Department of Electrical and Computer Engineering
McGill University, Montreal

Query-Based Runtime Monitoring in Real-Time and Distributed Systems

Ph.D. Thesis

Márton Búr

July 2021

A thesis submitted to McGill University in partial fulfillment
of the requirements of the degree of Doctor of Philosophy

Abstract

Modern smart and safe cyber-physical systems (CPSs) have complex interactions with their uncertain environment that is rarely known in advance, while operating in a trustworthy way. They heavily depend on intelligent data processing carried out over a heterogeneous and distributed computing platform with resource-constrained devices to monitor and control autonomous behavior. Due to these characteristics, design time verification and testing used in traditional safety-critical systems often become infeasible in practice. As such, runtime verification approaches are used to ensure correct operation.

This thesis addresses the challenges of runtime monitoring in smart and safe CPSs by adapting graph-like representations mainly used at design time to a runtime setting to provide an extra layer of safety to data-intensive critical systems.

First, we adapt runtime models for resource-constrained real-time environments to capture the system state in an dynamic knowledge graph that incorporates domain concepts. We propose to capture safety rules of runtime monitors as graph queries, which are evaluated over snapshots of the underlying runtime model of the system. Furthermore, we show how to derive deployable monitoring programs from high-level query specifications automatically.

Then, worst-case execution time (WCET) analysis is presented for these auto-generated query-based runtime monitoring programs to enable their use in hard real-time settings. To achieve this, we provide two complementary approaches. One approach uses a combination of traditional static analysis-based WCET computation and a state-of-the-art graph generation technique to synthesize so-called witness models up to a given model size where the query program is estimated to have the highest (i.e., worst-case) run time. The other approach provides on-line (i.e., runtime) estimates of execution time for a query program over a specific model by using a symbolic formula which relies on condensed graph model metrics.

Finally, we extend runtime models and queries to a distributed and resource-constrained setting. The runtime model is partitioned among the participating nodes in the platform, and it is consistently kept up-to-date in a continuously evolving environment by a time-triggered model management protocol. We provide a semantic treatment of distributed graph queries using 3-valued logic to incorporate uncertainties and delays in a semantically consistent way. Furthermore, our runtime models offer a (domain-specific) model query and manipulation interface over the reliable communication middleware of the Data Distribution Service (DDS) standard widely used in the CPS domain.

For each contribution, we evaluate the feasibility and scalability of our approaches using prototype implementations in the context of the MoDeS3 educational CPS platform.

Abrégé

Les systèmes cyber-physiques (CPS) modernes intelligents et sûrs ont une interaction complexe avec leur environnement incertain et rarement connu à l'avance tout en fonctionnant de manière fiable. Ils dépendent fortement d'un traitement intelligent des données effectué sur une plate-forme informatique hétérogène et distribuée avec des dispositifs de calculs aux ressources informatiques limitées pour surveiller et contrôler le comportement autonome. En raison de ces caractéristiques, la vérification et les essais effectués lors de la conception traditionnelle des systèmes sûrs et critiques sont la plupart du temps irréalisables en pratique. Étant donné ces contraintes, des approches de vérification d'exécution sont utilisées pour garantir un fonctionnement correct.

Cette thèse aborde les défis de la surveillance d'exécution dans les CPS intelligents et sûrs en adaptant des représentations de graphes principalement utilisées au moment de la conception à un modèle en temps réel de sorte à offrir une couche de sécurité supplémentaire aux systèmes critiques ayant un volume important de données.

Tout d'abord, nous adaptons les modèles d'exécution pour les environnements en temps réel à ressources informatiques limitées afin de capturer l'état du système dans un graphe de connaissances dynamique qui incorpore des concepts du domaine en question. Nous proposons de capturer les règles de sécurité des moniteurs d'exécution sous forme de requêtes graphiques qui sont évaluées à partir d'instantanés du modèle d'exécution sous-jacent du système. De plus, nous démontrons comment dériver automatiquement des programmes déployables de surveillance à partir de spécifications de requêtes de haut niveau.

Ensuite, l'analyse du temps d'exécution dans le pire des cas (WCET) est présentée pour ces programmes de surveillance d'exécution basés sur des requêtes générées automatiquement afin de permettre leur utilisation dans des paramètres en temps réel dur. Pour y parvenir, nous proposons deux approches complémentaires. Une approche utilise une combinaison du calcul WCET traditionnel basé sur l'analyse statique et une technique de génération de graphes de pointe pour synthétiser des modèles dits témoins jusqu'à une taille de modèle donnée où le programme de requête est estimé d'avoir le temps d'exécution le plus élevé (c.-à-d. , dans le pire des cas). L'autre approche fournit des estimations lors de l'exécution d'un programme de requête sur un modèle spécifique en utilisant une formule symbolique qui repose sur des métriques d'un modèle de graphe condensé.

Enfin, nous étendons les modèles d'exécution et les requêtes dans un environnement distribué et limité par les ressources informatiques. Le modèle d'exécution est partitionné entre les nœuds participants de la plate-forme et il est constamment mis à jour dans un environnement en constante évolution par un protocole de gestion de modèle déclenché par le temps. Nous fournissons un traitement sémantique des requêtes de graphes distribués en utilisant la logique ternaire pour incorporer les incertitudes et les retards d'une manière sémantiquement cohérente. En outre, nos modèles d'exécution offrent une interface de requête et de manipulation de modèle (spécifique au domaine) sur l'intergiciel de communication fiable du standard DDS (Data Distribution Service) largement utilisé dans le domaine CPS.

Pour chaque contribution, nous évaluons la faisabilité et l'évolutivité de nos approches en utilisant des prototypes d'implémentations dans le contexte de la plate-forme CPS éducative MoDeS3.

Acknowledgements

This thesis marks the end of my journey as a Ph.D. student at McGill University. Along the way, I have had the privilege to meet and work with excellent scientists and engineers from different parts of the world and solve challenging problems. Besides professional advancement, I am grateful for the lots of new friendships and the invaluable life experiences I gained during these unforgettable years in the wonderful city of Montreal. I would like to thank everybody who has supported me in any way throughout my studies. Additionally, I would like to give special thanks to the people who made it possible to achieve my goal.

Ph.D. Advisor. First and foremost, I would like to express my heartfelt gratitude towards my advisor Professor Dániel Varró for his excellent guidance and continuous support.

Ph.D. Committee Members. I extend my sincere thanks to Professor Bettina Kemme and Professor Gunter Mussbacher for the helpful feedback and insights they have provided me during my studies. I gratefully acknowledge the efforts of Professor Abdelwahab Hamou-Lhadj and Professor Gunter Mussbacher for reviewing and providing feedback on the thesis.

Colleagues. I also had great pleasure of working with the excellent colleagues at McGill University's Electrical and Computer Engineering Department, at the Critical Systems Research Group (FTSRG) in Budapest, and at IncQuery Labs Ltd. I am especially grateful for Kristóf Marussy, Gábor Szilágyi, and András Vörös for the countless hours of joint work. I thank the MoDeS3 developer team, especially Zsolt Mázló and Bálint Hegyi the hard work and technical support, and Mathieu Boucher for helping with the translation of the abstract to French. Last but not least, I am deeply indebted to Gábor Szárnyas for his assistance.

Financial support. This work would not have been possible without the generous financial support provided by the McGill Engineering Doctoral Award (MEDA) program maintained by the Faculty of Engineering and the Graduate Research Enhancement and Travel Awards (GREAT Awards). I also received financial support from the NSERC RGPIN-04573-16 project.

Friends and family. Finally, I would like to extend my deepest gratitude towards my close friends and family, who have continuously provided me encouragement, support, and love.

Preface

Each of my contributions presented henceforth is a result of the research conducted at the Department of Electrical and Computer Engineering at McGill University under the PhD supervision of Professor Dániel Varró. Part of the results in this thesis are included in the publications listed below. For each work, I provide a short summary about the individual contributions of authors, and include which chapters of the thesis are linked to the paper. A detailed description of the contributions is presented in Section 1.4.

Journal Papers

[*j*1] Márton Búr, Kirstóf Marussy, Brett Meyer, and Dániel Varró. Worst-case execution time calculation for query-based monitors by witness generation. *ACM Transactions on Embedded Computing Systems*, 2020. Accepted.

▷ *The concept of witness models used for worst-case execution time (WCET) estimation of real-time graph query programs is my contribution. Graph generation technique for static WCET estimation is the contribution of Kristóf Marussy. The evaluation of the proposed approach is joint work with Kristóf Marussy. Dániel Varró and Brett Meyer were helping the work as advisors by providing initial research ideas and continuous feedback. Results regarding WCET analysis are included in Chapter 7, while real-time graph data structures are added to Chapter 5.*

[*j*2] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems. *International Journal on Software Tools for Technology Transfer* 22(1), Sept. 2019, pp. 79–102. DOI: `10.1007/s10009-019-00531-5`.

▷ *The foundations of distributed query evaluation is my contribution. The distributed runtime model update protocol is designed by Dániel Varró and myself. Prototype implementation and evaluation is done by Gábor Szilágyi and myself. Dániel Varró and András Vörös were helping in the formal mathematical treatment and continuously provided advice and feedback. The distributed model update protocol is presented in Chapter 8, while the distributed query evaluation is presented in Chapter 9.*

Peer-Reviewed International Conference Papers

[*c*3] Márton Búr and Dániel Varró. Towards WCET estimation of graph queries@run.time. In: *IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, IEEE, Sept. 2019. DOI: `10.1109/MODELS.2019.00007`.

▷ *The on-line WCET estimation technique for runtime query-based monitoring programs is my contribution. The research roadmap for estimating WCET of query-based programs is a joint contribution with Dániel Varró. Part of the results are presented in Chapter 9.*

[*c*4] Márton Búr and Dániel Varró. Evaluation of distributed query-based monitoring over data distribution service. In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, IEEE, Apr. 2019. DOI: `10.1109/wf-iot.2019.8767281`.

▷ *The two different distributed query evaluation strategies are my contributions. Dániel Varró was helping the work as advisor by providing initial research ideas and continuous feedback. Part of the results are presented in Chapter 9.*

[*c*5] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries for runtime monitoring of cyber-physical systems. In: *International Conference on Fundamental Approaches to Software Engineering*, pp. 111–128. Springer International Publishing, 2018. DOI: `10.1007/978-3-319-89363-1_7`.

▷ *The foundations of distributed query-based monitors is my contribution. Prototype implementation and evaluation is a joint effort of Gábor Szilágyi and myself. Dániel Varró and András Vörös were helping in the formal mathematical treatment and continuously provided advice and feedback. Results are presented in Chapter 9.*

[*c*6] András Vörös, Márton Búr, István Ráth, Ákos Horváth, Zoltán Micskei, László Balogh, Benedek Horváth, Zsolt Mázló, and Dániel Varró. MoDeS3: model-based demonstrator for smart and safe cyber-physical systems. In: *NASA Formal Methods Symposium*, pp. 460–467. Springer International Publishing, 2018. DOI: `10.1007/978-3-319-77935-5_31`.

▷ *I have made contributions to the hierarchical query-based runtime monitors running in the MoDeS3 system, and have served as the lead architect of the project for over a year. The MoDeS3 project is a joint effort of many participants. András Vörös was leading the project. István Ráth, Ákos Horváth, Zoltán Micskei, and Dániel Varró helped the work as advisors and provided ideas about what the demonstrator should feature. László Balogh, Benedek Horváth, and Zsolt Mázló worked on the implementation. Despite my contribu-*

tions, due to the highly collaborative nature of the work, I do not claim novel scientific contributions related to the core MoDeS3 demonstrator platform.

[c7] Márton Búr, Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Local search-based pattern matching features in EMF-IncQuery. In: *8th International Conference on Graph Transformation*, pp. 275–282. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-21145-9_18.

▷ *The implementation of the local search-based query debugger and its integration to the Eclipse IDE is my contribution. Zoltán Ujhelyi coordinated the development and provided technical support during development, while Ákos Horváth and Dániel Varró were helping the work as advisors by providing initial research ideas and continuous feedback.*

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

Abbreviation	Description
BB	basic program block
BBB	Beage Bone Black
CC	cyclomatic complexity
CFG	control-flow graph
CPS	cyber-physical system
DDS	Data Distribution Service
DS	domain-specific
EMF	Eclipse Modeling Framework
ES	embedded system
FOL	first-order logic
GDS	global data space
HW	hardware
ILP	integer linear programming
IPET	implicit path enumeration technique
ISA	instruction set architecture
MBSE	model-based systems engineering
MCU	microcontroller unit
MoDeS3	Model-Based Demonstrator of Smart and Safe Systems
QoS	quality of service
RQ	research question
RV	runtime verification
SBC	single-board computer
SC	safety-critical
sCPS	smart cyber-physical system
VG	VIATRA Generator
VQL	VIATRA Query Language
WCET	worst-case execution time
WF	well-formedness

Part I

Preliminaries

Chapter 1
Introduction

1.1 Smart and Safe Systems

The Internet of Things (IoT) is a worldwide network that joins the computational capacity of cloud platforms with heterogeneous smart devices equipped with various sensors and actuators [Gut+16]. As part of IoT, smart cyber-physical systems (sCPS) are gaining increasing importance in everyday life: healthcare applications, autonomous cars, and smart robot and transportation systems are becoming more and more widespread [Cic+17]. However, they often have some safety-critical functionality: errors during operation can have serious financial consequences or cause damage to human life. The addition of smart data processing to safety-critical applications and the multidisciplinary nature of CPS make the engineering of such systems very complex.

Traditional embedded safety-critical software has requirements for ultra-high reliability which demand fault tolerance and extensive redundancy. An embedded system is often composed of federated components, where a component represents a function, such as drive-by-wire or breaking in case of a car. Each of these components is upfront separately tested and verified. The control software of the components is required to have mechanisms for synchronization, voting, and redundancy management [Rus01], which also need to be trustworthy.

Fog computing is the mainstream IoT architecture [MSW18], and it poses vastly different requirements on the underlying software. Unlike in safety-critical systems, software in this context is deployed to a dynamically changing, distributed platform with *heterogeneous devices* with various communication interfaces and *limited resources* including energy consumption, computational capacity, memory, and storage. For this reason, writing software for

IoT follows a substantially different approach in terms of platform size (fixed or varying), deployment (planned or ad-hoc), resource allocation (static or on-demand dynamic), bounded execution time (real-time or best effort), and reliability (fault tolerant or best effort) compared to traditional critical embedded systems.

