios.
- *Monitor state transition rules:* Temporal graph queries that can be executed on the current model state and evoke state transitions if there are any matches. The queries are defined as guards on the monitor's transition.
- *Monitor state configuration:* States of the instantiated, running monitor instances.

In the following, we discuss our monitor-driven design space exploration approach, and how the different blocks are working together to achieve efficient test generation. The components are coloured with **blue** in Figure 3.2.

- *System snapshot instance:* One of all the possible system model instances paired with a corresponding monitor state configuration. A System snapshot instance is generated from a previous instance by the DSE engine using exploration rules.
- *System snapshots:* A global tree of system snapshot instances. The DSE engine uses one of the leaf snapshot instances from the tree to generate the next one. The new instance is then appended to that leaf.
- *Solution acceptance criterion:* A function to decide that a snapshot instance is to be saved or discarded throughout the generation.
- *Guidance:* Information for the DSE engine based on the monitor about which already generated snapshot instances to store and in what order.
- *Objective function:* Describes how to derive a fitness value from the monitor by the DSE to determine which is the best snapshot instance to store next.
- *Exploration rules:* The DSE engine uses Exploration rules to determine which transformation rules to apply to generate the next instance model and which state transitions to fire to produce the next monitor state configuration.
- *DSE engine:* The main component, responsible for executing the automation, applying transformation rules while considering the guidance information, thus generating runnable test sequences.
- *Generated trajectories:* The generated, abstract test sequences are constructed from the system snapshots, and returned by the DSE.

3.3 Structural and behavioural specification of the domain

To generate tests in a model-driven way, our technique requires defining a (i) metamodel $\mathcal{M} \in M$ of the system domain (Definition 3), some (iii) model transformation rules $TR \subseteq \mathcal{TR}$ (Definition 12), to be applied on instance models (Definition 5), and an (ii) initial model $\mathcal{G} \in G$ in which all generated scenes can be originated.

Throughout this process, we are using qualitative abstraction, introduced in Definition 4, to reduce the quantified aspects of the modeled environment, focusing only on the structural aspects.

3.4 Specification of the monitor

Specification formulae Engineers will formulate abstract dangerous system behaviours Definition 2), that will eventually serve as guidance to produce diverse model sequences. These formulations $\theta \in \Theta$ can be constructed from some formal temporal specification languages like MFOTL Section 2.3.2.2, EPL Section 2.3.2.3, MTGL Section 2.3.2.1, etc. The language has to be able to capture structural changes in the model over time to express system behaviours.

Conversion to automaton The next step is to map these formulae representing dangerous scenarios to executable automata representation $\Theta \rightarrow A$.

- One approach is using an event automaton (Section 2.3.1), that is triggered by input events. In this case, a system model has to emit corresponding parametric events representing the changes in the model over time. This is the most suitable approach if we are using some EPL formalism and complex event processing.
- Another approach, which we are implementing in this work, is to directly evaluate the guards on the transitions on the current model state, using graph queries instead of events. This way, we don't have to define events on our models to emit, however, the monitor needs full access to the model state which is also a challenge. With this approach, formulating behaviors can be done with MFOTL formulae.

So far, the conversion between a temporal formula θ and a monitor $\mathcal{A}$ is done manually and will be demonstrated in the next chapter.

3.5 Monitor-driven design space exploration

Common representation As mentioned in the previous section, in order to evaluate graph queries on the system model, the monitor needs to have full access to the system state. This is achieved by integrating the monitor into the system model by creating a common representation or *system snapshot* and executing the monitor with the model transformations from the same domain (*monitor state transition rules*) as the system model. For example, matching a transition query on the system model returned bindings that can be considered as new state tokens. We can use graph transformations (Definition 13) to "remove the token object from the source state" and "place the new tokens into the target state".

The representation is visualized in Figure 3.3, where it is visible that the graph queries ϕ_i, on the edges of the automaton, are evaluated on the system model.

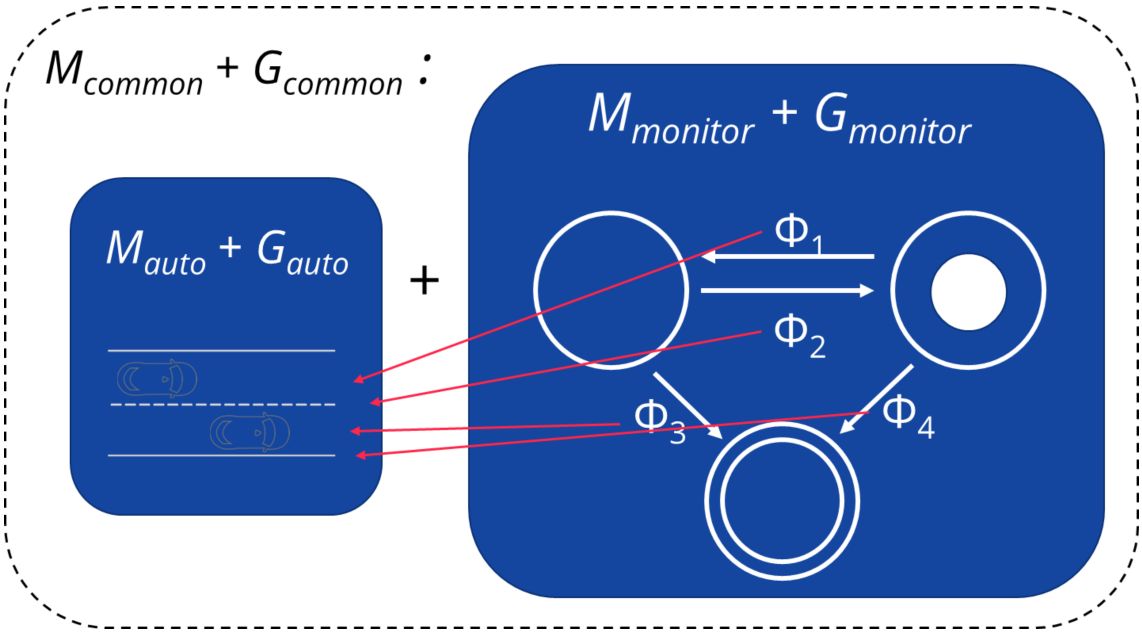


Figure 3.3: The common representation of the monitor $(\mathcal{M}_{mon}, \mathcal{G}_{mon})$ and our system model $(\mathcal{M}_{traf}, \mathcal{G}_{traf})$.

Objective function As new instance models are generated by changes in the system, the monitor's state configuration also changes depending on the evaluation of the transition queries on the model. Assuming that each state of the monitor holds some information about how close the system is to a certain behaviour, defined by θ, a fitness value can be calculated. As a rule of thumb, the fitness value can be calculated by any fitness or *objective function*, that favors state configurations that are closer to an accepting state or states. In the next chapter, we propose a specific fitness function that behaves accordingly.

Acceptance criterion While the objective function helps to decide which new model version is the best among all possibilities, the *acceptance criterion* can detect if we reached a model version that represents a dangerous scene. In other words, we succeeded in generating a trajectory that matches θ. This also can be told based on the current state of the monitor, for example by searching for tokens that are in an accepting state.

Generating trajectories We could use arbitrary search-based DSE strategies that build trajectories based on an objective function and acceptance criterion. As our specific goal is to explore all the dangerous trajectories in the least amount of time and resources, we will use an approach that implements a best-first strategy that finds model versions associated with the best fitness first.

Figure 3.4 shows an example using our AUTO domain, of navigating through model versions and building trajectories in the meantime. More specifically, Figure 3.4a displaying a short exploration where:

1. We start from an initial system and monitor state. The snapshot is marked with "Idle", which means, that the accept criterion is evaluated to false.
2. The DSE uses one of the *system transformation rules* to generate a new model version. As the new version is produced, the DSE applies *monitor state transitions* according to the new system state. As the **RED** dot suggests that in the current state, the acceptance criterion is evaluated to true, the trajectory $\hat{t} = (t_1, t_{2.1})$, is saved to a container named *system snapshots*.
3. The next best solution to explore is shown in step 3, where the monitor's state is one step closer to an accepting state.
4. In Step 4, we also get an accepted state, therefore the trajectory $\hat{t} = (t_1, t_{2.2}, t_{3.1})$ gets saved.
5. In Step 5, a new model version is produced without affecting the fitness value nevertheless, it can still turn out to be an accepted trajectory in the future, so the exploration must continue.
6. The system state at step 6 turns out to be equivalent to the one in step 1 (it can be decided using neighbourhood shapes in Section 4.3.1). This trajectory gets disposed to avoid recursion.

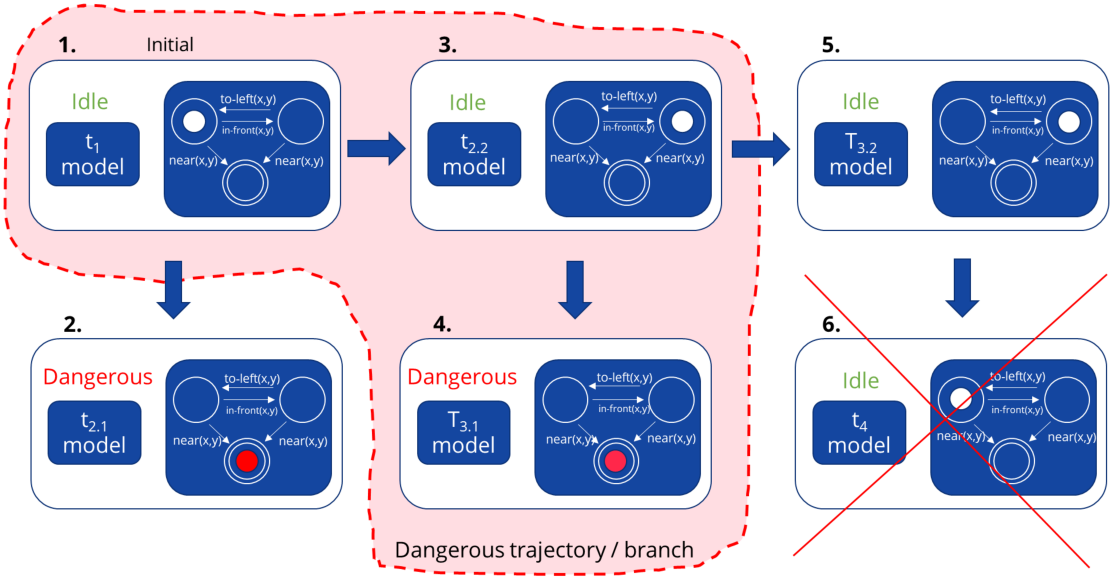
Example 8. *Figure 3.4b shows one of the accepted trajectory* $(t_1, t_{2.2}, t_{3.1})$ *with an initial instance model introduced in Example 3 and some possible transformations on it:*

1. *The ego car switches lanes, forcing a cyclist off the road.*
   (a) *to-left*$(ego, cyclist)$, *placed-on*$(cyclist, l)$, *Actor*$(cyclist)$ *relations disappear from the model.*