However, modern CPS [Kru+15; Nie+15; Szt+12], like self-driving cars, autonomous robots, and mission critical Internet of Things (IoT) systems, like telecare or smart surveillance applications, need to comply with safety standards and regulations, while providing autonomous behavior with complex interactions with their uncertain environment using intelligent data processing techniques over a heterogeneous computation platform.

1.2 Model-Based Development of Safety-Critical Systems

Many complex design tools used for developing traditional safety-critical systems (such as Capella[1], Papyrus[2], and Artop[3]) rely internally on graph-based models, queries, and transformations. Graph models are used to efficiently capture desired structure and behavior of the system under design at a high level of abstraction, thus they provide means to manage the design time complexity of such systems. They serve as inputs for various verification and code generation activities. Furthermore, in software configuration management, models are stored in central model repositories that facilitate collaboration and version control [KH10]. In this case, models may be split into multiple artifacts for scalability reasons.

Graph queries are often used in software tools for various purposes. One of their common applications is to capture well-formedness (WF) rules which can detect errors in the design models [Cse+02]. Moreover, graph transformations, which internally rely on graph queries, are used to generate code or synthesize test data from models [Maj+19; MSM17].

However, the use of such models and formalisms has been restricted to design time, i.e., no model queries or transformations run on an aircraft at runtime currently. A main reason for this is that any piece of software executed at runtime in a safety-critical system needs to satisfy various extra-functional requirements to ensure deterministic, predictable behavior [Sta18; Aer11a; Aer11b]. Such lack of guarantees is hardly surprising. Furthermore, in a hard real-time environment, both correct and timely execution is essential, otherwise an error or a deadline miss can lead to catastrophic consequences [Rie17]. On the one hand, traditional

[1]`https://www.polarsys.org/capella/download.html`
[2]`https://www.eclipse.org/papyrus/`
[3]`https://www.artop.org/`

real-time safety-critical systems have been able to compute timeliness guarantees like worst-case execution time (WCET), on the other hand, the deployed software uses *static memory allocation* and *a priori bounded, low-level data structures* – none of which provides sufficient flexibility for modern autonomous or self-adaptive CPS.

1.3 Assurance of Smart Systems: Hypothesis and Goals

A smart CPS needs to provide autonomous behavior in an uncertain environment which is rarely known in advance, and this frequently makes design time verification infeasible in practice [LS09; MP14]. For this reason, smart and safe CPSs frequently rely on runtime assurance techniques to ensure safe operation by monitoring. These techniques have evolved from formal methods, which provide a high level of precision, but offer a low-level specification language (with atomic predicates to capture information about the system) which hinders their use in every day engineering practice. Furthermore, classical runtime assurance approaches supporting distributed systems [EF17; Dug+12; Gar96] frequently use temporal logic for specifying safety properties and expected behavior, which excel in temporal properties but they typically fail to express complex structural requirements. Recent work in the field started to exploit rule-based [Hav15] techniques over a richer (relational or graph-based) data model.

Hypothesis. Design time graph models and graph queries can be adapted to a runtime setting to provide rich knowledge representation and efficient, predictable, query-based runtime assurance for resource-constrained and distributed CPS platforms.

Runtime models (also known as models@runtime [BBF09; SZ13]) provide a rich knowledge representation to capture the runtime state of the application data, services and platforms in the form of typed and attributed graphs [Ehr+06] to serve as a unifying semantic basis for various kinds of analysis techniques. For example, runtime models have been used for the assurance of self-adaptive systems (SAS) in [Che+11; VG14]. However, the direct adaptation of existing graph-based design time techniques and tools faces several challenges imposed by their runtime use in a smart system.

Critical embedded components. Software that operates a CPS comprising heterogeneous computing units needs to track available resources of the whole execution platform (e.g., computational capacity and memory bounds in edge devices) to ensure trustworthiness and other required quality of service (QoS) guarantees. In contrast, such resources are regarded as abundant in a desktop computer or a cloud environment. For this reason, software in a CPS is often specific for a target application.

While the models@runtime initiative has been promoting the use of models, queries and transformations at runtime with major recent advances [Che+11; VG14; Har+19], *existing approaches provide no timeliness guarantees* required for any critical applications. For example, while the GreyCat framework [Har+19] claims to have "near real-time" response times for online data analytics, but there are no guarantees for timely responses. Queries over runtime models should be able to read the captured information, this way they could be used to precisely capture runtime assurance goals. Such a querying capability, as well as the model management layer itself, should be tailored to the underlying real-time environments. In other words, if a query is evaluated over a graph model, it is expected to return the results within a given (hard) deadline. We refer to queries evaluated over runtime models as *queries@runtime*.

Distributed execution platform. Unlike in CPS design tools, where models are stored in a centralized way and they evolve slowly, the underlying graph model needs to be distributed and updated with high frequency based on incoming sensor information and changes in network topology. In addition, well-known challenges of distributed systems such as data consistency and fault-tolerance need to be tackled [KRV18; Hu+20].

Existing distributed runtime models [Har+15; Har+17] support graph node-level versioning and reactive programming with lazy loading to make the complete virtual model accessible from every node over a Java-based platform. However, the use of such runtime models for analysis purposes in *resource-constrained* smart devices or critical CPS components is problematic due to the lack of control over the actual deployment of the model elements to the execution units of the platform.

Engineering aspects. Besides the challenges mentioned above, there are two additional important requirements towards graph models@runtime and queries@runtime. (1) The runtime model and query execution should be *scalable* w.r.t the number of incoming model updates, model size, query complexity, or the number of platform units. Furthermore, (2) runtime models and queries should exploit the underlying communication platform which provides scalable solutions to some aspects of the named challenges (e.g., the Data Distribution Service [Par03] deals with reliable message delivery over the network without any central message broker).

Summary of thesis objectives. To elaborate on our hypothesis, we set the following objectives in the thesis:

Ob1. Provide *precise semantics* for runtime graph models and queries in the context of runtime monitoring in CPSs.

Ob2. Ensure *predictable execution times* for updates to runtime models and executions of graph query-based monitors.

Ob3. Enable the execution of graph queries over *resource-constrained platforms.*

Ob4. Provide *scalability evaluation* of models@runtime and queries@runtime in real-time and distributed CPS.

Ob5. Implement a software prototype for evaluation and *integrate it with existing tools and frameworks.*

1.4 Research Questions and Contributions

I formulate three main research questions in this thesis to cover the goals introduced in Section 1.3. For each research question, a short summary of the contributions is given.

1.4.1 Runtime Models

The first research question focuses on graph models as runtime knowledge representation.

RQ1. How to capture and continuously maintain the state of a real-time/distributed system and its operational context in a runtime graph model?

Contributions. I propose a representation of dynamically changing graph models for real-time embedded systems, and define model update protocols with prototype implementations featuring the following characteristics:

Co1.1. *Graph data structures for embedded systems*: I provide low-level data structures for storing dynamic graph data in real-time embedded systems to support deterministic execution times for data-driven monitoring programs with changing memory needs.

Co1.2. *Model manipulation interface and protocol*: I propose a runtime model offering a high-level model manipulation interface to be used by low-level sensors and high-level domain-specific applications. To support distributed CPS, I propose a novel update protocol to guarantee consistent model updates by enforcing the *single source of truth* principle.

Co1.3. *Deployment and integration to resource-constrained platforms*: I provide a prototype implementation of the runtime model interface that is deployable over resource-constrained devices. To support distributed CPS platforms, my prototype is integrated with the Data Distribution Service standard [Par03] as a reliable underlying messaging middleware between computing units.

Co1.4. *Scalability evaluation*: I carried out a scalability evaluation of the prototype in the context of the MoDeS3 demonstrator (as a physical CPS platform) and a simulated environment with increasing number of computing units in the platform. The presented approach is able to manage up to 420K model objects stored by 20 different units.

Results focusing on distributed CPS platforms were presented in the journal article [*j*2], and the MoDeS3 educational demonstrator, which I have been working on as the lead architect for over a year, was presented in a peer-reviewed international conference [*c*6].

1.4.2 Runtime Queries

Assuming that an up-to-date runtime graph model correctly reflects the current state of the system and its environment, the second research question addressed in the thesis is about graph query execution.

RQ2. How to evaluate graph queries on runtime models deployed over a real-time/distributed platform with resource constraints?

Contributions. I adapt graph query-based runtime monitors to resource-constrained environments. Furthermore, I propose distributed query strategies on top of distributed runtime models where each model element is managed by a dedicated computing unit of the platform. In this regard, I provide the following specific contributions:

Co2.1. *Query-based runtime monitors in resource-constrained environments*: I propose a technique for automatically synthesizing deployable embedded monitoring programs from high-level query specifications.

Co2.2. *Semantic description for queries with uncertain results*: I define precise semantics of graph query evaluation over runtime graph models using *3-valued logic* [Sob52] to uniformly capture contextual and communication uncertainty.

Co2.3. *Distributed query evaluation strategies*: I define a coordinator-driven (single executor) and a decentralized (multiple executors) strategy for evaluating graph queries over distributed runtime models.

Co2.4. *Graph queries as services over resource-constrained environments*: I provide a prototype implementation of query-based distributed monitors deployed over resource-constrained environments. I integrate this prototype with a reliable messaging middleware compliant with the Data Distribution Service standard [Par03].

Co2.5. *Scalability evaluation*: I provide a performance evaluation of our distributed query technique over the physical platform of the MoDeS3 demonstrator and also assess its performance using an open query benchmark [Szá+17] over a virtual CPS platform. The evaluation results show that the approach is capable of evaluating graph queries with different complexities over a distributed plaftorm of 20 computing units.

Our results regarding distributed graph query evaluation were published at multiple peer-reviewed international conferences [*c*7; *c*5; *c*4].

1.4.3 Timing Analysis

The third research question focuses on timing analysis of query-based runtime monitors to enable their application in hard real-time environments.

RQ3. How to compute safe and practical WCET bounds for queries at runtime deployed over a real-time platform?

Contributions. I present two complementary flow analysis techniques [LS03] for the high-level WCET analysis of *query programs running on a single platform node*. To address the highly data-dependent behavior of such monitors, our key idea is to provide WCET guarantees relative to the size of the model by exploiting domain-specific characteristics of graphs.

Co3.1. *Static WCET analysis*: I propose a static design-time WCET analysis method for data-driven monitoring programs derived from graph queries. The method incorporates results obtained from low-level timing analysis into the objective function of a modern graph solver [SNV18], and then it maximizes the execution time estimate (i.e., estimates the WCET) over a given model space to provide a safe upper bound for execution time.

Co3.2. *Witness model generation*: I provide *witness models* where the monitor is expected to take the most time to complete among all models up to a predefined size bound. The run time estimate of the monitor over such models is used as the WCET estimate.

Co3.3. *On-line WCET analysis*: When the runtime graph model exceeds the boundaries of design time WCET estimation, I provide an approach to fast yet conservative on-the-fly recomputation of safe execution time bounds exploiting runtime model statistics.

Co3.4. *Evaluation of the WCET estimates*: I perform experimental evaluation of query-based programs executed over a real-time platform over a set of generated models. Moreover, I compare WCETs obtained with the different approaches to show that the static WCET estimation approach can provide more precise estimates, while the on-line estimation can be recomputed rapidly without significant loss of precision.

A part of the results presented in the thesis were published in a peer-reviewed international conference [*c*3]. A journal paper focusing on static WCET estimation and witness model generation is currently under review [*j*1].

To help the reproducibility of the results presented in this thesis, we have shared the detailed measurement results online[4].

1.5 Structure of the Thesis

This thesis consists of 10 chapters including this first introductory chapter with motivation for query-based runtime monitors, research questions, and outline of contributions. The remainder of the thesis is structured as follows.

- Chapter 2 presents MoDeS3, a demonstrator system for smart and safe railway systems. This system is used for showcasing the techniques presented in this thesis.
- Chapter 3 presents the background on various modeling concepts.
- Chapter 4 discusses the results in the context of related work.
- Chapter 5 describes how to adapt runtime graph models to real-time embedded systems.
- Chapter 6 discusses the adaptation of query-based monitors to real-time embedded systems.
- Chapter 7 proposes WCET analysis techniques for query-based monitoring programs.
- Chapter 8 discusses a *distributed* runtime graph model management protocol.
- Chapter 9 presents our distributed query evaluation framework.
- Chapter 10 provides concluding remarks and describes future research directions.

At the end of each chapter, the Summary section briefly highlights the novel contributions of the chapter, and discusses the contribution of authors to the presented results and publi-

[4]`https://imbur.github.io/cps-query/`

cations. In such descriptions, first person singular is used to emphasize contributions made by the author of this thesis, while the role and contribution of authors (excluding my Ph.D. supervisor) are discussed in detail. Otherwise, the use of first person plural highlights the collaborative nature of the work leading to the contribution.

Chapter 2

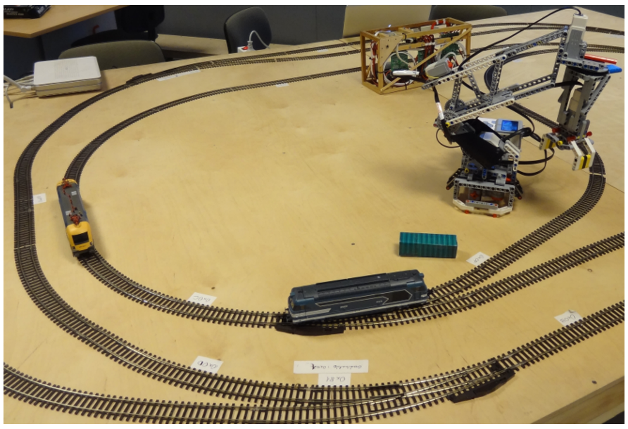
Motivating Case Study

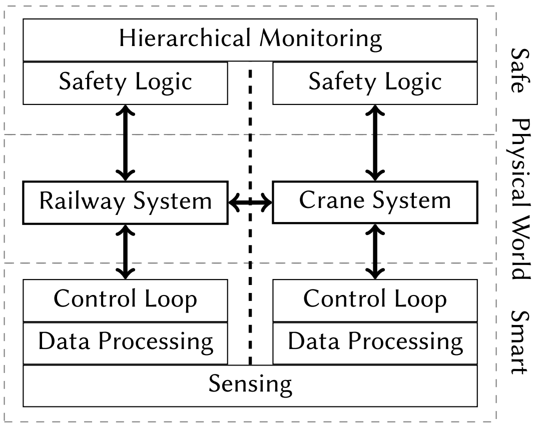
Our work shows several motivating examples where our runtime monitoring framework is employed. These examples are all related to each other and are part of a common exemplar of modern cyber-physical systems. This chapter presents this demonstrator system used in illustrations throughout the thesis. We provide a general overview of MoDeS3 in Section 2.1, and discuss the showcased design time and runtime assurance techniques in Section 2.2.

2.1 MoDeS3: a Demonstrator for Smart and Safe CPS

Both research and education of CPSs necessitate well-documented open-source demonstrator platforms which capture and reflect the essence of problems and challenges, yet are reasonably complex to highlight the key characteristics of CPSs and present them in the context of modern technologies. We introduce *MoDeS3: the Model-based Demonstrator for Smart and Safe Cyber-Physical Systems*, which aims to illustrate the combined use of model-driven development, intelligent data processing, safety engineering, and IoT technologies in the context of safety-critical system of systems with emerging safety hazards. This open-source project simultaneously serves as (1) a *research platform used for experimental evaluation* of CPS-related research, (2) a complex *educational platform* used for graduate and undergraduate teaching, and (3) an *IoT technology demonstrator* used by industrial partners and collaborators.

The MoDeS3 demonstrator as a smart and safe CPS. The physical layout of MoDeS3 is depicted in Figure 2.1a. At its core, there is a *model railway transportation system*, for which guarantees for the safe operation of trains, switches, and semaphores are required. Connected to a specific segment of the track, an automated *crane system* loads cargo on and off the trains. As such, it is a critical system in itself since the cargo cannot be dropped by the crane.


(a) Physical layout



(b) Architectural overview

Figure 2.1: The MoDeS3 CPS demonstrator system

Additionally, the MoDeS3 demonstrator represents a system-of-systems, since the railway and the crane system are physically located next to each other. In this case, new kind of hazardous situations may emerge which are not incorporated in any of the constituent systems. For instance, the rotating crane may collide with a train passing by along the track.

The computing platform of the MoDeS3 CPS demonstrator is built from industrial-grade hardware components. These components are often used in *embedded systems* applications, where processor, memory, and peripherals are tightly coupled with other electronic hardware to provide a dedicated function. Moreover, several components are used in *safety-critical* applications, where system failure can lead to catastrophic consequences.

To make the demonstrator more realistic, we adopted various safety assurance techniques ranging across design time formal verification and validation (V&V), runtime monitoring or testing on various levels of abstraction. A conceptual overview is provided in Figure 2.1b. Multiple levels of safety are applied: a distributed safety logic is responsible for the accident-free operation of the trains. Hierarchical monitoring is used to ensure the safe cooperation of the subsystems. A wide range of sensors serves as a rich information source for smart control and data analytics.

In a safety-critical application it is important to ensure that the software is able to provide the results by specific deadlines. Delayed responses in many cases are considered as errors and can have catastrophic consequences. For this reason, safety monitoring components of MoDeS3 are analyzed to ensure deterministic timing behavior.

In the following, we focus on the assurance techniques applied in the demonstrator. Since this work positions MoDeS3 as the motivating case study for data-driven runtime monitoring, we provide a detailed overview of this approach and its application.

2.2 Design- and Runtime Assurance

The development of safety-critical systems has a long history with well-established methodologies to ensure safe operation. The MoDeS3 demonstrator is built using Model-Based Systems Engineering (MBSE) where models are first-class citizens of the engineering process. SysML models are used to define the functional and the platform architecture of the system, while the Gamma Statechart Composition Framework [Mol+18] is used for the precise definition of the component level behavior. Gamma supports the design, verification and code generation for component-based reactive systems.

The MoDeS3 demonstrator incorporates various V&V approaches (such as model checking, structural completeness and consistency analysis) as well as fault-tolerance techniques — all of which are widely used in real systems. However, due to its complex and multidisciplinary nature, design time assurance cannot guarantee in itself the safe operation of inherently dynamic smart CPSs. Therefore, runtime certification [Rus08] using techniques like runtime monitoring [Med+15] or runtime verification [Hav15] complement design time assurance. Therefore, MoDeS3 integrates runtime monitoring and verification techniques on both component and system-level to flag violations of safety properties during the operation of the system and trigger appropriate counter-measures such as immediately stopping or slowing down trains. Our emphasis is on the combined use of design time and runtime V&V techniques when building MoDeS3 to address its safety requirements.

2.2.1 Design Time Formal V&V of Timing Properties

As a primary design time verification task, we carried out a formal analysis of logical and timing properties of the distributed safety logic of the accident prevention subsystem. We employed the Gamma Statechart Composition Framework [Mol+18] to form the composite behavior of statechart models. This composite model serves as the engineering input for the design time analysis. Gamma introduces an intermediate state machine language with some high-level constructs and precisely defined semantics [Tót+14] to serve as a bridge between engineering and formal models. This intermediate language also helps in the back-annotation

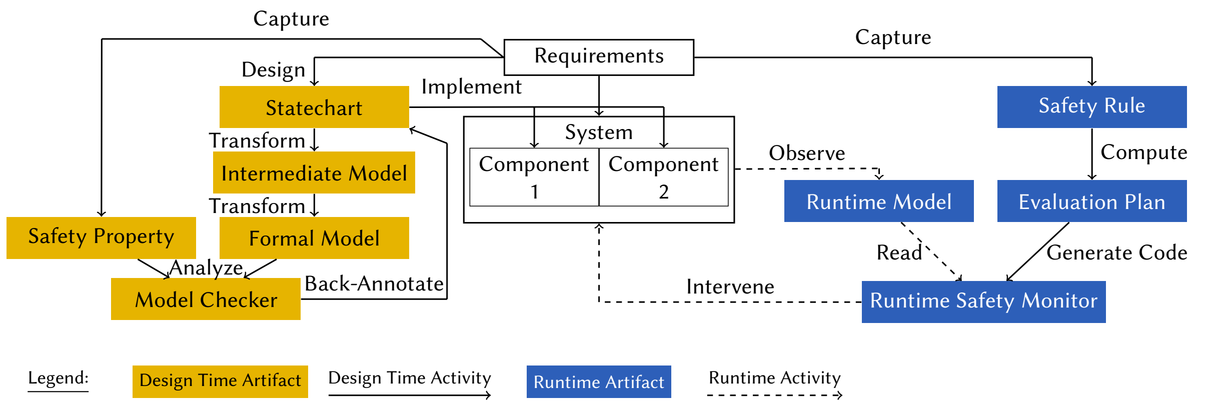


Figure 2.2: Overview of runtime verification in MoDeS3

of analysis results to statechart models. Formal verification is performed using the model checker UPPAAL [Beh+06], which is widely used for verifying timing properties.

The generated formal models address the verification of a single component against local properties as well as their interaction against global properties. However, these models are insufficient to reason about the correctness of the system in themselves. For that purpose, one needs to ensure the interaction between the physical world and the cyber world.

For this purpose, formal models are built to capture the (logical and physical) behavior of trains. Then a combined design time verification can reveal potentially unsafe situations, e.g., if trains move too fast, some accidents cannot be prevented. Investigating the counterexample retrieved by Gamma highlights that the situation could only happen if the trains are faster than the messages transmitted between the components. Unless there is a denial-of-service attack with flooding of messages, this is hardly the case in practice, but it is still a potential security threat. After extending the statechart models with timing assumptions on communication speed, we can formally prove that the safety logic prevents multiple trains from entering the same section of the track.

2.2.2 Runtime Safety Monitors

As smart and safe CPSs have complex interactions with an evolving environment and the physical world, we complement design time verification in MoDeS3 with runtime monitoring techniques on both component and system level. We provide here a summary of the *hierarchical system-level runtime monitoring* technique using graph reasoning with runtime models (see right part of Figure 2.2).

As traditional monitoring techniques handle events but do not cover data-dependent behavior or structural properties, runtime knowledge about the operational system is captured by a runtime model [BBF09]. A runtime model captures the current abstract snapshot of the system and its operational context, and changes in the underlying running system are constantly incorporated. Unlike a detailed design model, a runtime model only captures those aspects of the system, which are relevant for runtime monitoring and intervention. We define *runtime graph models* later in Section 3.1.1.

System-level safety monitoring is carried out using *runtime graph queries*, which detect runtime violations of safety rules (by the identification of changes in the match sets of graph queries) and trigger appropriate reactions. While graph models and queries are widely used in *design tools* of CPS, their use in the context of smart and safe CPS is an innovative aspect of the MoDeS3 demonstrator.

The right part of Figure 2.2 shows the steps during the design phase of automated monitor synthesis where *high-level query specifications are transformed into deployable, platform dependent source code* for each computing unit that will be executed as part of a monitoring service. We present a local search-based query evaluation strategy later in Section 3.2 and introduce a code generation approach from high-level specifications in Section 6.2. MoDeS3 reuses a *high-level graph query language, the VIATRA Query Language (VQL)* [Var+16] *for specifying safety properties of runtime monitors*, which is widely used in various design tools of CPS [Szt+14].

Example 1. In MoDeS3, there are several query-based monitors providing continuous feedback about the state of the system as well as check for violations of safety rules. We introduce them in increasing order of the length of their specification in VQL (see these specifications in Appendix A). These queries are used later in the thesis in the evaluation sections Section 6.3 and Section 9.3.

- **Train locations** (**tl**): A simple query to find pairs of trains and segments that describe the locations of each train.
- **End of siding** (**eos**): This query finds trains that are dangerously close (one segment distance) to an end of the track.
- **Close trains** (**ct**): The headway distance needs to be respected on the track, and this query highlights locations where two trains are only one free segment away from each other.

- **Misaligned turnout (mt)**: This monitoring rule detects a train approaching a turnout but the turnout is set to the other direction (causing the train to run off from the track).

2.2.3 Hierarchical Distributed Monitors

Figure 2.3 illustrates a self-contained excerpt of the demonstrator at runtime in which the model railway system has an added layer of safety to prevent trains from colliding and derailing with the help of such data-driven runtime safety monitors. The system is managed by a distributed monitoring service running on a network of heterogeneous computing units, such as Arduinos, Raspberry Pis, BeagleBone Blacks, etc. In Figure 2.3, three of such computing units are running the monitoring and controlling programs responsible for managing the different (disjoint) parts of the system. A computing unit may read its local sensors, (e.g., the occupancy of a segment, or the status of a turnout), collect information from other computing units, and it can operate actuators accordingly (e.g., change turnout state). All this information is reflected in a *distributed runtime model* which is deployed on the three computing units.

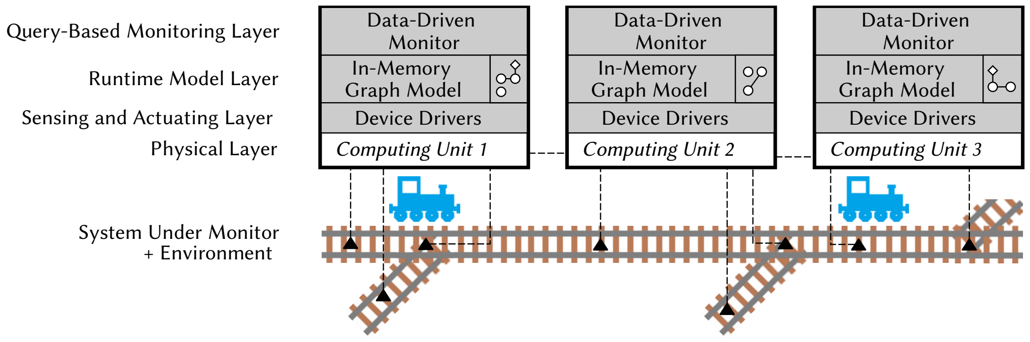


Figure 2.3: Runtime monitoring by graph queries in the MoDeS3 demonstrator

Our system-level runtime monitoring framework is *hierarchical* and *distributed*. Monitors executing graph queries may observe the local runtime model of a participant, and they can collect information from runtime models of different devices, hence providing a distributed architecture. Moreover, one monitor may rely on information computed by other monitors, thus yielding a hierarchical network.

Alerts from the monitoring services may trigger control commands of actuators (e.g., to change turnout direction) to guarantee safe operation. The monitoring and control programs are running in a real-time setting on the computing units.

Graph query-based runtime monitoring nicely complements traditional, component-level, automaton-based monitors deployed to embedded computers since critical signals raised by low-level monitors can be further propagated to the system-level as a hierarchy of events. As a consequence, we obtain a technique for the runtime monitoring of system-of-systems [Vie+16] where emerging and ad hoc hazardous situations can be incorporated and detected automatically also in the presence of complex structural constraints.

2.2.4 Timing Analysis of Real-Time Monitors

Query-based monitoring programs use a network of linked objects as data structures and *exhibit heavily input-dependent complex control and data flow.* While *safety-critical programs typically use statically allocated data with bounded input sizes* and they conservatively avoid many programming constructs, dynamically evolving data and advanced language constructs are inherent parts of data-intensive programs which can jeopardize timing predictability.

A wide range of existing timing analysis techniques (implemented in various tools such as aiT [FH04], Chronos [Li+07], OTAWA [CS06], and SWEET [Lis14]) can provide safe and tight WCET bounds for traditional critical embedded software. The most common technique used in state-of-the-art timing analyzers is the implicit path enumeration technique (IPET) [LM95] which relies on solving an integer linear program (ILP) at design time. However, there is a high degree of inherent design time uncertainty present in query-based monitoring programs. In particular, the unknown contents of the runtime knowledge graph capturing the system and its operating environment constitutes an enormously large input space which can compromise the accuracy of existing techniques.

To support the timing analysis of runtime monitors of MoDeS3, we combine WCET analysis with advanced graph solvers to efficiently find inputs up to a predefined size where the monitor is expected to take the longest to execute (referred to as *witness model*) and use the computed execution time over this model as the WCET. Furthermore, we provide a complementary approach to support cases where the runtime model exceeds the size of the witness model. In this latter case, we rely on a symbolic WCET formula which is instantiated using condensed model statistics about the snapshot of the underlying runtime model.