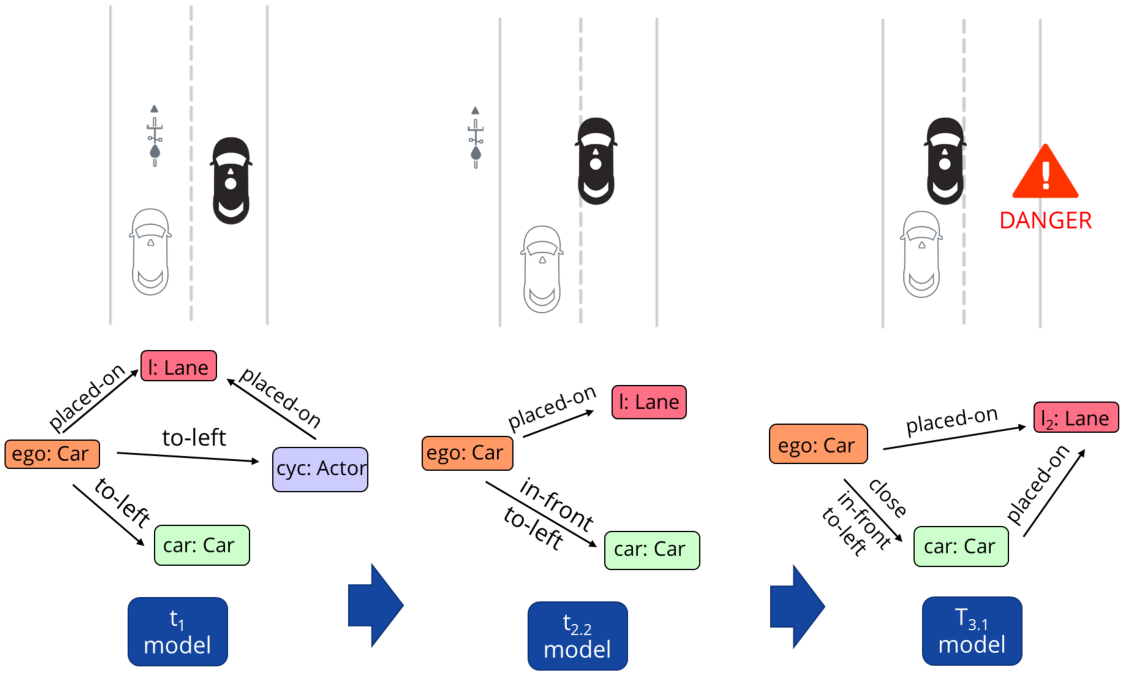
(b) in-front(ego, car) appears in the model.

2. *The ego car slows down in front of another car, causing a dangerous situation.*

   (a) close(ego, car), placed-on(car, l_2) appears in the model.



(a) Monitoring model changes at runtime using graph queries. Trajectories resulting in a dangerous scenario are detected.



(b) Model representation of a dangerous traffic scenario.

Figure 3.4: A possible example of a predefined dangerous scenario. The ego car switches lanes, forcing a cyclist off the road, then slows down in front of another car.

Chapter 4

Implementation

4.1 Specification of the monitor

In this section, we present the usage of MFOTL expressions to define dangerous system behaviours, then propose an automata formalization that is capable of monitoring these behaviours. We then show how we can store and run these monitors in a graph database.

4.1.1 High-level temporal specifications

Now we are specifying an abstract system behaviour based on $\mathcal{M}_{Traf}$, involving two cars c_1 and c_2, where c_1 conducts a dangerous overtaking from the left. We will use MFOTL formulae without any time intervals, however later in the chapter we will show an example of how to introduce time constraints to the same expression.

First, let us define some helper formulae:

- A car is close to and in front of another car (Their distance is 1 lanelet):
  $\theta_{\text{close-in-front}}(c_1, c_2) = \exists l_1, l_2 : (\text{in-front}(l_1, l_2) \wedge \text{placed-on}(c_1, l_1) \wedge \text{placed-on}(c_2, l_2))$
- A car is mid-distance to and in front of another car (Their distance is 2 lanelets):
  $\theta_{\text{mid-in-front}}(c_1, c_2) = \exists l_1, l_2, l_3 : (\text{in-front}(l_1, l_2) \wedge \text{in-front}(l_2, l_3) \wedge \text{placed-on}(c_1, l_1) \wedge \text{placed-on}(c_2, l_3))$
- A car is close and left to another car (Their distance is 1 lanelet):
  $\theta_{\text{close-to-left}}(c_1, c_2) = \exists l_1, l_2 : (\text{to-left}(l_1, l_2) \wedge \text{placed-on}(c_1, l_1) \wedge \text{placed-on}(c_2, l_2))$

Now use these formulae to define a behaviour by the relative position of these cars:

$$\begin{aligned}
\theta_{overtake} = &\exists c_1, c_2 : (\theta_{\text{mid-in-front}}(c_2, c_1) \rightarrow \Diamond(&& (4.1)\\
&\theta_{\text{close-in-front}}(c_2, c_1) \rightarrow \Diamond(&& (4.2)\\
&\theta_{\text{close-to-left}}(c_1, c_2) \rightarrow \Diamond(&& (4.3)\\
&\theta_{\text{close-in-front}}(c_1, c_2) \wedge \neg\theta_{\text{close-in-front}}(c_2, c_1))))) && (4.4)
\end{aligned}$$

The starting state is where the two car is mid-distance to each other and c_2 is in front of c_1 (4.1). Then in the next time unit, c_1 got close to c_2 (4.2). After that c_1 starts the overtake by merging into the left lane (4.3). In the end, c_1 merges back to the right in front of but still close to c_2 and completes the overtake (4.4).

4.1.2 Monitor formalization

For expressing temporal specifications of dangerous scenarios (Definition 2) as a run-time executable monitor automaton, we use an extension of the PTA formalism introduced in Section 2.3.1. To do this, will also reuse graph modeling elements (Section 2.2) as we propose a state machine formalization that is able to fire state transitions using graph predicates as transition guards.

Definition 16 (Monitor automaton). Based on Definition 8, let it be the algebraic representation of our monitor automaton a tuple:

$$\mathcal{A} = \langle Q, q_0, F, P, \Phi, \mathrm{T}, \Delta \rangle$$

where:

- Q: A finite set of states.
- q_0: The initial state $q_0 \in Q$.
- F: The set of accepting states $F \subseteq Q$.
- P: A finite set of object parameters.
- Φ: A finite set of graph queries defined over $\mathcal{M}_{Traf}$ and object parameters P.
- T: A set of clock objects, each mapped to the last global timestamp in milliseconds.
- $\Delta \subseteq Q \times \Phi \times 2^P \times 2^{\mathrm{T}} \times Q$: A transition relation. An element $(q, \phi, B, C, q') \in \Delta$ indicates a transition from state q to state q' on the satisfaction of a graph query $\phi \in \Phi$ with a sized parameter set $B \subseteq P$ and clock object set $C \subseteq \mathrm{T}$ to be reset. ▪

Semantics For a graph query ϕ and a transition $(q, \phi, B, C, q') \in \Delta$, the transition can be taken if matching ϕ on a graph model $\mathcal{G}_{Traf}$ with the matching function Ψ_{Traf}, $\Psi_{Traf}(\phi) \neq \emptyset$. In other words, a transition can be fired if the guard on the transition can be matched with at least one parameter binding on the model $\mathcal{G}_{Traf}$. This can be done because all $\phi \in \Phi$ is defined over the same metamodel $\mathcal{M}_{Traf}$ as $\mathcal{G}_{Traf}$.

Given $\mathcal{A} = \langle Q, q_0, F, P, \Phi, \mathrm{T}, \Delta \rangle$, the system can move to a new state q' upon the satisfaction of a graph query ϕ if and only if there exists a transition $(q, \phi, B, C, q') \in \Delta$ and a binding β such that for all parameters $b_i \in B$:

$$\beta(b_i) \in O_{\mathcal{G}}$$

The values bound to the parameters in binding β are objects from model $\mathcal{G}_{Traf}$. In case of timed behaviour, upon the firing of a state transition, all clock objects $\in C$ are mapped to a logical timestamp of the firing of the transition.

4.1.3 Model representation of the monitor automaton

We would like to store the state and signature of a monitor automaton in a graph database, in order to be able to merge the model of the automaton with a graph model G, representing the system under study. This way, state transitions can be executed by applying graph transformation rules defined on model G using the queries on the transitions.

Let us define a metamodel $\mathcal{M}_{mon} = \langle \Sigma_{mon}, \alpha_{mon}, TR_{mon} \rangle$ for the monitor $\mathcal{A} = \langle Q, q_0, F, P, \Phi, \mathrm{T}, \Delta \rangle$ where:

- TR_{mon} is a set of transformation rules extending the metamodel.
- $\Sigma_{mon} \subseteq Q \times 2^P$, $s = \langle q, \hat{p} \rangle \in \Sigma$, $\alpha_{mon}(s) = |\hat{p}|$. Where each symbol is represented by a tuple s that contains a monitor state q, and a parameter vector $\hat{p}$.

Conversion logic We map every state and all the possible parameter configurations in that state to a symbol in Σ_{mon}, where the arity of each symbol is the number of parameters that could possibly be bound to an object value in that state. Important to note that $\Sigma_{mon} \cup \Sigma_{Traf} = \emptyset$ must hold, where Σ_{Traf} is the set of symbols defined for the model on which we match every ϕ.

We map every transition $(q, \phi, B, C, q') \in \Delta$ to a set of graph transformation rules as:

- We initialize TR_{mon} with $\emptyset$.
- Then we extend ϕ to also check if tokens are present in q with bound parameters p:

  $$\phi_{ext}(\hat{p}) := \phi(\hat{p}) \wedge \langle q, \hat{p} \rangle(\hat{p})$$

  Note that $\langle q, \hat{p} \rangle$ here is a symbol we defined earlier identifying a state q and a vector of parameters $\hat{p}$, that could possibly be bound to an object value in that state.

- We assign transformation rules for every symbol with different bound parameter vector $\hat{p}$, where the state is q:

  For all $s = \langle q, \hat{p} \rangle$, where $\hat{p} \subseteq P$:

  Let us take the symbol corresponding to the next state q' and a new (extended) vector of bound parameters, $\hat{p}_{ext} = (p_1, p_2, \dots, B_1, B_2, \dots)$ as in the new state we have the newly bound parameters from B along with the already bound parameters from $\hat{p}$.

  $$s = \langle q', \hat{p}_{ext} \rangle$$

  We assign a transformation rule that removes the interpretation for s on object vectors, if $\phi_{ext}(\hat{p}_{ext})$ is satisfied. Intuitively, this rule removes a token from the source state in case of a transition firing.

  $$PUT_{from} := \langle \phi_{ext}(\hat{p}_{ext}), \hat{p}, s, 0 \rangle$$

  Now we assign a transformation rule that adds the interpretation for s_{ext} on object vectors, if $\phi_{ext}(\hat{p}_{ext})$ is satisfied. Intuitively, this rule adds a token to the target state in case of a transition firing.

  $$PUT_{to} := \langle \phi_{ext}(\hat{p}_{ext}), \hat{p}_{ext}, s_{ext}, 1 \rangle$$

  Finally, we extend the monitor's set of transformation rules TR_{mon}, which can be executed upon a monitor synchronization.

  $$TR_{mon} = TR_{mon} \cup \{PUT_{from}, PUT_{to}\}$$

  The execution of TR_{mon} is done by calling $\mathrm{X}(\mathcal{G}_{mon}, TR_{mon})$ (Definition 13). First, every transition query is matched so previous transformations won't affect future

query matches. Secondly, all transformations are applied with their corresponding bindings.