2.3 Summary

Modern smart and safe CPSs efficiently combine intelligent data processing features with safety-critical functionality. MoDeS3 is an educational demonstrator of an intelligent railway system, which was developed to showcase challenges present in the design and operation of such systems. This thesis relies on the demonstrator as a case study for data-driven runtime monitors. The MoDeS3 educational demonstrator was first presented in [*c*6], and I took part in the development of the hardware and software as lead architect for one year prior to joining McGill University, and have also made major contributions to the hierarchical runtime monitoring software component. Several students and colleagues participated in this project, and the coauthors of the related the paper had the following responsibilities. András Vörös was leading the project. István Ráth, Ákos Horváth, Zoltán Micskei, and Dániel Varró helped the work as advisors and provided ideas about what the demonstrator should feature. László Balogh, Benedek Horváth, and Zsolt Mázló worked on the implementation. Despite my contributions, due to the highly collaborative nature of the work, I do not claim novel scientific contributions related to the core MoDeS3 demonstrator platform.

Chapter 3

Background for Query-Based Monitors

The present chapter overviews core concepts used throughout this thesis. Definitions included here come from the fields of domain-specific modeling, graph queries, and domain-specific graph model generation.

The first part in Section 3.1.1 provides formal definitions for metamodels and instance models and Section 3.1.2 describes basic model update operations, both of which are necessary for Chapter 5 and Chapter 8, then Section 3.1.3 shows how one can formulate graph queries using first-order logic. Section 3.2 discusses the local search-based graph query evaluation algorithm employed in Chapter 6 and Chapter 9. Section 3.3 gives an overview of generating well-formed models for a domain which we use as a technique to innovatively derive worst-case execution time bounds for query-based runtime monitoring programs in Chapter 7. Finally Section 3.4 introduces the Data Distribution Service which is the underlying communication standard in our distributed runtime model and distributed query evaluation framework introduced later in Chapter 8 and Chapter 9. Figure 3.1 depicts the concepts and their usage in this thesis.

3.1 Runtime Models

Runtime models serve as a rich knowledge base for the system by capturing the runtime status of the domain, services, and platforms as a graph model, which serves as a common basis for executing various analysis algorithms. The following sections provide the fundamental definitions required to introduce runtime models and queries over them.

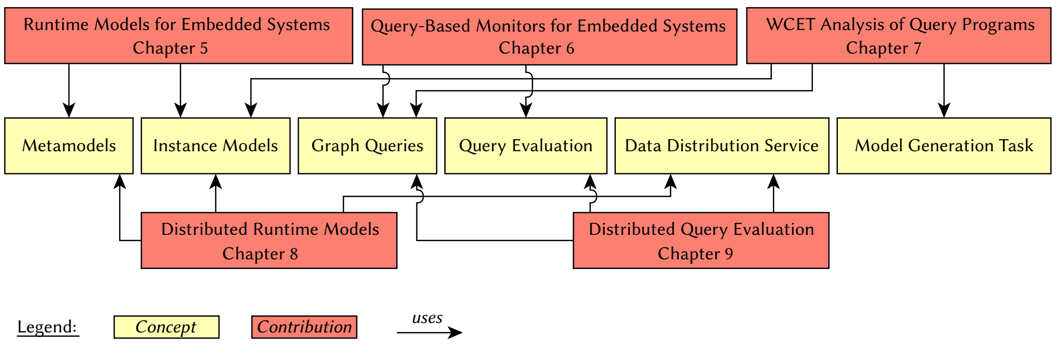

Runtime Models for Embedded Systems Chapter 5
Query-Based Monitors for Embedded Systems Chapter 6
WCET Analysis of Query Programs Chapter 7
Metamodels
Instance Models
Graph Queries
Query Evaluation
Data Distribution Service
Model Generation Task
Distributed Runtime Models Chapter 8
Distributed Query Evaluation Chapter 9
Legend:
Concept
Contribution
uses

Figure 3.1: Concepts introduced in this chapter and their usage later in the thesis

3.1.1 Metamodels and Instance Models

The core concepts (classes) in a domain, their attributes, and the relations (references) between those concepts are often captured in a *metamodel*. Additionally, a metamodel can include derived features (i.e., computed types which are based on other domain elements) represented by model queries. In this thesis, we formally capture metamodels by a logic signature and instance models as logic structures following [MSV18].

Definition 1 (Metamodel)**.** A *metamodel* is formally represented as a logic signature $\Sigma = \{\mathtt{C}_1, \dots, \mathtt{C}_m, \mathtt{A}_1, \dots, \mathtt{A}_n, \mathtt{R}_1, \dots, \mathtt{R}_o, \mathtt{q}_1, \dots, \mathtt{q}_p\}$ and an arity function $\alpha \colon \Sigma \to \mathbb{N}$, where $\{\mathtt{C}_i\}_{i=1}^{m}$ are unary *class symbols* (with $\alpha(\mathtt{C}_i) = 1$), $\{\mathtt{A}_j\}_{j=1}^{n}$ are binary *attribute symbols* (with $\alpha(\mathtt{A}_j) = 2$), $\{\mathtt{R}_k\}_{k=1}^{o}$ are binary *relation symbols* (with $\alpha(\mathtt{R}_k) = 2$), and $\{\mathtt{q}_l\}_{l=1}^{p}$ are n-ary *query names* (with $\alpha(\mathtt{q}_l) \in \mathbb{N}$).

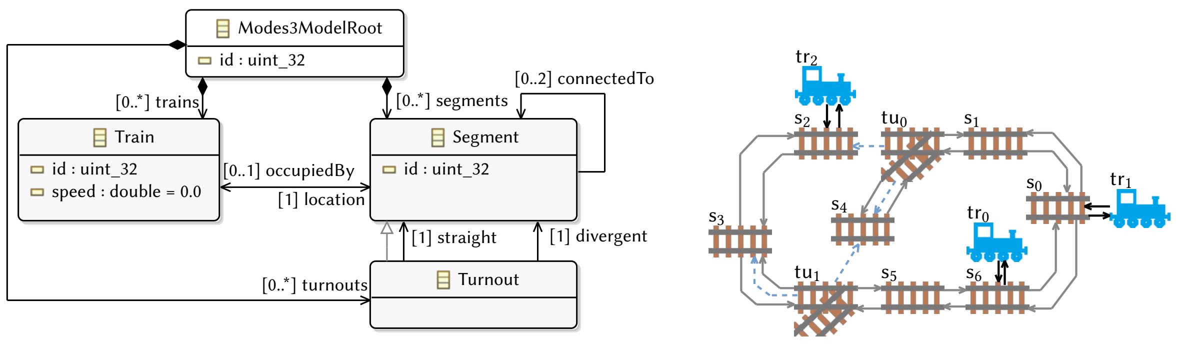
Definition 2 (Instance model)**.** An *instance model* over a metamodel Σ is a logic structure $M = \langle \mathcal{D}_M, \mathcal{I}_M \rangle$ where $\mathcal{D}_M = \mathcal{O}_M \sqcup \mathcal{V}_M$ is the model domain where $\mathcal{O}_M$ is a finite set of objects in M while $\mathcal{V}_M$ is the set of (built-in) data values (integers, strings, etc.). $\mathcal{I}_M$ provides interpretations for the class, attribute, and reference predicates in Σ such that

- $\mathcal{I}_M(\mathtt{C}_i) \subseteq \mathcal{O}_M$ is the set of objects of type $\mathtt{C}_i$ for each $\mathtt{C}_i \in \Sigma$,
- $\mathcal{I}_M(\mathtt{A}_j) \subseteq \mathcal{O}_M \times \mathcal{V}_M$ is the set of objects with attribute values of type $\mathtt{A}_j$ for each $\mathtt{A}_j \in \Sigma$, and
- $\mathcal{I}_M(\mathtt{R}_k) \subseteq \mathcal{O}_M \times \mathcal{O}_M$ is the set of relation links of type $\mathtt{R}_k$ for each $\mathtt{R}_k \in \Sigma$.

Based on the models@runtime paradigm [BBF09; SZ13], *runtime model* refers to the continuously updated graph data structure that serves as the up-to-date knowledge base about the system. At any given point, a *snapshot* of the runtime model is accessible, and this snapshot can be represented by an instance model. In contexts where it is not confusing, we may use the term *runtime model* as a shorthand for *snapshot of the runtime model.*

To provide a condensed characterization of instance models, we will collect various *model statistics* at runtime. For simplicity, we will restrict our attention to the type distributions (number of objects of each type).

Definition 3 (Model statistics)**.** The *model statistics* for an instance model M is a function $stats_M$:
$\{\mathtt{C}_1, \ldots, \mathtt{C}_l\} \rightarrow \mathbb{N}$ which denotes the number of objects of type $\mathtt{C}_i$, i.e. $stats_M(\mathtt{C}_i) = |\mathcal{I}_M(\mathtt{C}_i)|$.



(a) Metamodel with metamodel constraints

(b) System snapshot presented as an instance model with the following model statistics: 7×Segments, 2×Turnouts, 3×Trains

Figure 3.2: The MoDeS3 metamodel and instance model

Example 2. An excerpt of the MoDeS3 metamodel with metamodel constraints is shown in Figure 3.2a using the Eclipse Modeling Framework (EMF) notation[1]. A model has exactly one Modes3ModelRoot that contains all other objects within the model (as suggested by the containment references). One domain concept is Train. Class Segment represents a section of the railway track with the connectedTo reference which describes what other segments it is linked to (up to two). Moreover, each train maintains a location reference to a segment to describe its current position, while the direction of a train is not captured in the model to keep the presentation of the example short. Likewise, instances of the Segment class maintain a reference occupiedBy to express if they are currently occupied by a train. Moreover, a Turnout is a Segment that can change its connections between straight and divergent segments. In this example, {`Modes3ModelRoot`, `Train`, `Segment`, `Turnout`} $\subset \Sigma$ are unary class predicates, while {`Location`, `OccupiedBy`, `ConnectedTo`, `Divergent`, `Straight`, `Trains`, `Segments`, `Turnouts`} $\subset \Sigma$ are binary reference predicates. Furthermore, {`Id`, `Speed`} $\subset \Sigma$ are binary attribute predicates which describe if a model object stores a given value as its respective attribute.

Figure 3.2b shows a graphical presentation of an instance model capturing a snapshot of the MoDeS3 demonstrator with the following model statistics. The graph has a total of 12 objects. There are 9 Segment instances with their respective connectedTo references depicted as gray arrows, and two of them are also instances of Turnout. The turnout represented by tu_0 is capable of switching between segments s_2 and s_4, while tu_1 is capable of switching between segments s_3 and s_4, which shown by the dashed blue arrows. Additionally, there are three trains on the track $\mathsf{tr}_{0\ldots2}$ with their respective locations being s_6, s_0, and s_2 as marked by the black arrows.

3.1.2 Update Operations on Runtime Models

In order to be able to capture the contextual information of the underlying system, runtime graph models need to support the following update operations:

- **Object create**: This operation adds a new object o_{new} to the graph model. Formally, a new $o_{new} \in \mathcal{O}_M$ is created and the interpretation of its class symbol $\mathtt{C}_i$ is changed such that $o_{new} \in \mathcal{I}_M(\mathtt{C}_i)$.
- **Object delete**: This operation is the inverse of *object create*. It removes an object o_{delete} from the graph model. Formally, $o_{delete} \notin \mathcal{O}_M$ is enforced and the interpretation of its class symbol $\mathtt{C}_i$ is changed such that $o_{delete} \notin \mathcal{I}_M(\mathtt{C}_i)$.
- **Attribute update**: The old value $v_{old} \in \mathcal{V}_M$ of the attribute $\mathtt{A}_j$ of object $o \in \mathcal{O}_M$ is replaced with $v_{new} \in \mathcal{V}_M$ when this operation completes. Formally, $\langle o, v_{old}\rangle \notin \mathcal{I}_M(\mathtt{A}_j)$ and $\langle o, v_{new}\rangle \in \mathcal{I}_M(\mathtt{A}_j)$ are enforced.
- **Link add**: This operation creates a link of type $\mathtt{R}_k$ between $o_{src} \in \mathcal{O}_M$ and $o_{trg} \in \mathcal{O}_M$. Formally, $\langle o_{src}, o_{trg}\rangle \in \mathcal{I}_M(\mathtt{R}_k)$ is ensured.
- **Link remove**: This operation is the inverse of *link add*, and it removes a link of type $\mathtt{R}_k$ between $o_{src} \in \mathcal{O}_M$ and $o_{trg} \in \mathcal{O}_M$. Formally, $\langle o_{src}, o_{trg}\rangle \notin \mathcal{I}_M(\mathtt{R}_k)$ is ensured.

[1]`http://www.eclipse.org/emf`

Before each model update, the model manipulation middleware needs to ensure the appropriate consistency level. Applications that have higher consistency criteria may refuse a requested operation if the resulting model would violate a *well-formedness constraint*. Such well-formedness constraints are introduced in the subsequent section.

3.1.3 First-Order Logic Predicates for Queries Over Graph Models

First-order logic (FOL) predicates formulated using the metamodel symbols in Σ can be evaluated as *graph queries* over the logic structure of an instance model. Informally, base predicates check either for equality or for the existence of certain objects, attribute values, and references of a respective type (predicate) in the underlying graph model. Then complex predicates are derived by traditional FOL connectives (*not*, *exists*, *forall*, *and*, and *or*). Query hierarchies can be composed via *query calls*.

Definition 4 (Syntax of graph predicates). *First-order graph predicates* can be inductively constructed using the following rules.

- The constants 1 and 0 are atomic predicates.
- $\mathtt{C}_i$, $\mathtt{A}_j$, and $\mathtt{R}_k$ are atomic predicates.
- If φ_1 and φ_2 are predicates and u and v are variables, then the following expressions are predicates:

$$u = v, \qquad \neg\varphi_1, \qquad \exists v\colon \varphi_1, \qquad \forall v\colon \varphi_1, \qquad \varphi_1 \vee \varphi_2, \qquad \text{and} \qquad \varphi_1 \wedge \varphi_2.$$

Definition 5 (Graph query). A query $\mathtt{q} \in \Sigma$ is defined by the first-order logic predicate $\varphi_{\mathtt{q}}(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ denote free variables (not appearing in any quantifiers) of $\varphi_{\mathtt{q}}$.

Definition 6 (Variable binding). A *variable binding* $Z\colon \{v_1, \ldots, v_n\} \rightarrow \mathcal{D}_M$ is a mapping between variables of a predicate and elements of the model domain. A variable binding Z_p is a *partial binding* if it maps only a subset of the variables of a predicate to $\mathcal{D}_M$.

Definition 7 (Semantics of graph predicates). A graph predicate can be evaluated over an instance model M along a variable binding $Z\colon \{v_1, \ldots, v_n\} \rightarrow \mathcal{D}_M$ (denoted as $[\![\varphi_{\mathtt{q}}]\!]^M_Z$)

to return either *true* (1) or *false* (0) as follows:

$$\begin{array}{rlrl}
[\![1]\!]^M_Z := 1 & & [\![0]\!]^M_Z := 0 \quad [\![\mathtt{C}_i(v)]\!]^M_Z & := 1 \text{ iff } Z(v) \in \mathcal{I}_M(\mathtt{C}_i) \\
[\![\mathtt{A}_j(u,v)]\!]^M_Z & := 1 \text{ iff } \langle Z(u), Z(v)\rangle \in \mathcal{I}_M(\mathtt{A}_j) & [\![\mathtt{R}_k(u,v)]\!]^M_Z & := 1 \text{ iff } \langle Z(u), Z(v)\rangle \in \mathcal{I}_M(\mathtt{R}_k) \\
[\![u = v]\!]^M_Z & := 1 \text{ iff } Z(u) = Z(v) & [\![\neg\varphi]\!]^M_Z & := 1 - [\![\varphi]\!]^M_Z \\
[\![\exists v : \varphi]\!]^M_Z & := \max_{x\in O_M}\{[\![\varphi]\!]^M_{Z,v\mapsto x}\} & [\![\forall v : \varphi]\!]^M_Z & := \min_{x\in O_M}\{[\![\varphi]\!]^M_{Z,v\mapsto x}\} \\
[\![\varphi_1 \vee \varphi_2]\!]^M_Z & := \max([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z) & [\![\varphi_1 \wedge \varphi_2]\!]^M_Z & := \min([\![\varphi_1]\!]^M_Z, [\![\varphi_2]\!]^M_Z)
\end{array}$$

$$[\![\mathtt{q}(v_1,\dots,v_n)]\!]^M_Z := \exists Z' : Z \subseteq Z' \wedge \forall_{i\in\{1..n\}} Z(v_i) = Z'(v^c_i) : [\![\varphi_\mathtt{q}(v^c_1,\dots,v^c_n)]\!]^M_{Z'}$$

Definition 8 (Predicate evaluation). *Graph predicate* (or *query*) *evaluation* aims to find all variable bindings $Z\colon \{v_1,\dots,v_n\} \to O_M$ for a predicate φ that maps all free variables of the predicate to objects of M such that the predicate evaluates to *true*, i.e., $[\![\varphi]\!]^M_Z = 1$.

Definition 9 (Match set). The *match set* of a query predicate φ with free variables $v_1,\dots,v_n$ is

$$\mathit{Matches}(M,\varphi) = \{Z\colon \{v_1,\dots,v_n\} \to O_M \mid [\![\varphi]\!]^M_Z = 1\}.$$

One element in this set is called a *match*, while $|\mathit{Matches}(M,\varphi)|$ denotes the size of the match set.

Note that in our context, a match of a query will typically represent a violation of a well-formedness constraint of the domain or a hazardous situation with respect to a safety property.

Metamodel and well-formedness constraints

A domain metamodel is frequently complemented in practice with additional metamodel and well-formedness (WF) constraints to restrict the possible relationships between domain concepts. *Metamodel constraints* can be captured by FOL predicates and categorized as follows [MSV18].

1. A **type hierarchy constraint** defines a type system by *supertype* relations. For each object o, there shall be a single class $\mathtt{C}$, such that for any class object o is instance of $\mathtt{C}'$ iff $\mathtt{C}'$ is a supertype of $\mathtt{C}$.
2. A **type compliance constraint** restricts the classes $\mathtt{C}_1$ and $\mathtt{C}_2$ of objects at the ends of a reference $\mathtt{R}$: $\forall o_1, o_2 : \mathtt{R}(o_1, o_2) \to \mathtt{C}_1(o_1) \wedge \mathtt{C}_2(o_2)$.

3. A **multiplicity constraint** may be placed on lower bounds on the number of references adjacent to an object o: $\exists o_1 \dots o_l : \mathtt{R}_j(o, o_1) \land \dots \land \mathtt{R}_j(o, o_l) \land (\exists i, j \in \{1 \dots l\} : o_i = o_j \leftrightarrow i = j)$. Similarly, the same can be said for required upper bounds for a reference adjacent to o: $\exists o_1 \dots o_u : (\mathtt{R}_j(o, o_1) \land \dots \land \mathtt{R}_j(o, o_u) \land (\exists i, j \in \{1 \dots u\} : o_i = o_j \leftrightarrow i = j)) \rightarrow \nexists o' : \mathtt{R}_j(o, o') \land o_1 \neq o' \land \dots \land o_u \neq o'$
4. An **inverse relation constraint** prescribes that references $\mathtt{R}$ and $\mathtt{R}'$ always occur in pairs: $\forall o_1, o_2 : \mathtt{R}(o_1, o_2) \leftrightarrow \mathtt{R}'(o_2, o_1)$.
5. A **containment hierarchy constraint** ensures that models are arranged in a strict tree hierarchy via the *containment references* starting from a root object.

Example 3. The formula $\varphi_{\mathtt{LM}}$ illustrates a constraint of the metamodel for *location multiplicity* (LM). This formula evaluates to 1 for an object passed as a parameter if it is of type Train and it does not have exactly has one segment as its location (i.e., has zero or more than one). A query can be defined with the following formula: $\varphi_{\mathtt{LM}}(t) = \mathsf{Train}(t) \land (\neg(\exists s_1, s_2 : \mathsf{Location}(t, s_1) \land \mathsf{Location}(t, s_2) \rightarrow s_1 = s_2) \lor \neg(\exists s : \mathsf{Location}(t, s))$.

WF constraints of the domain can also be captured by FOL predicates [SV17; SNV18]. When a constraint is formalized as a FOL predicate, it captures erroneous model fragments. As such, we expect that the respective FOL predicate has empty match sets in a model. Formally, if a WF constraint captured by FOL predicate φ, then $Matches(M, \varphi) = \emptyset$ for all well-formed instance models M.

Example 4. The metamodel constraints of the MoDeS3 domain shown in Figure 3.2a allow the creation of a model that has no real-life counterparts. For example, metamodel constraints allow the creation of a Turnout that has two connectedTo references to two distinct Segments, but none of these Segments are the continuation of the Turnout in the straight or divergent directions. Such a turnout in an instance model would represent a physically impossible situation, therefore we exclude such cases from our analysis by introducing a WF constraint `TurnoutWithErroneousConnections` captured by the following FOL predicate:

$$\varphi_{\mathtt{TEC}}(t) = \mathtt{Turnout}(t) \land \exists s_1, s_2 : \mathtt{ConnectedTo}(t, s_1) \land \mathtt{ConnectedTo}(t, s_2) \land \neg(s_1 = s_2) \\ \land \neg(\mathtt{StraightOrDivergent}(t, s_1)) \land \neg(\mathtt{StraightOrDivergent}(t, s_2)),$$

where the called subquery `StraightOrDivergent` defined by the formula φ_{SD} checks if a Segment is a straight or divergent direction of a Turnout:

$$\varphi_{\text{SD}}(t,s) = \texttt{Turnout}(t) \land \texttt{Segment}(s) \land (\texttt{Straight}(t,s) \lor \texttt{Divergent}(t,s)).$$

Support for model constraints in modeling tools

In many industrial modeling tools, domain-specific WF constraints are captured either as OCL invariants [The14] or as graph queries [VB07; NNZ00]. We use FOL to formalize such constraints, which can be efficiently evaluated by underlying query engines like [Ujh+15] to validate models. The prototype implementations presented in this work rely on the syntax of the VIATRA Query Language (VQL) [Ber+11] in the definition of graph queries.

The expressiveness of the VQL converges to first-order logic with transitive closure, thus it provides a rich language for capturing a variety of complex structural conditions and dependencies between various entities in a graph model.

Example 5. Listing 3.1.1 shows the query `TurnoutWithErroneousConnections` formulated in VQL. It is defined by the predicate φ_{TEC} from the previous example to identify illegal Straight or Divergent continuations of a Turnout is shown below using the syntax of VQL. The formulation of the query employs a subquery `StraightOrDivergent` that accepts pairs of a Turnout and a Segment that are either in Straight or in Divergent connection. Comments next to each line show the corresponding FOL predicate.

```
pattern TurnoutWithErroneousConnections(t: Turnout) {         // φTEC(t) = Turnout(t)
  Segment.connectedTo(t, s1);                                 //    ∧∃s1, s2 : ConnectedTo(t, s1)
  Segment.connectedTo(t, s2);                                 //    ∧ConnectedTo(t, s2)
  s1 != s2;                                                   //    ∧¬(s1 = s2)
  neg find StraightOrDivergent(t, s1);                        //    ∧¬(StraightOrDivergent(t, s1))
  neg find StraightOrDivergent(t, s2);                        //    ∧¬(StraightOrDivergent(t, s2))
}
private pattern StraightOrDivergent(t: Turnout, s: Segment) { // φSD(t, s) = Turnout(t) ∧ Segment(s)
  Turnout.straight(t, s);                                     //    Straight(t, s)
} or {                                                        //    ∨
  Turnout.divergent(t, s);                                    //    Divergent(t, s)
}
```

Listing 3.1.1: Query capturing illegal connections of a turnout

3.2 Local Search-Based Graph Query Evaluation

Graph query evaluation or graph pattern matching (see Definition 8) is the process of finding all matches of a query over a specific model [VAS12; Var+15]. When query evaluation is initiated, an initial variable binding is gradually extended to retrieve matches from the model. In the distinguished case, when this initial variable binding is empty (i.e., does not map any free variables of the query to any element of the domain set $\mathcal{D}_M$), query evaluation seeks matches from the entire model.

Various query evaluation strategies exist in the literature [Gal06]. Our runtime monitoring framework uses a *local search-based query evaluation* strategy to find matches of monitoring queries based on [Var+15]. To obtain efficient performance at runtime, query evaluation is guided by a *search plan* [Var+15], which maps each predicate in the query to a single pair of $\langle$*Step number*, *Operation type*$\rangle$. In this tuple, the first value specifies the order in which query evaluation should attempt to satisfy the respective predicate. The second value can be either *extend* or *check*, depending on the current variable binding while the predicate is evaluated:

- An **extend operation** evaluates a predicate with at least one free variable. Execution of such operations requires iterating over all potential variable substitutions and selecting the ones for which the predicate evaluates to 1.
- A **check operation** evaluates a predicate with only bound variables. Execution of such operations determines if the constraint evaluates to 1 over the actual variable binding.

The calculation of query search plans is out of scope of the current work, but they should be created and optimized based on domain-specific information about the model statistics of the models queries are evaluated on. Search plans were shown to be highly efficient if they are updated as the properties of the undelying model changes [Var+15].

Example 6. Table 3.1 shows a possible search plan for the `TurnoutWithErroneousConnections` query. Each row represents a search operation. The first column is the assigned operation number (Index). The second column (Predicate) shows which predicate is evaluated by the given step and the third column shows the variables that are already bound by the previous operations when the current operation begins execution. The fourth column shows the search operation type (check or extend) which is based on the variable bindings prior to the execution of the search operation: if the predicate parameters are all bound, then it is a check, otherwise, it is an extend. For extend operations, we underline the variable that is bound by the step.

Table 3.1: Search plan for query `TurnoutWithErroneousConnections`

Index	Predicate	Bound variables	Search op. type
0	$\texttt{Turnout}(\underline{t})$	$\emptyset$	extend
1	$\texttt{ConnectedTo}(t, \underline{s_1})$	$\{t\}$	extend
2	$\neg\texttt{StraightOrDivergent}(t, s_1)$	$\{t, s_1\}$	check
3	$\texttt{ConnectedTo}(t, \underline{s_2})$	$\{t, s_1\}$	extend
4	$\neg\texttt{StraightOrDivergent}(t, s_2)$	$\{t, s_1, s_2\}$	check
5	$\neg(s_1 = s_2)$	$\{t, s_1, s_2\}$	check

The search plan in Table 3.1 is considered typical as it executes check operations as early as possible. Check operations are relatively simple to perform compared to extend operations, thus early execution helps excluding variable bindings that will not yield a match. Furthermore, except for the first step, the search plan always binds one variable at a time via navigating on an edge, this way it avoids costly Cartesian products when finding candidate variable bindings.

The pseudocode of a recursive query evaluation algorithm which interprets a search plan given as input to find all matches over a (centralized) model is shown in Algorithm 3.2.1. The recursive ExecuteQuery function takes the query q, the index *idx* of the current operation in the search plan, and a *partial binding* Z_p (i.e., a binding that may not bind all free variables of the query) as parameters. In line 2, the executor looks up the search plan for the given query from a global storage. Then, in line 3, the algorithm checks if *idx* points to the end of the operation list. If this is the case, a match has been found and should be returned. Otherwise, the matching procedure continues by initializing an empty match set (line 4) and extracting the predicate enforced in the current search step and storing this predicate to `PRED` (line 5). Based on the variable bindings at the current stage of the query evaluation, the algorithm categorizes the current operation as either an extend or a check in line 6. In case of an extend operation (lines 7-13), all potential variable bindings are calculated (lines 7-8) and the predicate `PRED` is evaluated on them (line 9). For each new partial binding Z_p' obtained this way, the matching process recursively continues with the next search step (lines 10-11). If the current search operation is categorized as a check and the execution continues in line 14, then `PRED` is applied over the values mapped by Z_p partial binding. If this evaluation returns 1, the evaluation proceeds recursively with the next search operation (line 15-16). Finally, in line 18, all matches found in subsequent search steps are returned. To find all matches in the model for query q, ExecuteQuery should be called with parameters (q, 0, Z_p) (lines 20-22).