Example 9. *In Figure 4.1a, we present an example monitor with some parameter bindings in different states:*

overtakeStarted	*overtakeSucceeded*	*overtakeCanceled*
$car_1 \rightarrow$ *001-xxx*	$car_1 \rightarrow$ *003-yyy*	$car_1 \rightarrow$ *004-qqq*
$car_2 \rightarrow$ *002-zzz*	$car_2 \rightarrow$ *002-zzz*	$car_2 \rightarrow$ *002-zzz*

The automaton monitors an overtake of two cars. In the presented state configuration, "001-xxx" is currently overtaking "002-zzz" while "003-yyy" have already overtaken "002-zzz" and "004-qqq" failed the overtake.

We can map this automaton state configuration to a graph model by applying the described mechanism.

1. *Construct the symbol set from the states and all possible parameter vectors that can be bound in them:*

$$\begin{aligned}\Sigma = \{&\langle intial, ()\rangle, && s_0^{()}\\ &\langle initial, (car_1, car_2)\rangle, && s_0^{(car_1,car_2)}\\ &\langle overtakeStarted, (car_1, car_2)\rangle, && s_1^{(car_1,car_2)}\\ &\langle overtakeSucceeded, (car_1, car_2)\rangle, && s_2^{(car_1,car_2)}\\ &\langle overtakeCanceled, (car_1, car_2)\rangle\}. && s_3^{(car_1,car_2)}\end{aligned}$$

   Note that initial state can also be bound with 2 parameters and zero, thus the state transitions are also different in the two cases. We denoted the symbol signatures with $s_{stateId}^{parameters}$ *for simplicity.*

2. *The objects of the model are the values bound to the parameters of our queries. With the symbols above, we can now specify the relationships between the objects in this state configuration shown in Figure 4.1b:*

$$\begin{aligned}&I(s_1^{(car_1,car_2)})(\textit{001-xxx}, \textit{002-zzz}) = 1\\ &I(s_2^{(car_1,car_2)})(\textit{003-yyy}, \textit{002-zzz}) = 1\\ &I(s_3^{(car_1,car_2)})(\textit{004-qqq}, \textit{002-zzz}) = 1\end{aligned}$$

3. *We define the transformation rules of the state transitions. TR:*

   (a) $s_0^{()} \rightarrow s_0^{(car_1,car_2)}$

$$\begin{aligned}&\langle s_0^{()} \wedge \textit{in-front}(car_2, car_1), (), s_0^{()}, 0\rangle\\ &\langle s_0^{()} \wedge \textit{in-front}(car_2, car_1), (car_1, car_2), s_0^{(car_1,car_2)}, 1\rangle\end{aligned}$$

   (b) $s_0^{()} \rightarrow s_1^{(car_1,car_2)}$

$$\begin{aligned}&\langle s_0^{()} \wedge \textit{to-left}(car_1, car_2), (), s_0^{()}, 0\rangle\\ &\langle s_0^{()} \wedge \textit{to-left}(car_1, car_2), (car_1, car_2), s_1^{(car_1,car_2)}, 1\rangle\end{aligned}$$

   (c) $s_0^{(car_1,car_2)} \rightarrow s_1^{(car_1,car_2)}$

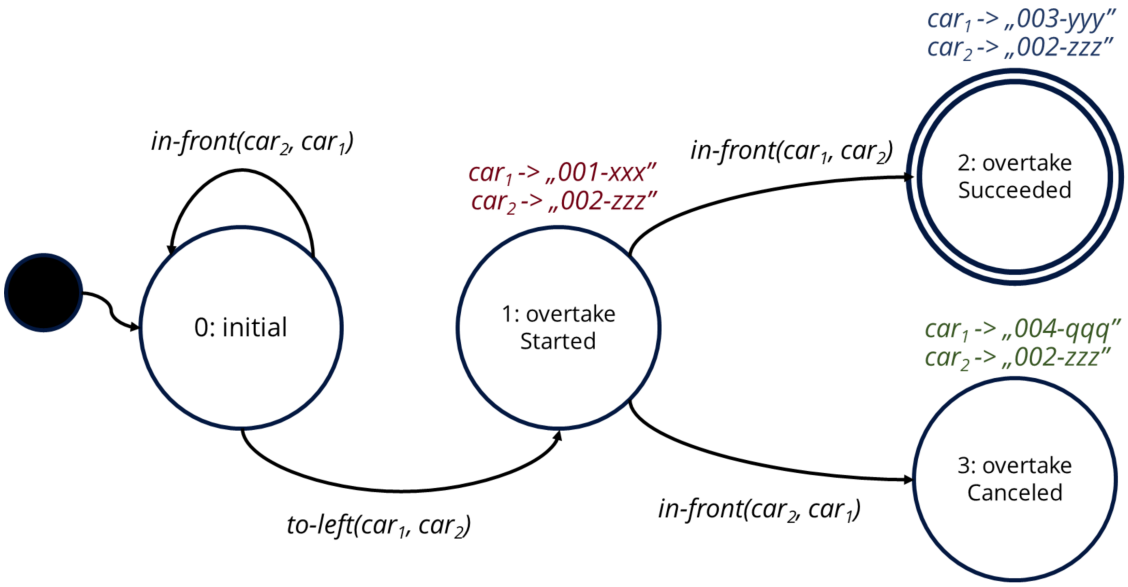
$$\langle s_0^{(car_1,car_2)} \wedge \textit{to-left}(car_1, car_2), (car_1, car_2), s_0^{(car_1,car_2)}, 0\rangle$$
$$\langle s_0^{(car_1,car_2)} \wedge \textit{to-left}(car_1, car_2), (car_1, car_2), s_1^{(car_1,car_2)}, 1\rangle$$

(d) $s_1^{(car_1,car_2)} \rightarrow s_2^{(car_1,car_2)}$

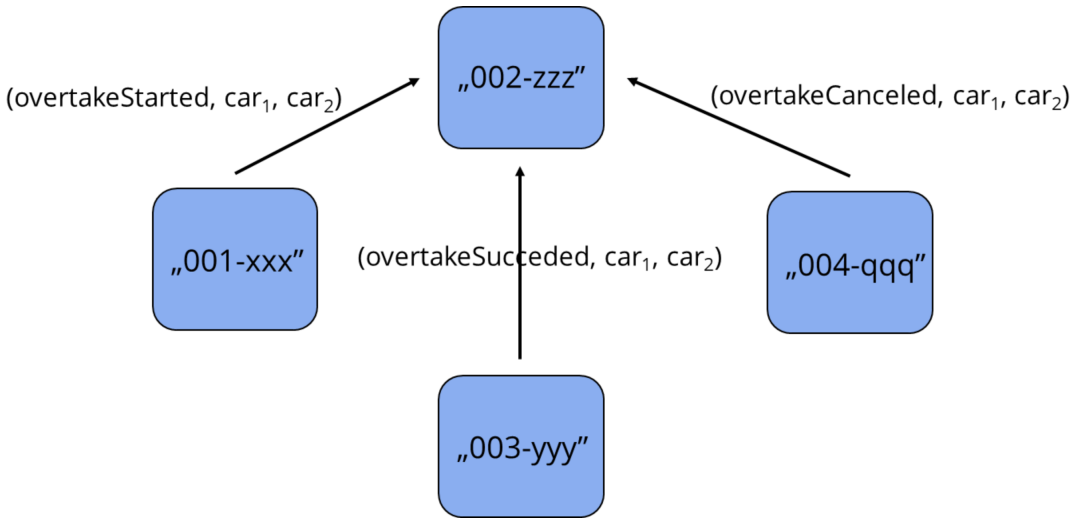
$$\langle s_1^{(car_1,car_2)} \wedge \textit{in-front}(car_1, car_2), (car_1, car_2), s_1^{(car_1,car_2)}, 0\rangle$$
$$\langle s_1^{(car_1,car_2)} \wedge \textit{in-front}(car_1, car_2), (car_1, car_2), s_2^{(car_1,car_2)}, 1\rangle$$

(e) $s_1^{(car_1,car_2)} \rightarrow s_3^{(car_1,car_2)}$

$$\langle s_1^{(car_1,car_2)} \wedge \textit{in-front}(car_2, car_1), (car_1, car_2), s_1^{(car_1,car_2)}, 0\rangle$$
$$\langle s_1^{(car_1,car_2)} \wedge \textit{in-front}(car_2, car_1), (car_1, car_2), s_3^{(car_1,car_2)}, 1\rangle$$



(a) An example monitor state configuration with bound parameters.



(b) Graph model representation of the current state of the monitor in Figure 4.1a.

Figure 4.1: An example for converting a monitor automaton into a graph representation that is compatible with our metamodel in Example 2.

4.1.4 Instantiating the monitor

Let us have $\mathcal{G}_{mon} = \langle O_{mon}, I_{mon} \rangle$ defined over $\mathcal{M}_{mon}$. We define an initial model configuration that is the same for every automaton, consisting of one object, representing the token in the initial state q_0.

- $O_{mon} = \{initial\}$, the object representing the token in the initial state.
- Now we add the interpretation to *initial* for the symbol $s_0 = \langle q_0, () \rangle$:

$I_{mon}(s_0)(initial)$	$= 1$	Meaning that we have a token in q_0 with an empty vector of bound parameters.

4.2 Monitor-driven design space exploration

In this section, we discuss the implementation of the *system snapshots* from the independent system model and monitor. We define our proposed objective function and *acceptance criterion* and provide an explanation of how we handle time intervals included in dangerous scenario specifications. In the end, we discuss how these different components are used to derive abstract dangerous trajectories by running DSE with heuristics.

4.2.1 Common system-monitor representation

To be able to drive the monitor automaton, matching its transition queries on the system model, we merge the model representation of the monitor $(\mathcal{M}_{mon}, \mathcal{G}_{mon})$ and our system model $(\mathcal{M}_{Traf}, \mathcal{G}_{Traf})$ into a common representation $(\mathcal{M}_{com}, \mathcal{G}_{com})$ the following way:

1. We take a common metamodel $\mathcal{M}_{com} = \langle \Sigma_{com}, \alpha_{com}, TR_{com} \rangle$ where:
   - $\Sigma_{com} = \Sigma_{Traf} \cup \Sigma_{mon}$
   - $\alpha_{com}(s) = \begin{cases} \alpha_{mon}(s) & \text{if } s \in \Sigma_{mon} \\ \alpha_{Traf}(s) & \text{if } s \in \Sigma_{Traf} \end{cases}$
   - $TR_{com} = TR_{Traf} \cup TR_{mon}$. Note that applying state transition transformations will not affect the system model. This is because every transformation rule that we previously defined to fire state transitions only contains symbols from Σ_{mon} and $\Sigma_{mon} \cup \Sigma_{Traf} = \emptyset$.
2. We specify the common model $G_{com} = \langle O_{com}, I_{com} \rangle$ defined on metamodel signature $\langle \Sigma_{com}, \alpha_{com}, TR_{com} \rangle$:
   - $O_{com} = O_{Traf} \cup O_{mon}$
   - $I_{com}(s)(O) = \begin{cases} I_{mon}(s)(O) & \text{if } s \in \Sigma_{mon} \text{ and } O \text{ is a tuple,} \\ & \text{where } O_i \in O_{mon} \\ I_{Traf}(s)(O) & \text{if } s \in \Sigma_{Traf} \text{ and } O \text{ is a tuple,} \\ & \text{where } O_i \in O_{Traf} \end{cases}$

The representation is visualized in Figure 3.3, where it is accentuated that the graph queries on the edges of the automaton are evaluated on the system model and not on the monitor.

4.2.2 Running the exploration

At this point, we introduced all components that are necessary to generate a sufficient trajectory set where in all trajectories, a certain dangerous system behaviour has been detected.

Throughout the exploration, for selecting the next unexplored decision to refine, we use a combined exploration strategy with best-first search heuristic, backtracking, back jumping, and random restarts with an advanced design space exploration framework [22, 35, 4]. We denote this strategy as a function:

$$strategy(t_{com}, TR_{com}, \mathcal{F}_{fitness}) \rightarrow G_{com}$$

where:

- t_{com}: A trajectory as a vector of $\mathcal{G}_{com} \in G_{com}$. Based on the already generated models, the DSE can decide how to produce the next model. These models are common representations of the system and the monitor.
- TR_{com}: A set of transformation rules that can be applied on $\mathcal{G}_{com}$.
- $\mathcal{F}_{fitness}$: The fitness function.

We define our exploration function as the following:

Definition 17 (Monitor guided exploration).

$$EXPLORATION(\mathcal{M}_{com}, \mathcal{G}_{com}, \mathcal{F}_{fitness}, \mathcal{F}_{accept}, n, maxLength) \rightarrow 2^{G^n}$$

where:

- $\mathcal{M}_{com}$: The metamodel constructed from $\langle \Sigma_{mon}, \alpha_{mon}, TR_{mon} \rangle$ and $\langle \Sigma_{Traf}, \alpha_{Traf}, TR_{Traf} \rangle$.
- $\mathcal{G}_{com}$: The initial model constructed from $\mathcal{G}_{mon}$ and $\mathcal{G}_{Traf}$.
- $\mathcal{F}_{fitness}$: The fitness function.
- $\mathcal{F}_{accept}$: The acceptance criterion predicate.
- $n \in \mathbb{N}^+$: The number of trajectories to generate.
- $maxLength \in \mathbb{N}^+$: A threshold criterion for trajectory length. Based on this number, we can prevent infinite-length trajectories.

Algorithm 1 Executing guided design space exploration

```
1: function EXPLORATION(M_com, G_com, F_fitness, F_accept, n, maxLength)
2:     Solutions ← ∅
3:     while |Solutions| < n do
4:         G ← G_com
5:         traj ← (G_com)
6:         while F_accept(traj) ≠ 1 ∧ |traj| < maxLength do
7:             G ← strategy(traj, TR_Traf, F_fitness)
8:             Insert G at end of traj
9:         if F_accept(traj) = 1 then
10:            Solutions ← Solutions ∪ traj
       return Solutions
```

Algorithm 1 shows a simplified description of the implementation:

1. In line 2 we initialize our solution trajectories with an empty set.
2. Line 3 is where our main loop starts. While the number of our solutions hasn't reached the required number n, the algorithm keeps generating new trajectories.
3. One by one, we are building up trajectories between lines 6 and 9. If a trajectory is not yet accepted, and it hasn't reached the maximum length criterion, a new model $\mathcal{G}$ is appended to it.

4. In line 7, we use the predefined best-first heuristic to derive a new model from the previous models. Inside the strategy, the given graph transformation rules, one of TR_{Traf} and then all of TR_{mon} are applied to produce a new model, where the monitor contained in the model is the closest possible to an accepting state, in other words, the fitness function is minimal (Definition 18).

5. If a trajectory satisfies our $\mathcal{F}_{accept}$ acceptance criterion predicate, it gets included in the solution set (Lines 9-10). ▪

4.2.3 Fitness function

During the generation, for every monitor state configuration, we need to calculate a fitness value that is going to represent how "close" the monitor is to matching the dangerous behaviour on the system, we defined earlier.

We propose a function $\mathcal{F}_{fitness} : G_{com} \to \mathbb{R} \cap (0, 1]$, that maps a merged model $\mathcal{G}_{com}$ a merge of a monitor automaton and a system model to a real number between 0 and 1:

Definition 18 (Fitness function). At this point, we extend our automation semantics $\mathcal{A} = \langle Q, q_0, F, P, \Phi, \mathrm{T}, \Delta \rangle$ with a $W : Q \to \mathbb{R}_{\geq 0}$, weight function, mapping a non-negative real numeric value to every state of the automaton. Intuitively, a larger weight means "the monitor is closer to an accepting state". $W(q_0) := 0$, implying that a monitor is the furthest away from matching the defined behaviour.

Let us have:

- $\mathcal{A} = \langle Q, q_0, F, P, \Phi, \mathrm{T}, \Delta, W \rangle$, An automaton with the newly defined weight function.
- $\mathcal{M}_{mon} = \langle \Sigma_{mon}, \alpha_{mon}, TR_{mon} \rangle$, the metamodel generated from the automaton.
- Ψ_{com}, graph query matcher defined on model $\mathcal{G}_{com}$

$\mathcal{F}_{fitness} : G_{com} \to \mathbb{R} \cap (0, 1]$:

1. First, we calculate a weighted sum by taking every symbol in Σ_{mon} with the signature $\langle q, \hat{p} \rangle$, where q is a state $\in Q$ and $\hat{p}$ is a vector of bound parameters, and matching them on $\mathcal{G}_{com}$ with Ψ_{com}. The sum of the number of bindings for that symbol times $W(q)$, the weight assigned to q in the automaton gives a weighted sum with the intuition of "The more tokens we have in automaton states with larger weights, the larger number we get.":

$$weightedSum = \sum_{s=\langle q,\hat{p} \rangle \ \in \Sigma_{mon}} |\Psi_{com}(s(\hat{p}))| \times W(q)$$

2. Second, we normalize this number to the interval $(0, 1]$, in the following way:

$$\mathcal{F}_{fitness}(\mathcal{G}_{com}) = \frac{1}{weightedSum + 1}$$

So reversing the intuition: "The more tokens we have in states with larger weights, we get a fitness value closer to 0". This is exactly what we want because, for our exploration strategy, we will favor the model producing the smallest fitness value. ▪

4.2.4 Acceptance criterion

The acceptance criterion is a predicate that decides if we should add a trajectory to our solution set or not. As we are generating dangerous trajectories (Definition 15) that are detected by the monitor, we want to accept only those trajectories where the monitor is in an accepting state for the last model $\mathcal{G}_{com}$ of the trajectory.

Definition 19 (Acceptance predicate). We propose a predicate function $\mathcal{F}_{accept} : G^n_{com} \to \{0, 1\}$, that takes a trajectory $\hat{\mathcal{G}} = (\mathcal{G}_1, \ldots, \mathcal{G}_n)$ and maps it to a truth value, 1 if the trajectory is accepted and 0 if it is not.

Let us have:

- $\mathcal{G}_n \in \hat{\mathcal{G}}$, the last model of the trajectory.
- $\mathcal{A} = \langle Q, q_0, F, W, P, \Phi, \mathrm{T}, \Delta \rangle$, its original automaton.
- M_{com} = The metamodel constructed from $\langle \Sigma_{mon}, \alpha_{mon}, TR_{mon} \rangle$ and $\langle \Sigma_{Traf}, \alpha_{Traf}, TR_{Traf} \rangle$.
- Ψ_n, graph query matcher defined on model $\mathcal{G}_n$.

$$\mathcal{F}_{accept}(\mathcal{G}_{com}) = \begin{cases} 1 & \text{if } \exists q, \hat{p} : q \in F, s = \langle q, \hat{p} \rangle \in \Sigma_{mon}, |\Psi_n(s(\hat{p}))| > 0 \\ 0 & \text{otherwise} \end{cases}$$

The function takes all accepting states $q \in F$ and checks, for every symbol $s \in \Sigma$ containing q, whether there is any parameter configuration $\hat{p}$, that can be matched for s. If there is any match for an accepting state, the function returns true, otherwise false. Intuitively, it checks whether there exists any token in an accepting state. ▪

4.2.5 Extension with time-dependent behaviour

So far, we introduced monitors that are able to reason about model transformation behaviours by transitioning between states when a specific model configuration is matched. Now we specify how we implement transitions that depend on timing for engineers to be able to define behaviours like

"An overtaking between two cars, that is conducted in minimum 4, but maximum 15 seconds":

$$\begin{aligned} \theta_{overtake} = & \exists c_1, c_2 : (\theta_{\text{mid-in-front}}(c_2, c_1) \to \Diamond_{[4000,15000]}(\\ & \theta_{\text{close-in-front}}(c_2, c_1) \to (\\ & \theta_{\text{close-to-left}}(c_1, c_2) \to (\\ & \theta_{\text{close-in-front}}(c_1, c_2) \wedge \neg \theta_{\text{close-in-front}}(c_2, c_1))))) \end{aligned}$$

When initializing a model $\mathcal{G}_{Traf}$, let us include a clock object in our system model by setting the interpretation:

$$I_{Traf}(Clock)(clock) = d, clock \in O_{Traf}$$

where:

- $Clock \in C \in \Sigma$: A unary symbol. Here, we define a new domain for $Clock$ other than $\{0, 1\}$: $dom(Clock) = \mathbb{N}$ (Definition 3). The value from dom represents a system or simulated time measured in milliseconds.
- d: An initial value of our clock at the end of the generation. In this case, consider this 0.

The global clock object's value is refreshed by a transformation rule $PUT_{refresh}$ with a predefined frequency with a timestamp T coming from an outside clock:

$$PUT_{refresh} = \langle Clock(clock), (clock), Clock, T \rangle$$

$$TR_{Traf} = TR_{Traf} \cup \{PUT_{refresh}\}$$

This step is necessary because in order to query the clock in a state transition guard, for instance, the clock needs to be up-to-date. A graph query does not take any time value as a parameter, so it can only query the clock if it is stored in the database.