Algorithm 3.2.1: Query execution algorithm outline

```
 1 Function ExecuteQuery(q, $idx$, $Z_p$) is
 2     $searchPlan \leftarrow$ LookupPlan(q)
 3     if size($searchPlan$) = $idx$ then return $\{Z_p\}$ ;
 4     $matches \leftarrow \emptyset$
 5     PRED $\leftarrow$ predicate evaluated by $searchPlan[idx]$
 6     if $searchPlan[idx]$ is extend then
 7         for $e$ in {all candidates in $\mathcal{D}_M$} do
 8             $Z'_p \leftarrow Z_p \cup \{v_F \mapsto e\}$
 9             if $[\![\text{PRED}]\!]^M_{Z'_p} = 1$ then
10                 $next \leftarrow idx + 1$
11                 $matches \leftarrow matches \cup$ ExecuteQuery(q,$next$,$Z'_p$)
12             end
13         end
14     else if $[\![\text{PRED}]\!]^M_{Z_p} = 1$ then
15         $next \leftarrow idx + 1$
16         $matches \leftarrow matches \cup$ ExecuteQuery(q,$next$,$Z_p$)
17     end
18     return $matches$
19 end
20 Procedure FindAllMatches is
21     $allMatches \leftarrow$ ExecuteQuery(q, 0, $\emptyset$)
22 end
```

3.3 Model Generation Problems

The challenge of model generation lies in the very large number of the possible relationships between various objects in a model. Metamodels capture classes and relationships that can be present in an instance model, while well-formedness rules pose additional constraints on how different relationships can be added between objects, or what attribute values objects may have. The complexity of metamodels and WF constraints necessitates the need of using advanced algorithms for generating models.

Automated synthesis of domain-specific graph models has been actively researched in the field of model-based software engineering [SNV18; SVV16; Bro+06; FSB04]. Hereby, we revisit some core concepts.

A model generation task takes the following four required inputs:

- A *metamodel* Σ with $\{\mathtt{C}_1, \dots, \mathtt{C}_m, \mathtt{A}_1, \dots, \mathtt{A}_n, \mathtt{R}_1, \dots, \mathtt{R}_o\} \subseteq \Sigma$ class, attribute, and reference predicate symbols.

- A *theory* of constraints $\mathcal{T} = \{\varphi_1, \dots, \varphi_n\}$ expressed as FOL *(error) predicates.*
- *Type scopes* $\mathcal{S} \colon \{C_1, \dots, C_k\} \to \mathbb{IV}_{\mathbb{N}}$, where $\mathbb{IV}_{\mathbb{N}}$ is the set of natural number intervals. Type scopes specify the minimum and the maximum number of instances of objects by type, i.e., if $\mathcal{S}(C_i) = [L_i, U_i]$, then solution models must contain at least L_i and at most U_i instances of the class $C_i \in \Sigma$.
- An *objective function* which is a linear function that assigns a real number to a model based on the number of matches for selected predicates and assigned weights. Formally, a linear objective function is $f(M) = \sum_{i=1}^{n} |\mathit{Matches}(M, \psi_i)| \cdot w_i$, where ψ_i are predicates and $w_i \in \mathbb{Z}$ are weights.

Definition 10 (Theory and scope satisfaction by a model)**.** An instance model M *satisfies* theory $\mathcal{T}$ and the type scopes $\mathcal{S}$, written as $\mathcal{T}, \mathcal{S} \models M$, if

- no constraints are violated, i.e., for all error predicates $\varphi \in \mathcal{T}$: $\text{Matches}(M, \varphi) = \emptyset$, and
- the number of model elements of a specific type satisfy the type scope (written as $\mathcal{S} \models \mathit{stats}_M$); formally, for all class symbols $\mathtt{C}_i \in \Sigma$, $\mathit{stats}_M(\mathtt{C}_i) \in \mathcal{S}(\mathtt{C}_i)$.

Definition 11 (Solution of model generation)**.** The *solution* of the model generation task is a set of models that are instances of the input metamodel, satisfy all constraints, and respect the type scopes:

$$\mathit{solutions}(\Sigma, \mathcal{T}, \mathcal{S}) = \{M \mid M \text{ is an instance of the metamodel } \Sigma \text{ and } \mathcal{T}, \mathcal{S} \models M\}.$$

Definition 12 (Optimal solution of model generation)**.** The *optimal solutions* of the model generation are solution models that maximize the value of the linear objective function.

$$\begin{aligned} &\mathit{optimal}(\Sigma, \mathcal{T}, \mathcal{S}, f) = \\ &\qquad \{M \in \mathit{solutions}(\Sigma, \mathcal{T}, \mathcal{S}) \mid \forall M' \in \mathit{solutions}(\Sigma, \mathcal{T}, \mathcal{S}) : f(M') \le f(M)\}. \end{aligned}$$

Our work in Chapter 7 relies on the model generator presented in [SNV18] which was proved to be complete and sound in [Var+18]. Informally, it is able to derive all instance models in a domain (up to a designated size defined by the scopes) which satisfy the constraints.

Example 7. Figure 3.3 presents three different instance models from the MoDeS3 domain (see the metamodel in Figure 3.2a) to illustrate the model generation problem. Well-formedness constraints for the `ConnectedTo` references are defined and added to $\mathcal{T}$ below: a connection between two segments needs to be symmetrical (captured by the formula $\varphi_{symm-conn}$) and a segment cannot be connected to itself (captured by $\varphi_{refl-conn}$). Furthermore, a model scope $\mathcal{S}$ is provided to limit the maximum number of objects of a given type (up to three `Segments`, two `Trains`, and one `Turnout`). Finally, an objective function f is defined that is maximized by the optimal solutions. This function counts the number of objects in the model (all types have the same weight assigned), and counts the pairs of trains which are adjacent to each other, i.e., are located on segments which are directly connected to each other.

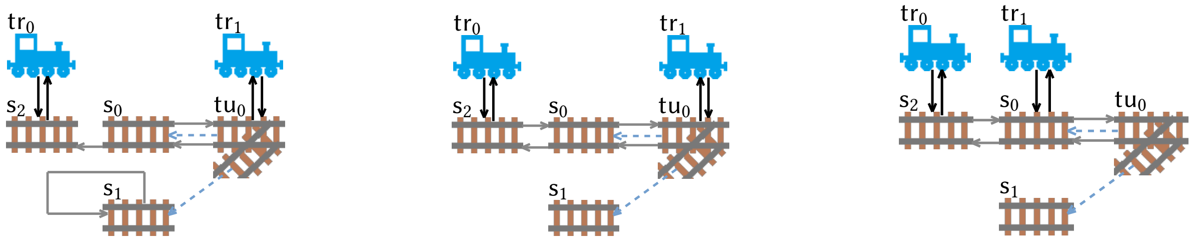
$$\varphi_{symm-conn}(s_1, s_2) = \texttt{ConnectedTo}(s_1, s_2) \land \neg\texttt{ConnectedTo}(s_2, s_1) \qquad \in \mathcal{T},$$
$$\varphi_{refl-conn}(s) = \texttt{ConnectedTo}(s, s) \qquad \in \mathcal{T}$$
$$\mathcal{S}(\texttt{Segment}) = [0, 3], \qquad \mathcal{S}(\texttt{Turnout}) = [0, 1], \qquad \mathcal{S}(\texttt{Train}) = [0, 2]$$
$$f(M) = stats_M(\texttt{Segment}) + stats_M(\texttt{Turnout}) + stats_M(\texttt{Train}) + \text{number of adjacent } \texttt{Train} \text{ pairs}$$

First of all, all three models presented in Figure 3.3 respect the model scope $\mathcal{S}$, i.e., there are not more instances of the types (`Segment`, `Turnout`, `Train`) than the predefined upper bounds (3,1,2, respectively) and there are no lower bounds set by $\mathcal{S}$.

The first model M_a in Figure 3.3a is not a well-formed instance model, because both $\varphi_{symm-conn}$ and $\varphi_{refl-conn}$ have matches, thus the well-formedness requirements are violated: a connection between s_0 and s_2 exists but not the other way around; and s_1 is connected to itself. Due to these these well-formedness constraint violations, M_a is not considered to be a solution to the model generation problem.

Model M_b, on the other hand, is a well-formed model part of the solution set, i.e., this model is derived by the model generator. The objective function $f(M_b) = 6$ is, however, not maximized by M_b because the trains in the model are not on adjacent segments.

Finally, M_c represents an optimal solution. It both respects the model scope, satisfies all well-formedness constraints, and maximizes $f(M_c) = 8$, because it has two pairs



(a) Objective function value of $f(M_a) = 3 + 1 + 2 + 0 = 6$

(b) Objective function value of $f(M_b) = 3 + 1 + 2 + 0 = 6$

(c) Objective function value of $f(M_c) = 3 + 1 + 2 + 2 = 8$

Figure 3.3: Three instance models from the MoDeS3 domain

of adjacent trains ($\langle tr_0 tr_1 \rangle$ and $\langle tr_1 tr_0 \rangle$) in addition to having the maximum number of model objects allowed by the scope.

3.4 The Data Distribution Service Middleware

The OMG specification for Data Distribution Service (DDS) [Par03] provides a common application-level interface for *data-centric* implementations over a *publish-subscribe* communication model. Additionally, the specification defines the main features suitable for application in embedded self-adaptive systems. Since we rely on DDS and its services for building trustworthy distributed applications in Chapter 8 and 9, we provide an overview based on the DDS Architecture Overview by Pardo [Par03].

In data-centric systems, every data object is uniquely identified in a virtual *global data space* (shortly, GDS), regardless of its physical location. For this reason, both the applications and the communication middleware need to provide support for unique identifiers of data objects. Furthermore, this identification enables the middleware to keep only the most recent version of data upon updates, thus respecting the performance and fault-tolerance requirements of real-time applications (that make a centralized solution impractical). By keeping the most recent data, the middleware can provide up-to-date information to new participants of the network.

A simplified UML class diagram containing the concepts of DDS that implement the publish-subscribe communication model is depicted in Figure 3.4. Participant is the top-level entity in a DDS application, so we assume that each deployed program has exactly one instance of it, and we refer to communicating programs as *participants*. Participants may have an arbitrary number of Subscribers and Publishers that handle the actual reading and writing of data, respectively. DataReaders and DataWriters are contained within Subscribers and Publishers. The sole role of DataWriters is to inform their corresponding Publishers that the state of the data object is changed, i.e., calling DataWriter::write() will not necessarily cause immediate communication. Similarly, the task of a Subscriber is to decide when to invoke DataReader::take() that reads the new data values.

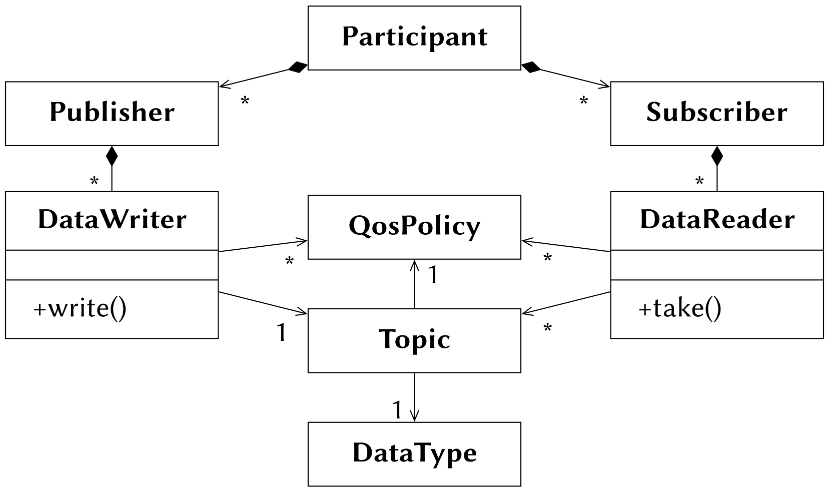


Figure 3.4: UML class diagram of DDS classes

Unlike classic publish-subscribe protocols[2,3], a Topic is more than a routing label for messages in DDS: a Topic is always associated with exactly one predefined DataType. For each DataType, a set of attributes are configured to serve as a *key*, thus the topic and the key together are used for identifying data objects in the global data space. Additionally, this coupling between Topic and DataType (with additional QosPolicy settings) enables optimizations such as pre-allocating the resources needed to send or receive messages of a Topic.

Real-time DDS by [KKS12] is an extension of the DDS standard, which tailors DDS to fit the need of real-time application scenarios. Among other novelties, the work also shows how quality of service (QoS) and quality of data (QoD) specifications can be used to ensure reliable and timely messaging, even over unstable or slow networks. Additionally, DDS is also capable of detecting and reporting violations of QoS contracts to participants. Thus we may assume reliable and timely delivery of messages by the underlying middleware in the current work.

[2]`https://mqtt.org/`
[3]`https://www.amqp.org/`

3.5 Summary

This section provided foundations for the rest of the thesis by defining core concepts and introducing basic techniques and algorithms in a precise way. This includes formal definitions of metamodels, instance models, graph queries, and the model generation problem. Moreover, we described runtime graph models and the local search-based query evaluation technique. Finally, we provide an overview of the Data Distribution Service standard that provides a decentralized publish-subscribe protocol for distributed systems.

Chapter 4
Related Work

This section gives an overview on the current state-of-the-art in four general topics: (1) runtime models, (2) distributed dynamic graphs, (3) runtime monitoring and (4) worst-case execution time analysis. The topics included here cover the fields relevant to the contributions presented in the subsequent chapters of this thesis.

4.1 Runtime Models

4.1.1 Concept of Models@Runtime

A holistic vision about models@runtime was presented by Blair et al. [BBF09]. They see runtime models as means to enable online execution of important tasks that currently require safety-critical systems to be offline, thus yielding significant benefits. Examples of such tasks are the self-* properties, such as reconfiguration to optimize performance (self-adaptation) and recovery from errors (self-healing). Our runtime graph model presented in this thesis is an implementations of the models@runtime concept.

The concept of *Liquid Models* [MW16] refers to the feedback loop from models@runtime to the design time model artefacts. This paper identifies four major research challenges in connection with the gap between design time and runtime concepts, namely (1) integration of distributed and heterogeneous data streams, (2) providing a reactive model stream processing mechanisms, (3) realizing scalable model mining techniques on top of model streams and (4) propagating back and capturing the observed knowledge from operation to design models. The first three challenges are closely related to the management of runtime models, while the last one is focusing on the feedback for design time. Their key idea is that sensor data streams available at runtime should be combined with advanced algorithms to refine models (not only graph models) created at design time. The authors see DDS [Par03] as a key enabler technology that allows timely and reliable data delivery, which we have successfully employed in our distributed monitoring framework.