Now we change the domain of every symbol $s = \langle q, \hat{p} \rangle$ in $\mathcal{M}_{mon}$:

$$dom(s) = 2^{\mathrm{T} \to \mathbb{N}}$$

where each value represents a set of bindings $\tau(c) = t, c \in \mathrm{T}, t \in \mathbb{N}$. And here T is the set of clock objects defined in $\mathcal{A}$. Now, in our model, $\mathcal{G}_{mon}$ every interpretation for each symbol (state) and every object binding will carry information about multiple clocks in the monitor. and this information will mean that "at what time, each clock was set to the time of the global clock ($Clock()$)". We can use this information in our graph queries with some syntactic extensions, to construct logic expressions on the passage of time since the last "clock reset". For example, this could be done like so: Take the monitor symbols with the following signature:

$$s = \langle q, \hat{p} \rangle$$

Introduce a new predicate that takes a symbol, a clock, and a time value as a parameter:

$$TC_{greater}(\langle q, \hat{p} \rangle, \tau, t) \Leftrightarrow \begin{cases} 1 & \text{if } \exists clock : I_{mon}(Clock)(clock) - I_{mon}(s)(\beta(\hat{p}))(\tau) > t \\ 0 & \text{otherwise} \end{cases}$$

Where $\beta(\hat{p})$ is a possible object binding. Finally, we define a query, where we check for bindings where the last reset was more than 5 milliseconds ago.

$$\phi_{>5}(\hat{p}) := TC_{greater}(s, \tau, 5)$$

The last thing is to define actions that can be specified in the monitor to "reset" certain clocks. Let us have a transition $(q, \phi, B, C, q') \in \Delta$ from $\mathcal{A}$. Let the clock object set C represent all the clocks that are to be reset upon the firing of the transition. Transformation rules can be added to TR_{mon} to implement this behaviour similarly to state transitions in Section 4.1.3.

4.3 Analyzing generated trajectories

In this section, our goal is to make us able to draw some conclusions on the resulting trajectory set, besides knowing that they represent the system behaviour that we would like to test. The metric we chose to observe, which we call the diversity of trajectory sets,

is a value that implies the average distance between trajectories. First, we introduce some existing techniques like neighbourhood shapes and Jaccard Similarity coefficient, which we used to derive the distance metric, then, we explain the metric itself.

4.3.1 Neighbourhood shapes of graphs

The internal local structures of a model $\mathcal{G} \in G$ are called neighbourhoods. One can derive several diversity metrics [37], by classifying different ranges of neighbourhoods in $\mathcal{G}$.

Definition 20 (Neighbourhood Range).

The neighbourhood range means that two objects (nodes) are considered identical if they are indistinguishable within a distance of navigations or hops. [31] ▪

Definition 21 (Model shapes). Let us define a function based on [37], that maps a set of neighbourhood shapes up to k navigations to a model $\mathcal{G}$:

$$N^k : G \to \{nhb_1^1, \ldots, nhb_n^1, \ldots, nhb_1^k, \ldots, nhb_m^k, \}$$

where nhb_k^i is the neighbourhood for an object $o_i \in O_\mathcal{G}$ with the range of k. Respectively n is the number of neighbourhoods with range 1 in the model (basically every $o \in O_\mathcal{G}$, where $C(o)$ is evaluated to true), and m is the number of neighbourhoods with range k. ▪

Implications for Model Diversity A model with higher internal diversity would have a greater variety of unique local structures (a larger set of $N^k(\mathcal{G})$), suggesting that the model is rich in its representations. On the contrary, a model with low internal diversity would indicate repetitive or homogeneous local structures.

4.3.2 Jaccard similarity coefficient

The Jaccard coefficient, often denoted as $J(A, B)$, is a widely used metric to assess the similarity between two finite sample sets. By comparing the size of the intersection of the sets to the size of their union, this coefficient offers a value between 0 and 1; where a value of 0 indicates no overlap and 1 represents complete similarity (i.e., both sets are identical) [30].

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

4.3.3 Introducing diversity metric

We introduce a modified version of the neighbourhood-based internal diversity metric (Section 4.3.1), to derive the structural diversity of generated trajectory sets. Neighbourhoods are also used in [3] for this purpose. This metric evaluates the average diversity by comparing the neighbourhood structures of the respective graph models within two trajectories. To determine the total diversity of a trajectory set, we compute the average of the pairwise diversities across the entire set.

Definition 22 (Diversity metric for trajectory set). Let $T = \{t_1, t_2, \ldots\}$ be the trajectory set we want to calculate the diversity of. Let $N^k(t^i)$ represent the neighbourhood structures of the i^{th} model in a trajectory t up to k navigations. Let $s(t)$ represent the size of a trajectory t. Let MAX represent the maximum trajectory size in set T.

1. For each trajectory $t \in T$, we extend t such that:

$$s(t) \leftarrow MAX$$

by copying the last model and appending it until the desired size is achieved.

2. For each model in trajectories $t^i \in t \in T$, we classify $N^3(t^i)$ up to 3 navigations.

3. To compare the distance of two trajectories, $t_1 \in T, t_2 \in T, t_1 \neq t_2$ we take all $t_1^i \in t_1$ and all $t_2^i \in t_2$ and calculate the average pairwise Jaccard similarity coefficient introduced in 4.3.2 of the models with the same index using $N^3(t_1^i)$ and $N^3(t_2^i)$.

$$d_{trajectory}(t_1, t_2) = \frac{\sum_{i=1}^{MAX} \frac{|N^3(t_1^i) \cap N^3(t_2^i)|}{|N^3(t_1^i) \cup N^3(t_2^i)|}}{MAX}$$

4. Then, in order to derive the final similarity value, we take the average pairwise similarity of all trajectories in T:

$$similarity(T) \Leftrightarrow \begin{cases} 0 & \text{if } |T| < 2 \\ \sum_{i=1}^{n} \sum_{j=i+1}^{n} d_{trajectory}(t_i, t_j) & \text{otherwise} \end{cases}$$

This metric gives an intuition on how similar the trajectories are in a trajectory set. If the calculated value is large, that means we have rather similar trajectories. In the following, we demonstrate the formula through an example.

Example 10. *Suppose we have a trajectory set T consisting of two trajectories:*

$$T = \{t_1, t_2\}$$

Where:

$$t_1 = \{m_1^1, m_1^2\}$$
$$t_2 = \{m_2^1, m_2^2, m_2^3\}$$

Given the trajectories, we have:

$$MAX = 3$$

1. *Extend each trajectory in T to the size of MAX.*

$$t_1 \leftarrow \{m_1^1, m_1^2, m_1^2\}$$
$$t_2 \leftarrow \{m_2^1, m_2^2, m_2^3\}$$

(Note: t_1 was extended by copying m_1^2 once.)

2. *Define the neighbourhood structures up to 3 navigations for each model in both trajectories (for illustration purposes):*

$$N^3(m_1^1) = \{a, b\}$$
$$N^3(m_1^2) = \{b, c\}$$
$$N^3(m_2^1) = \{a, c\}$$

$$N^3(m_2^2) = \{a, b, c\}$$
$$N^3(m_2^3) = \{b\}$$

3. *Calculate the pairwise Jaccard similarity for models in t_1 and t_2 using the formula:*

$$Jaccard(t_1^i, t_2^i) = \frac{|N^3(t_1^i) \cap N^3(t_2^i)|}{|N^3(t_1^i) \cup N^3(t_2^i)|}$$

For $i = 1$:

$$Jaccard(m_1^1, m_2^1) = \frac{|\{a, b\} \cap \{a, c\}|}{|\{a, b\} \cup \{a, c\}|} = \frac{1}{3}$$

For $i = 2$:

$$Jaccard(m_1^2, m_2^2) = \frac{|\{b, c\} \cap \{a, b, c\}|}{|\{b, c\} \cup \{a, b, c\}|} = \frac{2}{3}$$

For $i = 3$:

$$Jaccard(m_1^2, m_2^3) = \frac{|\{b, c\} \cap \{b\}|}{|\{b, c\} \cup \{b\}|} = \frac{1}{2}$$

The average pairwise Jaccard similarity for the models in t_1 and t_2 is:

$$d_{trajectory}(t_1, t_2) = \frac{\frac{1}{3} + \frac{2}{3} + \frac{1}{2}}{3} = \frac{1 + \frac{3}{2}}{3} = \frac{5}{6}$$

4. *Since T only has 2 trajectories, the average pairwise similarity is just:*

$$similarity(T) = d_{trajectory}(t_1, t_2) = \frac{5}{6}$$

Thus, the similarity of the trajectory set T is $\frac{5}{6}$.

Chapter 5

Evaluation

We conduct an experimental evaluation of the work to answer the following research questions:

RQ1 How does our monitor-based guided exploration scale compared to other DSE approaches in terms of generation time, for different initial model sizes?

RQ2 How does our monitor-based guided exploration scale compared to other DSE approaches in terms of generation time, for different numbers of generated solutions?

RQ3 How structurally diverse trajectories can be generated with our approach compared to random exploration from various sized initial models?

The Java implementation of the test environment is available on GitHub[1] under the public Eclipse license[2].

5.1 Setup

5.1.1 Compared approaches

Throughout the evaluation, we will be comparing the performance of our guided DSE (**Guide**) with DSE without objective function using random exploration (**Rand**) also used as a basis of comparison in [1]. Random exploration basically takes a pseudo-random number as the fitness value for each transformation rule and all of their object bindings, then chooses the rule and binding with the smallest value. That way, it simulates a random choice for the next generation step.

5.1.2 Case studies

Due to the absence of systematically constructed performance benchmarks for the evaluation of trajectory generation, we evaluated our approach in the context of 3 different domains (and the corresponding DSLs) that include complex structural and well-formedness constraints. The first domain also serves as the main running example of this work.

[1] https://github.com/fdominik98/refinery/tree/store-monitor-integration
[2] https://www.eclipse.org/legal/epl-2.0/

- **Traffic situation** (Traf) models the domain of a real-life traffic situation where we first formalize the abstract scenario of a dangerous take-over from the left between two cars, then based on the formalization we generate several different trajectories. Each trajectory will represent a scenario where a dangerous situation has occurred.
- **Message routing** (Rout) was taken from a previous study [33] in which they considered a system where a sender sends messages at non-deterministic time points, which then have to be transmitted to a receiver via a network of routers within a given time limit. The original study used MTGL to identify messages that were just sent, track them over time, and check if their individual deadlines were met. However, our aim is to generate various scenarios where the messages are not received on time. To achieve this, we use the negated version of their MTGL formalization to build our monitor, which guides the exploration.
- **Gesture recognition** (Rec) Originated from [16], where a human's motion is recorded with the help of tracking points on the body. While in the original article, the goal was to recognize patterns of movement formalized with CEP, here we are generating several variations of that gesture using that same formalization as our monitor.