The MegaM@Rt2 ECSEL Project [Afz+17] is a proposal from 2017 to solve recent challenges in real-world MDE scenarios with a special focus on CPS: quality assurance for development, integration, and maintenance, as well as ensuring traceability between design time and runtime. MegaM@Rt2 aims to deliver a framework of tools and methods for (1) systems engineering and continuous development, (2) related runtime analysis, and (3) global model and traceability management, respectively. A follow-up of the proposal [Cru+20] provides a summary of the diverse open-source tooling that was built during the project. Nevertheless, the developed tools in the project are aiming to assist runtime verification of CPS focusing on temporal behavior rather than employing data-driven monitors.

4.1.2 Frameworks for Models@Runtime

The models@runtime paradigm [BBF09; SZ13] serves as the conceptual basis for the Kevoree framework (KMF) [Mor+14]. This framework originally aimed at providing an implementation and adaptation of the de facto EMF standard for runtime models [Fou+12]. KMF allows sharing objects between different *nodes*, as opposed to our current work where the model elements can only be modified by their host participant, thanks to the single source of truth principle. Additionally, several assumptions applied to KMF heavily depends on the Java programming language and the Eclipse modeling framework, which questions its applicability to resource-constrained environments.

In their more recent work, the authors of KMF introduce GreyCat [Har+19], an implementation for *temporal graphs*. By adding timestamps to graph nodes, it allows identifying a node along its individual timeline. The tool can be used on top of arbitrary storage technologies, such as in-memory or NoSQL databases. As opposed to our approach, they use a per-node locking approach to prevent inconsistencies in the graph.

Other distributed, data-driven solutions include the Global Data Plane (GDP) [Zha+15]. This work suggests a data-centric approach for model-based IoT systems engineering with a special focus on cloud-based architectures, providing flexibility and access control in terms of platform components and data produced by sensors. However, data in this case is represented by time series logs, which is considered as low-level representation compared to abstract graph models employed by our approach.

Adaptive exchange of distributed partial models was studied by Götz et al. [Got+15]. The authors propose a role-based model synchronization approach for efficient knowledge sharing. First, they identify three strategies for model synchronization. Then, with the help of different roles, they show optimizations for knowledge sharing in terms of performance, energy consumption, memory requirements, and data privacy. Furthermore, the authors only provide implementation recommendations, but the work lacks practical evaluation of the idea. In contrast, data ownership is exclusive and is determined by the platform in our approach. Furthermore, we provide the computation of global, system-wide queries based on the information partitioned data across the platform.

The framework CHROMOSOME (or shortly, XME) [Buc+14] offers tools to compose a modular architecture of a system that enables adding and removing applications and adapting to changes in the hardware topology while still respecting requirements typical in embedded systems for each component (e.g., timing, safety). They consider two types of adaptivity, *system dynamics* and *reflection*. The former refers to the ability of adapting to appearing and disappearing platform participants, which corresponds to fault tolerance. Reflection, on the other hand, means that the system can interpret the services it provides and is able to adapt its functionality to provide optimal performance, such as routing network traffic to a different, less busy channel. XME shares several concepts with other projects, such as providing a general platform abstraction layer in a similar way as it is in ROS [Qui+09], allowing *plug & play* support in RACE [Som+13], or providing (a subset of) QoS guarantees included in DDS [Par03]. In essence, the focus of XME lies in modeling the platform and the running services, which is more specific than our conceptual runtime model which is domain independent.

4.2 Distributed Management of Dynamic Graphs

4.2.1 Management of Graphs in Distributed Systems

In the book on Graph Data Management [SL18], Shao and Li provide an overview of the challenges of storing, processing, and querying (i.e., finding patterns in) large graphs having 100M+ elements. They also compare the capabilities of 15 representative graph processing systems, and dissect the challenges that stem from the distributed nature of the underlying storage. There are some quantitative comparisons of run times for a few scenarios of such systems, however, these are based off mathematical calculations considering their underlying computation complexity. Finally, they discuss some approximation algorithms and alternative graph representation forms, namely matroids and graph embedding, that can potentially speed up otherwise complex operations, but no specific details are provided.

LEOPARD [HA16] is a dynamic, edge-oriented graph partitioning algorithm for distributed graph storages. The main goal is to achieve a graph partitioning in which the ratio of cut edges is low, i.e., both ends of edges are in the same partition. They approach the partitioning problem by saying that dynamic graphs are very much similar to solving the *one-pass partitioning problem*, as the changes to the graph can be regarded as a stream. LEOPARD can dynamically select vertices and reassign them to new partition time to time in order to keep the cut ratio low. Furthermore, vertex replication follows the *Minimum-Average Replication Algorithm* (also presented in the paper) that tells how many copies and where those copies should be created.

In general, a fundamental difference between the runtime graph management protocol proposed in this thesis and the available data partitioning algorithms in large graphs is the decision about allocation of a given graph element. While in database systems, data can be allocated at a selected location and it can be moved to optimize performance, our assumption in distributed CPSs is that data is available at the location where it is created (e.g., from a sensor reading).

4.2.2 Graph Pattern Matching

Gallagher [Gal06] thoroughly surveyed the practical approaches for collecting matches of graph patterns. This work, however, only focuses on the foundational algorithms for finding subgraphs that satisfy all conditions prescribed by the patterns, and does not discuss distributed versions of the algorithms.

Distributed graph query evaluation over fragmented data was first presented by Ma et al. [Ma+12] while further algorithms were subsequently reported by others as well [Mit+14; Pet+14; KTG14]. The IncQuery-D framework [Szá+14] provides support for distributed incremental model queries deployed over a cloud infrastructure. It builds an in-memory middleware layer on top of a distributed model storage system, and uses the Rete algorithm [For82] for incremental maintenance of query results. As a key limitation, these distributed graph query evaluation techniques use a cloud-based execution environment, thus they are not directly applicable for a heterogeneous execution IoT platform with low-memory computation units where there is no network and device capacity to forward all data to a central location in a cloud. Furthermore, cloud-based storage and processing of data can be problematic from a privacy perspective in some cases and thus local processing is desired.

4.2.3 Distributed Graph Databases

There are existing databases that use graphs as the underlying data representation. One of such databases is JanusGraph (formerly known as TITAN)[1]. It provides support for storing and querying very large graphs by running over a cluster of computers. In addition to storing data in a distributed way within a cluster, it also supports fault tolerance by replication and multiple simultaneous query executions by transactions. Even though it claims to execute complex graph traversals in real time, the framework provides no QoS assurance regarding response time.

OrientDB[2] is a multimodel database that has a native graph database engine where graph data may or may not be defined by a corresponding schema. However, in case of both JanusGraph and OrientDB, deployment of the database to memory-constrained devices is not supported by default, which is a fundamental need for distributed CPSs.

Furthermore, a distributed graph database specifically for the data management of IoT systems is presented by Ueta et al. [Uet+16]. They rely on the *property graph data model* [CDH00], where nodes of a graph can represent users, devices, data directories, data items, and actual data, while edges represent data access and data ownership. A prototype implementation provides CRUD operations for the data. This implementation incorporates lightweight REST servers running on each platform unit to allow remote execution of operations.

4.3 Runtime Verification

4.3.1 Runtime Verification Approaches

For continuously evolving and dynamic CPSs, an upfront design time formal analysis needs to incorporate and check the robustness of component behavior in a wide range of contexts and families of configurations, which is a very complex challenge. Thus consistent system behavior is frequently ensured by runtime verification (RV) [LS09], which checks (potentially

[1]https://janusgraph.org/
[2]https://orientdb.org/

incomplete) execution traces against formal specifications by synthesizing verified runtime monitors from provenly correct design models [MP14; JTF17]. These approaches focus on the temporal behaviour of the system: runtime verification of data-driven behaviour is not their main goal.

Recent advances in RV (such as MOP [Mer+12] or LogFire [Hav15]) promote to capture specifications by rich logic over quantified and parameterized events (e.g., quantified event automata [Bar+12] and their extensions [DLT15]). Moreover, Havelund proposed to check such specifications on-the-fly by exploiting rule-based systems based on the RETE algorithm [Hav15]. However, these techniques consider relations between events and do not take models as first class citizens of the runtime analysis.

Traditional RV approaches use variants of temporal logics to capture the requirements [BLS11]. Recently, novel combinations of temporal logics with context-aware behaviour description [Gön+16; HMM13] (developed within the R3-COP and R5-COP FP7 projects) for the runtime verification of autonomous CPS appeared and provide a rich language to define correctness properties of evolving systems. These approaches introduced the concept of context models, which can also be represented in the graph based approach of this thesis. Recently, monitoring approaches to analyse topological properties in a discrete space appeared [Nen+15]. Qualitative and quantitative analysis is supported. However, complex data driven behaviour is not the focus of the approach.

4.3.2 Runtime Monitoring in Resource-Constrained Environments.

The tool polyLarva [Col+12] provides means to adjust the possible overhead imposed by the runtime checks performed during monitoring. The Brace framework [Zhe+16] supports monitoring in distributed and resource-constrained environments by incorporating dedicated units in the system to support global evaluation of monitoring goals.

4.3.3 Runtime Verification of Distributed Systems.

While there are several existing techniques for runtime verification of sequential programs, Mostafa and Bonakdarpour [MB15] claim that much less research was done in the area for distributed systems. Furthermore, they provide the first sound and complete algorithm for runtime monitoring of distributed systems based on the 3-valued semantics of LTL.

Bauer and Falcone [BF16] focus on evaluating LTL formulae in a fully distributed manner for components communicating on a synchronous bus in a real-time system. These results can provide a natural extension of our work into the temporal directions. Additionally, a machine learning-based solution for a scalable fault detection and diagnosis system is presented by Alippi et al. [ANR17] that builds on correlation between observable system properties.

4.3.4 Resource Monitoring Frameworks

A cloud-centric vision for Internet of Things is presented by Gubbi et al. [Gub+13], which is a dominant approach in healthcare IoT applications. Typically, data collection from the environment is carried out at the edge of the network, then intelligent processing is performed on a cloud platform [Has+15; Fan+14]. However, our focus differs from this in two ways, namely (i) all the processing is done on the network edge due to assumed network and device performance limitations and to ensure that sensitive data is not exposed and (ii) they rely on low-level sensor data directly for monitoring and analytics rather than building an abstract knowledge base.

The work presented in [Sha+10] uses runtime models to monitor cloud platforms. It defines a metamodel to describe SaaS cloud services, and model their allocation to physical hosts and runtime performance characteristics. Instances of this model are created at run time based on the incoming information obtained from the various services. Data collection is done by agents monitoring the services, and data aggregation is in a centralized database for monitoring data.

4.4 Worst-Case Execution Time Analysis

Methods for efficiently analyzing timing properties and computing precise WCET bounds of programs have been actively researched since real-time systems appeared. Detailed surveys on such methods are available, for example, by Kozyrev [Koz16] and Wilhelm et al. [Wil+08]. Static WCET analysis techniques have two major categories [Wil+08].

- *High-level analysis* works with the abstract flow of a program, mainly using the control flow graph or the control flow automata obtained from the source or machine code [LS03; Fer+08; Bla+10].
- *Low-level analysis* techniques focus on platform-specific details (e.g., memory, caches, pipelines, and branch prediction) when assessing timing properties [BP05; SS07; Pua06].

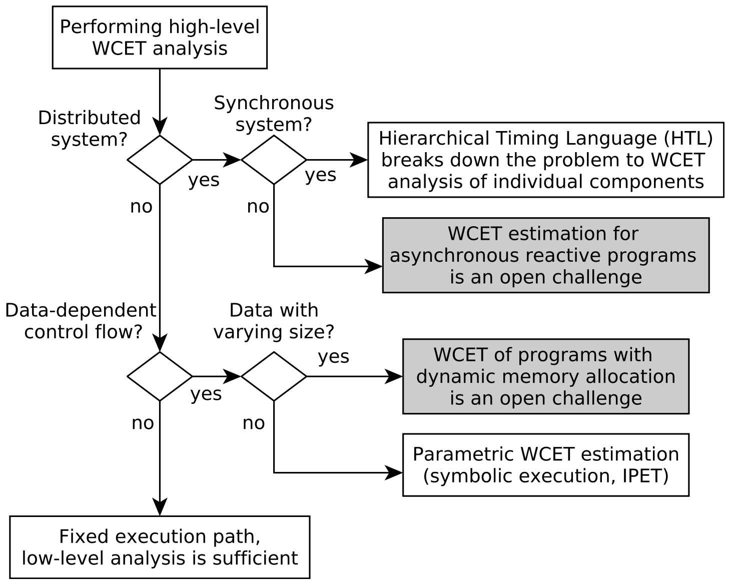

Performing high-level WCET analysis
Distributed system?
Synchronous system?
yes
yes
no
no
Hierarchical Timing Language (HTL) breaks down the problem to WCET analysis of individual components
WCET estimation for asynchronous reactive programs is an open challenge
Data-dependent control flow?
Data with varying size?
yes
yes
no
no
WCET of programs with dynamic memory allocation is an open challenge
Parametric WCET estimation (symbolic execution, IPET)
Fixed execution path, low-level analysis is sufficient

Figure 4.1: Summary of high-level static analysis techniques for computing WCET

Various high-level, static WCET analysis techniques have been developed to provide safe timing estimates for the execution of various types of programs. Figure 4.1 summarizes the most popular methods used for WCET analysis. The implicit path enumeration technique (IPET) [LM95] is commonly used for this purpose. This technique analyzes the program paths (control flow) to determine what sequence of instructions will execute in the extreme case. The path analysis in IPET is based on solving and integer linear programming problem.

Moreover, initial WCET analysis support has been provided for distributed systems by Ghosal et al. [Gho+06] by breaking down the problem into WCET analysis of communicating components using the *Hierarchical Timing Language* (HTL). HTL organizes tasks into a hierarchical, tree-like structure that has the complete program as its root. All tasks are executed periodically, and dedicated communicator processes coordinate the exchange of information between the ones running on different hosts.