5.1.3 Measurement environment

- Software tools: The evaluation workflow is integrated into the Refinery Tool [39] alongside the solution itself, as the measurement leverages many of its functionalities like neighbourhood calculation and visualization. The workflow is written in Java programming language and run in JRE 19.0.2. For the development, I used Intellij IDEA 2023.1.2.
- Hardware setup:
  - *Model:* HP Omen 15
  - *Processor:* Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz 2.60 GHz
  - *Installed RAM:* 16.0 GB
  - *Operating system:* Windows10 Pro
  - *System type:* 64-bit operating system, x64-based processor
- Methodology: Prior to each evaluation sequence, we conducted 10 additional cycles and disregarded the outcomes in order to warm up the Java Virtual Machine. After the warm-up phase, we carried out 30 evaluation cycles for each configuration and calculated the median of the results. Before each run, the garbage collector was called explicitly.

5.2 RQ1: Scalability for different model sizes

Safety critical and autonomous systems can have extremely large state space, for which, generating test scenarios is a hard task. If a model consists of a large number of nodes and relationships, we assume that random search-based techniques will have a hard time finding dangerous scenarios and consume a significant amount of resources. On the other hand, we assume that our approach will handle models with larger state space more efficiently, as it is designed to find the subset of trajectories of our interests, prior to those not matching our temporal behaviour patterns.

Setup For case studies Traf and Rout, we prepared different initial model configurations with increasing model size and complexity. We denote the different sizes of models in the context of each case study as Small for the smallest and least complex model in the domain, Medium implying a medium-sized model, and Large for the model with the highest number of nodes and complexity.

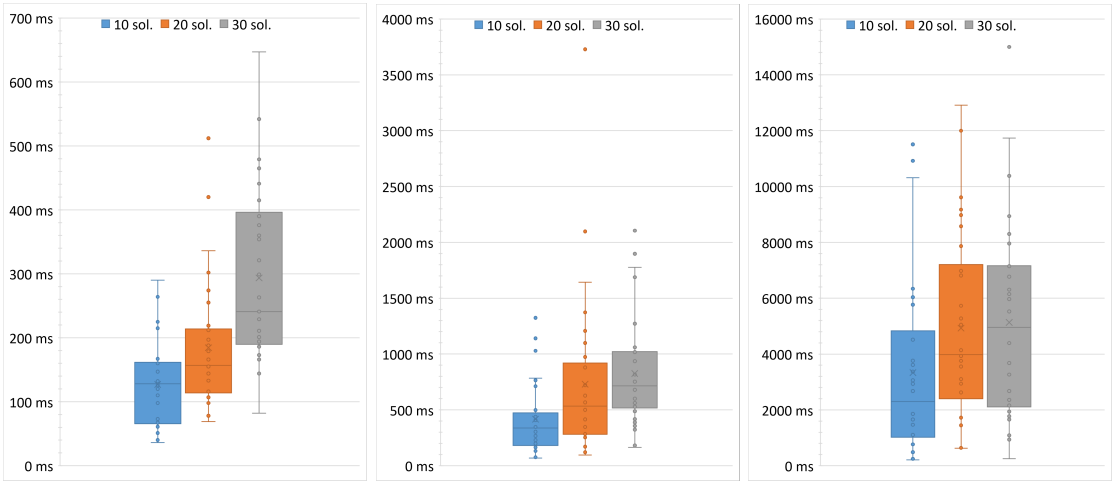
- In the Traf domain, we wired together the lanelets like in Example 3 and placed cars on them that are moving between neighbouring lanelets. We scale the initial model by adding cars and extending the lanelet grid, thus the number of possible trajectories is exponentially increased:

Initial model	Nodes	Relationships	Description
Small	22	82	10 left lanelets, 10 right lanelets, 2 cars. Representing a 10-long road with 2 lanes.
Medium	83	323	2x20 left lanelets, 2x20 right lanelets, 3 cars. Representing a 20-long road with 4 lanes.
Large	185	725	3x30 left lanelets, 3x30 right lanelets, 5 cars. Representing a 30-long road with 6 lanes.

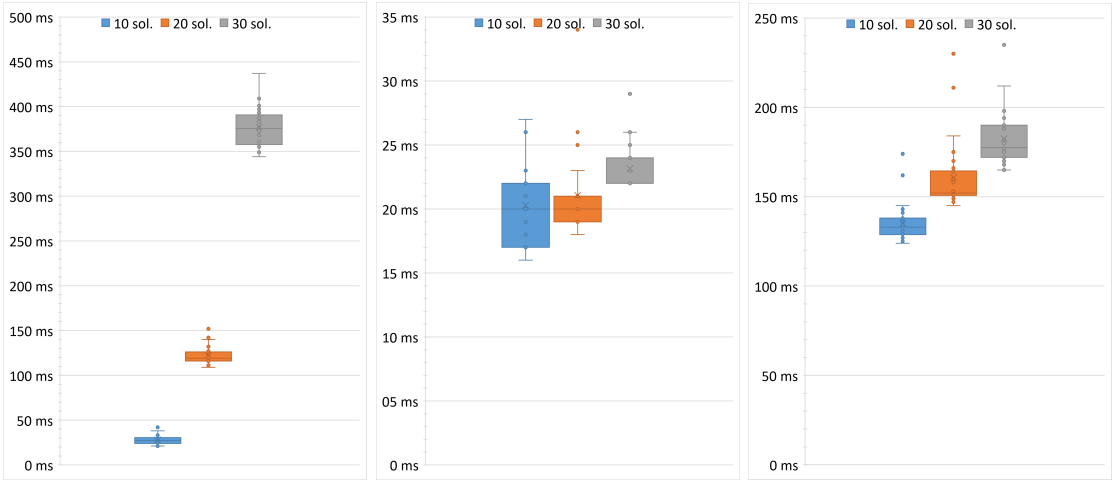
- In the Rout domain, a router network is modeled, where the sender takes a message and passes it on to the next hop. Here, we can leverage the timed automata extension introduced in Section 4.2.5, by defining a time limit for every message that has been sent. If the limit is too high, it can be impossible to find a strategy where all messages reach timeout. If the time limit is too low, it is too easy to make the messages reach timeout. We scale the initial model by adding more messages and routers. We lengthen and widen the possible paths a message can reach its destination through, the receiver node. This also comes with increasing the time limit:

Initial model	Nodes	Relationships	Description
Small	10	7	3 messages, 5 routers, 1 sender, 1 receiver. We use a time limit of 6 time units for each message.
Medium	17	15	6 messages, 10 routers, 1 sender, 1 receiver. We use a time limit of 16 time units for each message.
Large	26	24	9 messages, 15 routers, 1 sender, 1 receiver. We use a time limit of 24 time units for each message.

Result In Figure 5.1 and Figure 5.2 we can see execution time measurements (in milliseconds) for Auto and Rout. Each diagram shows three measurements, as we generated 10 (**blue**), 20 (**orange**), and 30 (**gray**) solutions. Each column of the figures identifies a model size (Small, Medium, Large) and each row a strategy, (Rand, Guide). This way we can compare Rand and Guide for each model size, and we can see how well they scale.



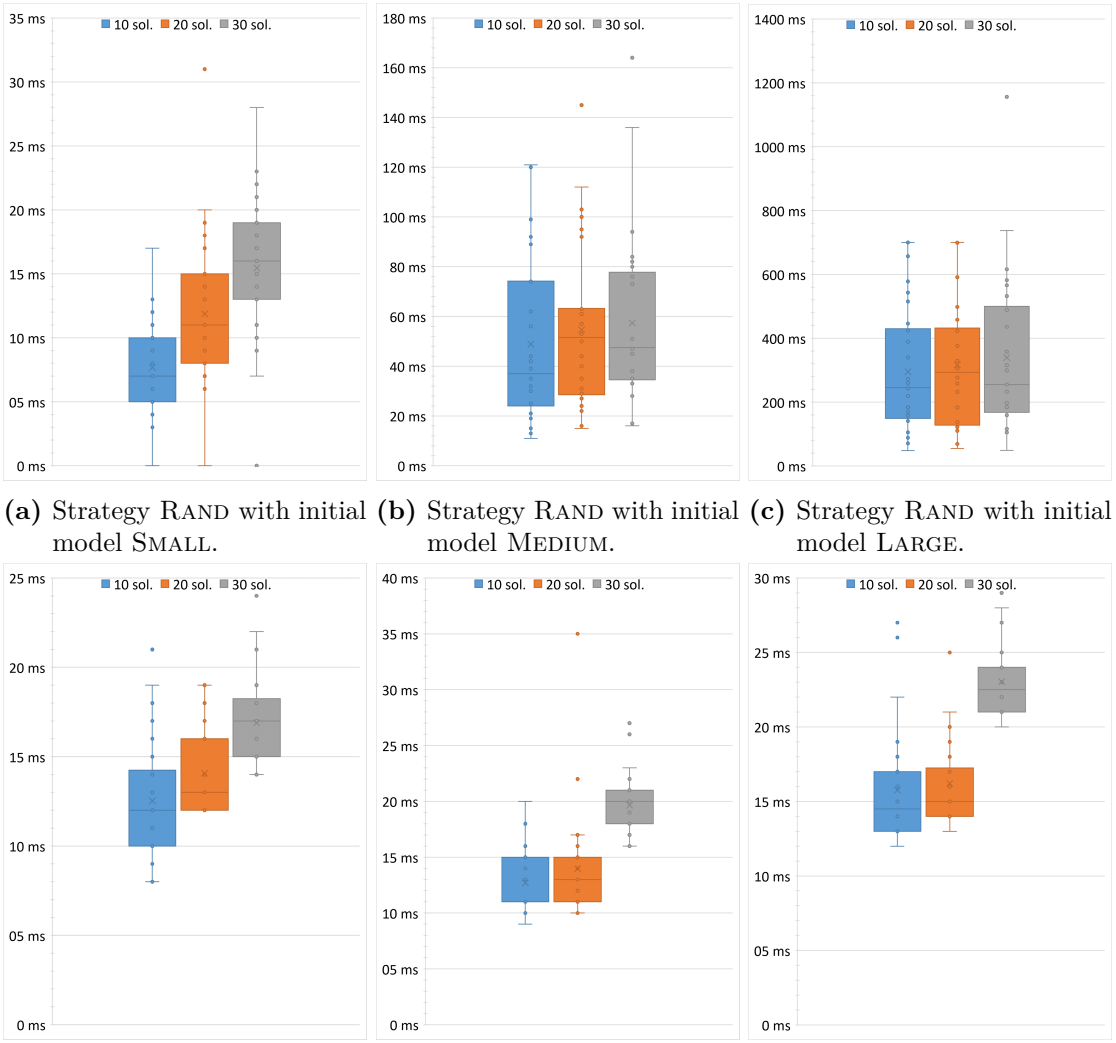
(a) Strategy RAND with initial model SMALL. **(b)** Strategy RAND with initial model MEDIUM. **(c)** Strategy RAND with initial model LARGE.



(d) Strategy GUIDE with initial model SMALL. **(e)** Strategy GUIDE with initial model MEDIUM. **(f)** Strategy GUIDE with initial model LARGE.

Figure 5.1: Execution times of the explorations in the TRAF domain.

As we compare Figure 5.1a and Figure 5.1d, we can see very little difference, GUIDE is generating trajectories slightly faster. In the next column, however (Figures 5.1b and 5.1e) the RAND strategy converges to 1000 ms while the GUIDE strategy drops to 25 ms. The tendency is no different in the case of the LARGE model (Figures 5.1c and 5.1f). We can observe a huge 6x rise in execution time with RAND, while GUIDE kept the previous order of magnitude.



(a) Strategy RAND with initial model SMALL. **(b)** Strategy RAND with initial model MEDIUM. **(c)** Strategy RAND with initial model LARGE.

(d) Strategy GUIDE with initial model SMALL. **(e)** Strategy GUIDE with initial model MEDIUM. **(f)** Strategy GUIDE with initial model LARGE.

Figure 5.2: Execution times of the explorations in the ROUT domain.

We experience the same tendency with the ROUT measurements. Following Figures 5.2d to 5.2f, the execution time stays between 10 and 30 milliseconds for all three models. However, if we look at the RAND Figures 5.2a to 5.2c it emerges from 10 to 400 milliseconds model by model.

Discussion *RA1: We empirically found proof that our GUIDE approach is not sensitive to model size and complexity increases at scales where the RAND approach visibly finds difficulties.*

5.3 RQ2: Scalability for amount of solutions

When testing safety-critical systems, engineers would like to generate as many meaningful test scenarios as possible to cover the behaviour of the system more expansively. Generating trajectories becomes harder and more expensive as we increase the size of the result set.

In the following, we would like to see how well our approach scales when systematically increasing the number of solutions to generate.

Setup In the REC domain, the model structure is more static than the last examples. We have 8 fixed nodes representing the human body, that have an abstract position property $y \in \mathbb{N} \cup [2, 10]$. They are also connected with fixed relationships:

Nodes	Relationships	Description
8	7	body, head, 2 hands, 2 shoulders, 2 elbows, and relations joining them.

Result In Figure 5.3 and Figure 5.4 we can see execution time measurements (in milliseconds) for REC. Each diagram shows six measurements, as we generated 20 (**blue**), 30 (**orange**), 40 (**gray**), 50 (**yellow**), 60 (**dark blue**), 70 (**green**) solutions. We observe RAND in Figure 5.3 and GUIDE) in Figure 5.4 to compare the execution time in each case.

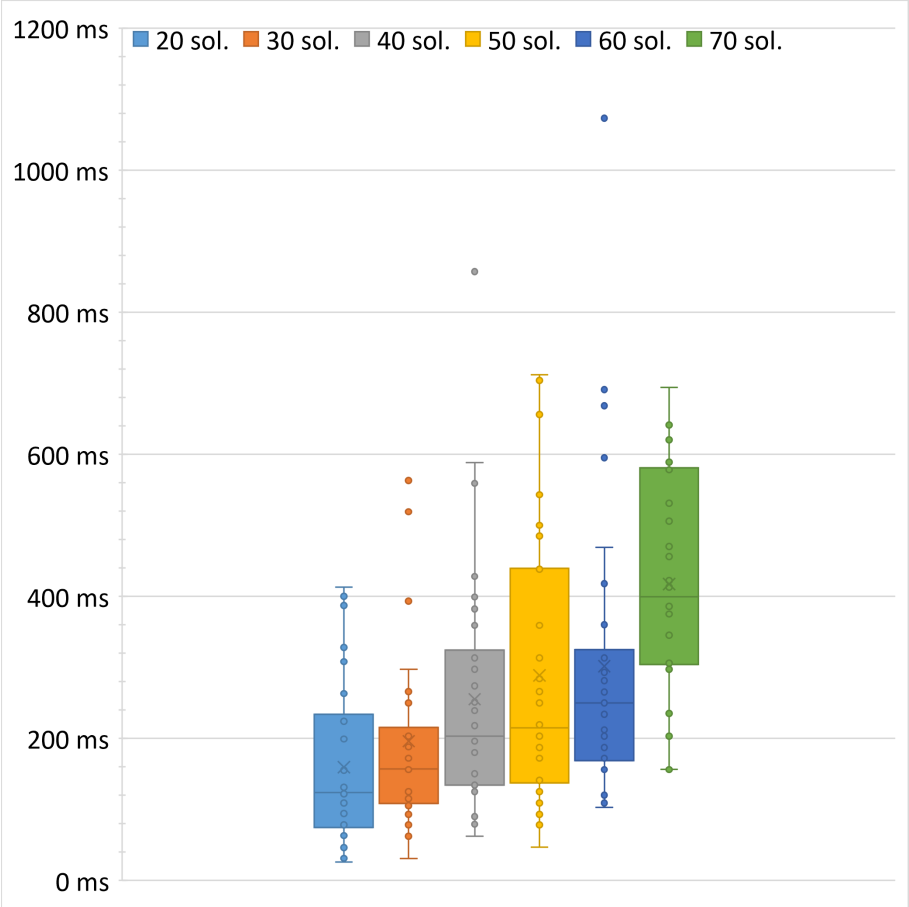


Figure 5.3: Runtime execution scalability evaluation with strategy RAND and initial model SMALL.

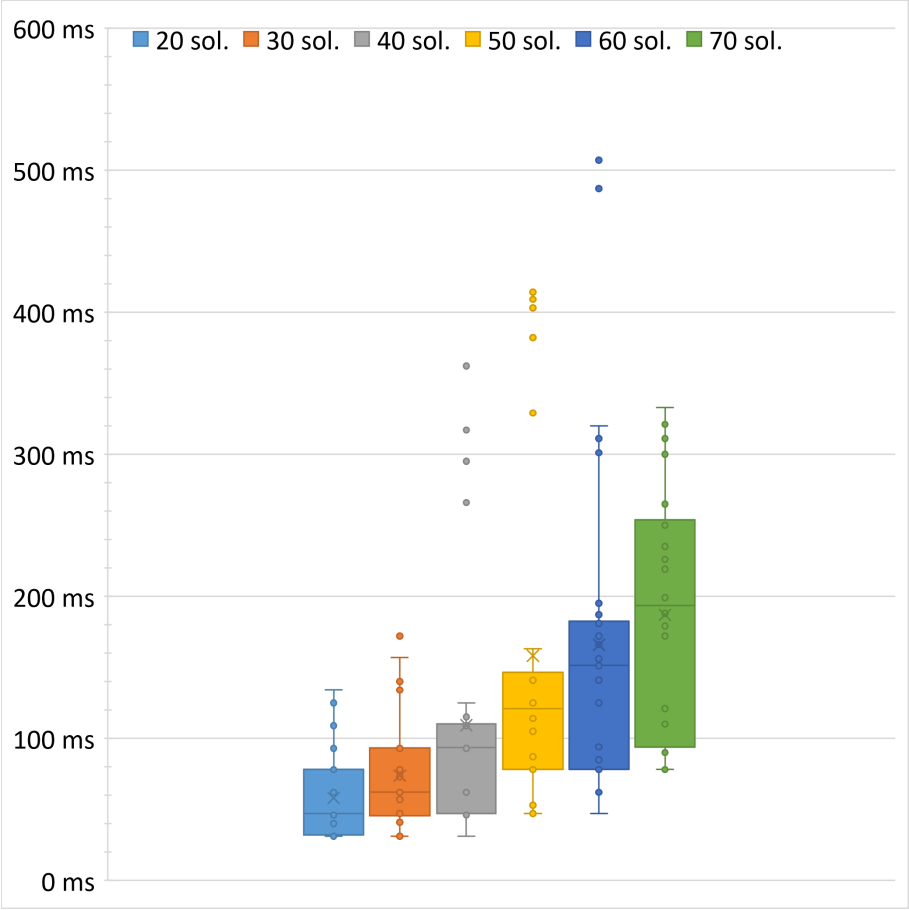


Figure 5.4: Runtime execution scalability evaluation with strategy GUIDE and initial model SMALL.

Discussion *RA2: Based on the two diagrams, we can state that the GUIDE approach is generally faster than the RAND approach, both for smaller and larger solution sets. Also, the execution time increases more consistently in the case of the GUIDE approach, while in the case of the RAND approach, the execution times may vary more due to its random behaviour.*

5.4 RQ3: Diversity

Test scenario diversity is crucial in the field of model-driven testing because it ensures that the testing process exhaustively explores the vast space of possible behaviours in a software system. By incorporating a wide range of test scenarios, testers can uncover edge cases and non-obvious bugs that might be missed with a more homogenous test suite.

Setup To evaluate the structural similarity of test sets generated by our approach and the random exploration approach, we use our trajectory diversity metric introduced in Section 4.3.3. We take the generated trajectory sets from the previous examples and evaluate how similar the trajectories are in one set. We do this for every domain, comparing both strategies, for different sizes of solution sets.

Result Tables 5.1 to 5.3 contain the median of the measured average trajectory similarities for each configuration. Large similarity values imply more similar trajectories, while small values imply a more diverse trajectory set.

		Generated trajectories		
Initial model	Strategy	10	20	30
Small				
	Rand	0.44830	0.51262	0.45707
	Guide	0.67151	0.64889	0.65409
Medium				
	Rand	0.49646	0.48315	0.5083
	Guide	0.4796	0.47037	0.51846
Large				
	Rand	0.40588	0.41359	0.40287
	Guide	0.59314	0.67694	0.65425

Table 5.1: Similarity evaluation on the Traf domain

		Generated trajectories		
Initial model	Strategy	10	20	30
Small				
	Rand	0.59853	0.60791	0.59856
	Guide	0.70748	0.62498	0.64225
Medium				
	Rand	0.46880	0.58701	0.59315
	Guide	0.60496	0.71618	0.65707
Large				
	Rand	0.31644	0.49058	0.52215
	Guide	0.56295	0.59015	0.51735

Table 5.2: Similarity evaluation on the Rout domain

	Generated trajectories		
Strategy	20	30	40
Rand	0.31078	0.33508	0.32586
Guide	0.64381	0.61574	0.69366
	50	60	70
Rand	0.33162	0.30526	0.30417
Guide	0.55740	0.58950	0.58205

Table 5.3: Similarity evaluation on the Rec domain

Discussion *RA3: Random heuristic seems to be generating more diverse scenarios in general for each configuration. This is expected, as our approach is designed to explore a more specific system behaviour than random. we conclude that really strong correlations between the solution size, the model size, and the diversity cannot be discovered according to the measurements.*

5.5 Threats to validity

Internal validity To strengthen internal validity, we use warm-up measurements before each trajectory set generation, moreover, we conduct repeated measurements and analyze the distribution of the results when considering execution time (**RQ1** and **RQ2**). For deriving diversity (**RQ3**), we take the median of the similarity values calculated after each measurement. Additionally, we explicitly call the garbage collector between executions.

External validity To mitigate threats to external validity, we take three different case studies, used in various sources in the literature. Each case study has a different model structure and behaviour. Apart from that, we use three different model sizes for Traf and Rout to gain a more thorough understanding of their scalability.

Chapter 6

Related work

Monitoring approaches In [33, 21] authors enabled a satisfaction check for MTGL conditions by finite typed graph sequences. They introduced a mapping of a finite typed graph sequence into a single graph, with history representing the changes over the graphs of the sequence. With this reduction, they can match equivalent queries on the graph with history instead of the sequence.

In [16] the authors use Parametric Timeout Automata to monitor complex event patterns. We use a similar approach to monitor model changes, but instead of events, we apply graph queries.

DSE approaches Model-driven guided design space exploration implemented over graph transformations is widely used in the field.

Rule-based design space exploration Model-checking approaches to analyze graph transformation systems are similar to our approach, as they also perform state space exploration. One can categorize them as compiled approaches such as [8] [7], which translate graphs and graph transformation rules into off-the-shelf model checkers to carry out verification, and interpreted approaches like [6], which store system states as graphs and directly apply transformation rules.

Logic Solver Approaches There are approaches that map a model generation problem into a logic problem, which is solved by underlying SAT/SMT-solvers. Complete frameworks with standalone specification languages like [24] (using the Z3 SMT-solver [17]), and Alloy [23] (using SAT-solvers like Sat4j [25]).

In [29] the authors focus on automated model generators, which represent tests in the form of graph models. The work uses multiplicity reasoning to configure graph generators by numeric constraints to focus model generation on the relevant fragment of models and filter out unrealistic models.

In [3] the authors can automatically derive consistent graph models that satisfy both structural and attribute constraints. For that purpose, the structural constraints are satisfied along partial model refinement, while attribute constraints are satisfied by repeatedly calling the Z3 SMT-solver [17] or other solvers.

In [18] the authors explore the inclusion of search directly in model transformations, without the need for an intermediate representation.

Chapter 7

Conclusion and future work

In this work, we proposed a novel approach for engineers to automatically generate abstract, model-based test scenarios for complex context-dependent systems, using design-time monitoring and temporal logic specifications. To achieve this, we extended the Parametric Timed Automata formalism with graph pattern matching and provided a formalism to execute the automata by applying graph transformation rules. We introduced a specialized fitness function for guiding DSE based on the monitoring automata, ensuring the derivation of relevant test sequences with sufficient time cost. To measure the diversity of the generated test sequences, we introduced a technique to derive a single metric on the similarity of the generated result set. To verify the approach, we integrated the proposed monitors into the open-source Refinery graph processing framework and conducted experimental evaluation of the applicability and scalability of the proposed approach, both in terms of test sequence diversity and generation time. We did this on multiple case studies, including scenario generation for traffic situations, transmit-receiver networks, and gesture recognition.

As a future work, we aim to integrate our solution with a concretization framework to derive expansive concrete scenarios that can be executed in a simulation environment. In this new framework, we intend to reuse the same monitor to evaluate the system under test.

On the implementation side, the generation can be further optimized by priorly detecting monitor states, where the satisfaction of a defined behaviour is already known to be infeasible, for example in the case of timed behaviors.

Additional comparisons can be made with other DSE techniques like Alloy [23], by implementing the model generation problems for our case studies with graph constraints, for example in the form of a graph with history [33, 21].

Bibliography

[1] Hani Abdeen, Dániel Varró, Houari Sahraoui, András Szabolcs Nagy, Csaba Debreceni, Ábel Hegedüs, and Ákos Horváth. Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 289–300, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. DOI: 10.1145/2642937.2643005. URL https://doi.org/10.1145/2642937.2643005.

[2] Étienne André, Didier Lime, and Mathias Ramparison. Parametric updates in parametric timed automata. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 39–56, Cham, 2019. Springer International Publishing. ISBN 978-3-030-21759-4.

[3] Aren Babikian, Oszkar Semerath, and Daniel Varro. *Automated Generation of Consistent Graph Models with First-Order Logic Theorem Provers*, pages 441–461. 04 2020. ISBN 978-3-030-45233-9. DOI: 10.1007/978-3-030-45234-6_22.

[4] Aren A. Babikian, Oszkár Semeráth, Anqi Li, Kristóf Marussy, and Dániel Varró. Automated generation of consistent models using qualitative abstractions and exploration strategies. *Software and Systems Modeling*, 21(5):1763–1787, Oct 2022. ISSN 1619-1374. DOI: 10.1007/s10270-021-00918-6. URL https://doi.org/10.1007/s10270-021-00918-6.

[5] Aren A. Babikian, Oszkár Semeráth, and Dániel Varró. Concretization of abstract traffic scene specifications using metaheuristic search, 2023.

[6] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, pages 14–29, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45832-6.

[7] Luciano Baresi and Paola Spoletini. On the use of alloy to analyze graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations*, pages 306–320, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38872-2.

[8] Luciano Baresi, Vahid Rafe, Adel T. Rahmani, and Paola Spoletini. An efficient solution for model checking graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 213(1):3–21, 2008. ISSN 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.04.071. URL https://www.sciencedirect.com/science/article/pii/S1571066108002867. Proceedings of the Third Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2007).

[9] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32759-9.

[10] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018. ISBN 978-3-319-75632-5. DOI: 10.1007/978-3-319-75632-5_1. URL https://doi.org/10.1007/978-3-319-75632-5_1.

[11] David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 1–18, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14295-6.

[12] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2), may 2015. ISSN 0004-5411. DOI: 10.1145/2699444. URL https://doi.org/10.1145/2699444.

[13] Nelly Bencomo, Sebastian Götz, and Hui Song. Models@run.time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling*, 18(5): 3049–3082, Oct 2019. ISSN 1619-1374. DOI: 10.1007/s10270-018-00712-x. URL https://doi.org/10.1007/s10270-018-00712-x.

[14] Philipp Bender, Julius Ziegler, and Christoph Stiller. Lanelets: Efficient map representation for autonomous driving. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pages 420–425, June 2014. DOI: 10.1109/IVS.2014.6856487.

[15] Boqi Chen, Kristóf Marussy, Sebastian Pilarski, Oszkár Semeráth, and Daniel Varro. Consistent scene graph generation by constraint optimization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394758. DOI: 10.1145/3551349.3560433. URL https://doi.org/10.1145/3551349.3560433.

[16] István Dávid, István Ráth, and Dániel Varró. Foundations for streaming model transformations by complex event processing. *Software & Systems Modeling*, 17(1): 135–162, Feb 2018. ISSN 1619-1374. DOI: 10.1007/s10270-016-0533-1. URL https://doi.org/10.1007/s10270-016-0533-1.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[18] Joachim Denil, Maris Jukss, Clark Verbrugge, and Hans Vangheluwe. Search-based model optimization using model transformations. In Daniel Amyot, Pau Fonseca i Casas, and Gunter Mussbacher, editors, *System Analysis and Modeling: Models and Reusability*, pages 80–95, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11743-0.

[19] I. Nyoman Pande Wahyu Dharmawan and Riyanarto Sarno. Book recommendation using neo4j graph database in bibtex book metadata. In *2017 3rd International*

Conference on Science in Information Technology (ICSITech), pages 47–52, 2017. DOI: 10.1109/ICSITech.2017.8257084.

[20] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 8–29, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67531-2.

[21] Holger Giese, Maria Maximova, Lucas Sakizloglou, and Sven Schneider. Metric temporal graph logic over typed attributed graphs. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to Software Engineering*, pages 282–298, Cham, 2019. Springer International Publishing. ISBN 978-3-030-16722-6.

[22] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3):399–436, Sep 2015. ISSN 1573-7535. DOI: 10.1007/s10515-014-0163-1. URL https://doi.org/10.1007/s10515-014-0163-1.

[23] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, apr 2002. ISSN 1049-331X. DOI: 10.1145/505145.505149. URL https://doi.org/10.1145/505145.505149.

[24] Ethan K. Jackson, Tihamér Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 653–667, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24485-8.

[25] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. ISSN 1574-0617. DOI: 10.3233/SAT190075. URL https://doi.org/10.3233/SAT190075. 2-3.

[26] István Majzik, Oszkár Semeráth, Csaba Hajdu, Kristóf Marussy, Zoltán Szatmári, Zoltán Micskei, András Vörös, Aren A. Babikian, and Dániel Varró. Towards system-level testing with coverage guarantees for autonomous vehicles. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 89–94, Sep. 2019. DOI: 10.1109/MODELS.2019.00-12.

[27] Florence Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, volume 3. Citeseer, 1991.

[28] Kristóf Marussy, Oszkár Semeráth, Aren A. Babikian, and Dániel Varró. A specification language for consistent model generation based on partial models. *J. Object Technol.*, 19:3:1–22, 2020. URL https://api.semanticscholar.org/CorpusID:226239687.

[29] Kristóf Marussy, Oszkár Semeráth, and Dániel Varró. Automated generation of consistent graph models with multiplicity reasoning. *IEEE Transactions on Software Engineering*, 48(5):1610–1629, 2022. DOI: 10.1109/TSE.2020.3025732.

[30] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

[31] Arend Rensink and Dino Distefano. Abstract graph transformation. *Electron. Notes Theor. Comput. Sci.*, 157(1):39–59, may 2006. ISSN 1571-0661. DOI: 10.1016/j.entcs.2006.01.022. URL https://doi.org/10.1016/j.entcs.2006.01.022.

[32] Jesús Rosa Bilbao, Juan Boubeta Puig, et al. Mode driven engineering for complex event processing: A survey. *Journal of Object Technology*, 2022.

[33] Sven Schneider, Maria Maximova, and Holger Giese. Probabilistic metric temporal graph logic. *CoRR*, abs/2106.08418, 2021. URL https://arxiv.org/abs/2106.08418.

[34] Fabian Schuldt, Simon Ulbrich, Till Menzel, Andreas Reschka, and Markus Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, 09 2015. DOI: 10.1109/ITSC.2015.164.

[35] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 969–980, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. DOI: 10.1145/3180155.3180186. URL https://doi.org/10.1145/3180155.3180186.

[36] Oszkár Semeráth, Aren A. Babikian, Anqi Li, Kristóf Marussy, and Daniel Varró. Automated generation of consistent models with structural and attribute constraints. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 187–199, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370196. DOI: 10.1145/3365438.3410962. URL https://doi.org/10.1145/3365438.3410962.

[37] Oszkár Semeráth, Rebeka Farkas, Gábor Bergmann, and Dániel Varró. Diversity of graph models and graph generators in mutation testing. *International Journal on Software Tools for Technology Transfer*, 22(1):57–78, Feb 2020. ISSN 1433-2787. DOI: 10.1007/s10009-019-00530-6. URL https://doi.org/10.1007/s10009-019-00530-6.

[38] Peter Struss. Model-based and qualitative reasoning: An introduction. *Ann. Math. Artif. Intell.*, 19:355–381, 04 1997. DOI: 10.1023/A:1018916007995.

[39] Prof. Dániel Varró, Ficsor Attila, Garami Bence, Golej Márton Marcell, Marussy Kristóf, and Semeráth Oszkár. Refinery, 2023. URL https://github.com/graphs4value/refinery. Accessed: 2023-10-17.