4.4.1 Program Flow Analysis

Timing analyzers for program flow analysis often employ some version of the implicit path enumeration technique (IPET) [LM95]. The general idea behind this method is to use the control flow graph (CFG) of the program to create an integer linear program (ILP) where each variable encodes the number of executions of a corresponding basic program blocks, and the objective function is to maximize their total execution time. Besides the IPET method, several *tree-based* methods exist which use a tree representation of the program (obtained from the source code or compiled binary) and apply some traversal to find the longest path in a program [Lim+95; CB02; BFL17]. In any case, the effectiveness of these methods rely on precise program *flow facts* (e.g., loop bounds, infeasible paths) to be able to determine a safe and tight WCET estimate. Although there are several advanced (semi-)automated techniques available today to derive additional constraints on the program flow thus improving the precision of the WCET estimate [Gus+06; Erm+07; CJ11; KKZ13; Lis14], there is still a significant manual effort needed to specify flow facts [Abe+15]. Most closely related to our current work is [KKZ13], which uses uses abstraction refinement to reduce WCET estimates by *squeezing*. This squeezing technique operates by checking if the WCET estimate is produced by a feasible program path. If the path is feasible, a WCET estimate is found, but if it is not, then a new constraint is added to the WCET estimation problem and a new WCET is computed. This process can be repeated until either a WCET path is found to be feasible or the WCET estimate falls below a desired threshold. However, this approach cannot exclude longest execution paths from the program which are infeasible due to *complex domain-specific constraints on the inputs*.

4.4.2 Parametric WCET Computation

An efficient algorithm for parametric WCET calculation is presented by Bygde et al. [BEL11]. Constant WCET estimates may not be useful, since WCET may heavily depend on many parameters and configurations that are only known at runtime. Thus, a classic WCET analysis result yielding a safe upper bound is typically impractical. However, a parametric WCET derives an upper bound as a formula, rather than a constant for a program, which can then be instantiated at runtime with the corresponding variable values.

4.4.3 Real-Time Database Queries

In real-time databases [OS95], access to data has strict time constraints. The work in [HOT89] presents a data sampling-based statistical method to evaluate aggregate queries in a database. There is a trade-off between time available for query execution and the precision of the estimate. Such estimations would not be acceptable in a monitoring setting where precise query results are expected. The real-time object-oriented database RODAIN [TR96], which targets telecommunication applications, does not support *hard real-time* transaction (i.e., query) types, because it is considered too costly for the target domain. However, our objective is exactly to provide such guarantees over graph models to support hard real-time applications.

4.4.4 WCET of Graph-Based Computations

Graph models and queries have been often used in design models and tools of real-time systems [Jür03; Gie+03; Bur+04]. However, in the context of data-driven monitors, these techniques are applied at runtime for monitoring deployed in the real-time system itself, for which only a few related papers exist. One of the few related works that investigates real-time properties of graph-based techniques is by Burmester et al. [Bur+05]. Motivated by the expressiveness of *story diagrams* [Fis+98], the authors evaluate the applicability of this high-level modeling formalism to recognize hazardous situations in real-time systems. Their work investigates worst-case execution times of imperative programs generated from such story diagrams by executing measurements of manually created worst-case inputs. In contrast, our work aims to automatically synthesize worst-case well-formed input models as part of static analysis.

4.4.5 Hard Real-Time Monitors in Embedded Systems

One of the earlier works in the field is The Temporal Rover [Dru00]. This framework can generate monitoring code from temporal logic formulae with low overhead, but the verification of properties is done in a large part on a powerful remote host, while our method does not rely on any external component. The concept of *predictable monitoring* was introduced in [ZDG09] where static scheduling techniques were used to show that a monitor fits its allocated time frame, but the analysis of monitoring tasks is out of its scope which is the topic of this thesis. Finally, synchronous component execution and observable program states are the main assumptions made in [Pik+10] to support sampling-based monitoring of input streams in real-time systems, whereas our work targets monitors executing complex queries over a graph model capturing contextual information on a high-level of abstraction.

4.4.6 Dynamic Memory Allocation in Embedded Systems

A well-known challenge in programs with dynamic memory needs is the ability to precisely predict the behavior of the memory allocator [Wil+10]. In general, allocators do not provide guarantees about the memory addresses reserved at runtime. This makes low-level WCET analysis problematic because there is no information about what cache sets will the newly allocated memory belong to. This way every time a dynamically allocated memory is accessed, the analyzer needs to assume that the all contents of the cache is invalidated. A further issue with using dynamic memory allocation is that the allocator itself is using some internal data structure for tracking in-use memory blocks. This way, whenever an allocation is initiated, the access to these internal data structures pollutes the cache.

A solution in [Wil+10] to the nondeterminism of memory allocators is to use deterministic ones instead [Mas+04; HRW08]. Such deterministic allocators are able to provide guarantees regarding the placement of the allocated memory blocks and they serve allocations in $O(1)$ time, but the memory tends to be more fragmented compared to traditional allocators, which may result in poor memory utilization.

Another approach to circumvent the limitations of dynamic allocation is to a priori compute the memory usage of the application [HR09]. The idea is to allocate memory in advance that the program will need at runtime. This method optimizes the reserved memory size by reusing some data structures multiple times for different purposes at runtime. In this case, however, detailed information is required about the memory needs of the program, which is not always available.

4.5 Summary

This chapter introduced the related work for this thesis and covered four major fields: runtime models, distributed graphs, runtime verification, and worst-case execution time calculation. We now summarize the limitations in the state-of-the-art and outline the unique features of the approaches presented in this thesis.

Runtime models. While the use of graphs as the underlying knowledge representation is not new, the adaptation of runtime graph models to real-time, resource constrained environments has been neglected. One of the main objectives of this thesis is to provide such models usable with the platform of modern CPSs.

Dynamic graphs. Most systems keep graphs in large databases and partition data so that it can be efficiently stored, queried, or updated. However, the objective of this thesis is addressing a different problem which has not been investigated: how to capture graph data close to the source of the data, while providing various guarantees (fault tolerance or consistency).

Runtime verification. Novel rule-based runtime monitoring solutions have started getting more attention in the last 5 years. This thesis presents one of such innovative approaches, and we show how to use graph queries for formulating the monitoring objectives. Monitoring programs synthesized from these high-level declarative descriptions focus on complex relationships in the data available about the underlying system at a given time, rather than detecting sequences of events.

Worst-Case Execution Time Analysis. Modern smart and safe CPS applications are predicated on different ideas than traditional safety critical systems, and they often use intelligent data processing algorithms while carrying out critical tasks. Although there is a significant body of work available in the literature about how to statically analyze programs to derive timing properties, the results presented in this thesis focus on automatically incorporating domain-specific knowledge about the program inputs into the analysis process to provide tighter estimates, which is unprecedented in the field of WCET analysis.

Part II

Runtime Graph Models and Queries in Real-Time Systems

CHAPTER 5

Adaptation of Runtime Graph Models to Embedded Systems

In the last decade, with the growing interest towards self-adaptive systems, new approaches were developed which enable the management and maintenance of graph models at runtime [GHT09; Mor+14; Har+19]. Capturing the contextual information of a cyber-physical system in an evolving graph model opens the way for the adaptation of existing graph-based techniques to runtime. However, there has been very limited discussion about runtime model representations suitable for real-time embedded environments.

This chapter lists high-level requirements of using such runtime graph models in hard real-time systems as the underlying knowledge base for runtime monitors. Concrete suggestions for the capabilities and features of a runtime graph model manipulation API for embedded systems are provided in Section 5.1. Furthermore, we evaluate our prototype implementation of the proposed model API by comparing the execution characteristics of the same initial implementation across different hardware platforms in Section 5.2.

5.1 Graph Data Structures for Embedded Systems

A snapshots of the underlying runtime model of the system can be represented by an instance model, as discussed earlier in Section 3.1.1, and the structure of this model directly impacts the performance of query evaluation. Since an embedded device may have limited available CPU and memory resources, a lightweight data structure is desirable to efficiently capture runtime graph models. While the in-depth discussion and trade-off analysis of possible graph

```c
 1 typedef struct {
 2   uint16_t segment_id;
 3   Train *train;
 4   Segment *connected_to[2];
 5   uint8_t connected_to_count;
 6 } Segment;
 7 
 8 typedef struct {
 9   uint16_t train_id;
10   double speed;
11   Segment *location;
12 } Train;
```

Listing 5.1.1: Classes of the MoDeS3 domain

```c
typedef union {
  Segment segment;
  Train train;
} Object;
struct Modes3ModelRoot {
  Object objects[MAX_OBJECT_COUNT];
  uint16_t object_count;
  uint16_t segments[MAX_OBJECT_COUNT];
  uint16_t segment_count;
  uint16_t trains[MAX_OBJECT_COUNT];
  uint16_t train_count;
} runtime_model;
```

Listing 5.1.2: Generic graph object and model root

Listing 5.1: Example implementation of a generic graph data structure with Segment and Turnout domain classes

data structures is beyond the scope of this work, we present one set of requirements and assumptions about the supported operations of the underlying model.

A possible implementation of the metamodel classes of the MoDeS3 overarching running example is introduced below.

> **Example 8.** Listing 5.1.1 shows a possible C implementation of data structures for Segment and Train classes present in the metamodel previously introduced in Figure 3.2a. Lines 2, 9, and 10 are fields created from respective attributes. Furthermore, in this example, we implement references (i.e., links in the graph model) as pointers (line 3 and 11) or pointer arrays with sizes (lines 4 and 5).

5.1.1 Dynamic Element Allocation

The runtime model serves as the knowledge base about the underlying system and its environment. For this reason, it needs to accommodate graph models without a theoretical a priori upper bound for model size. A solution to this is to dynamically reserve and free memory at runtime. However, a major drawback of such an approach is the nondeterminism that lies in the typical implementation of such allocator functions (e.g., `malloc` and `free` in C libraries), as they often do not guarantee neither return value (i.e., memory allocation can be unsuccessful) nor execution time (i.e., allocation time depends on factors hidden from the user of the function). By employing such functions, the embedded software inherently becomes more complex and hinders the assurance of reliable operation, and makes it especially difficult to estimate the WCET. Deterministic allocators [HRW08] can mitigate this issue, but there is a very limited number of platforms which have such deterministic implementations available.

Based on [HR09], we suggest an alternative to avoid dynamic memory allocation in real-time systems by allocating the maximum amount of memory that is physically possible as runtime graph model storage. A major advantage of this is that only the allocated memory is determined at compile time, the type (and distribution) of graph objects stored is runtime information. However, as a trade-off, the approach can be moderately wasteful with the memory space (as discussed later in Section 5.2) in order to make the program execution deterministic.

> **Example 9.** Listing 5.1.2 shows how the graph model container `Modes3ModelRoot` from Figure 3.2a (lines 5-12) can allocate static memory for generic graph objects represented by the union type `Object` (lines 1-4) in C. The maximum used memory by the graph is preallocated in line 6 by the `objects` array whose length equals to the maximum number of `Object` that fits into the memory of the device (denoted by the constant `MAX_OBJECT_COUNT`). However, the information about what type of graph object (`Train` or `Segment`) is stored at a location in the `objects` array is only determined at runtime, which gives flexibility to the proposed model storage.

5.1.2 Object Indexing

As query evaluation typically starts by iterating over all elements of a given type or accessing specific objects (see Section 3.2), it necessitates efficient object access, e.g., by maintaining a real-time index for memory resident data [CK96].

In this work, we propose a simple indexing mechanism that relies on a mandatory, unique (integer) `id` attribute to be present for each type, which also encodes the type of the object. We use these unique identifiers as means to access objects in the graph individually. For each type $\mathtt{C}_i$, we create a lookup table that uses the identifier as the key, and the return value is the pointer (or offset) to the object in the memory.

> **Example 10.** In Listing 5.1.2, the arrays `segments` (line 9) and `trains` (line 11) implement the lookup tables for the respective types and they keep track of the *indexes* (i.e., pointer offsets) of the objects within the `objects` array. The `id` attributes of a given `Segment` or `Train` model object is used to index these arrays. We also assume an ID

space where the most significant bits encode the type of the object, allowing the implicit selection of the lookup array (`segments` or `trains`) to be indexed.

5.1.3 Continuous Maintenance of Model Statistics

As objects in the graph model are created and deleted, high-level model statistics [Var+15] such as the number of instances of each type $\mathtt{C}_i$ (i.e., $\mathit{stats}_M(\mathtt{C}_i)$) in the model should be maintained continuously to allow real-time access to them. As discussed later in Section 7.1, the graph query execution is heavily data-dependent, and the runtime graph model statistics provide a succinct view of the model which is useful in estimating the execution times of query programs.

Example 11. Runtime model statistics are captured in `Modes3ModelRoot` by the counters in lines 7, 9, and 11. These are simple variables which are incremented/decremented as part of object creation/deletion.

5.1.4 Navigability Along Edges

Many steps in query evaluation navigate along the edges (references) of selected objects to find further appropriate variable substitutions for unbound query variables. A simple way to support this feature is by, e.g., maintaining direct pointers in the objects to reachable objects. Representing links between objects with pointers is highly efficient from a performance viewpoint. However, in this case dangling edges may occur, which need to be accounted for when navigating along the references (e.g., by checking the existence of the target object).

Example 12. The example classes in Listing 5.1.1 encode references in line 11 (reference to a single object) and in lines 4–5 (reference to a set of objects).

5.1.5 Reduced Memory Footprint

For every CPU and instruction set architecture (ISA), there are *alignment rules* regarding the placement of data to memory addresses for efficient handling. In practice, this often requires the insertion of *padding bytes* between members of complex data structures to ensure that the data conforms to the alignment rules of the execution platform. However, in embedded systems with about tens of kilobytes of available memory, it can provide valuable gains in storage capacity if the data in the memory could be organized without such paddings inserted. This tight arrangement of data without any padding bytes is referred to as *packed layout* and it is often employed in serialized network packets [ELK08].

On the x86 architecture, which is common in desktop computers, accessing unaligned data can cause slower data fetching, but it is an operation that is generally supported by the hardware of the CPU. However, in typical resource-constrained environments x86 is not the dominant architecture, several applications use platforms where access to unaligned data is not supported. Such unaligned accesses often lead to a hard fault at runtime, and they can only recover by resetting the system. However, on execution platforms where arbitrary data access is possible and only limited memory is available the trade-off between the size of the saved memory space and the imposed run time overhead needs to be evaluated.

The memory layout can be changed by using compiler switches and nonstandard language constructs in case of the C programming language. For example, such a construct for the GCC compiler is `__attribute__((packed))` which tells the compiler to ignore the default data alignment rules for the type and place data members of a structure to the memory with no padding between them.

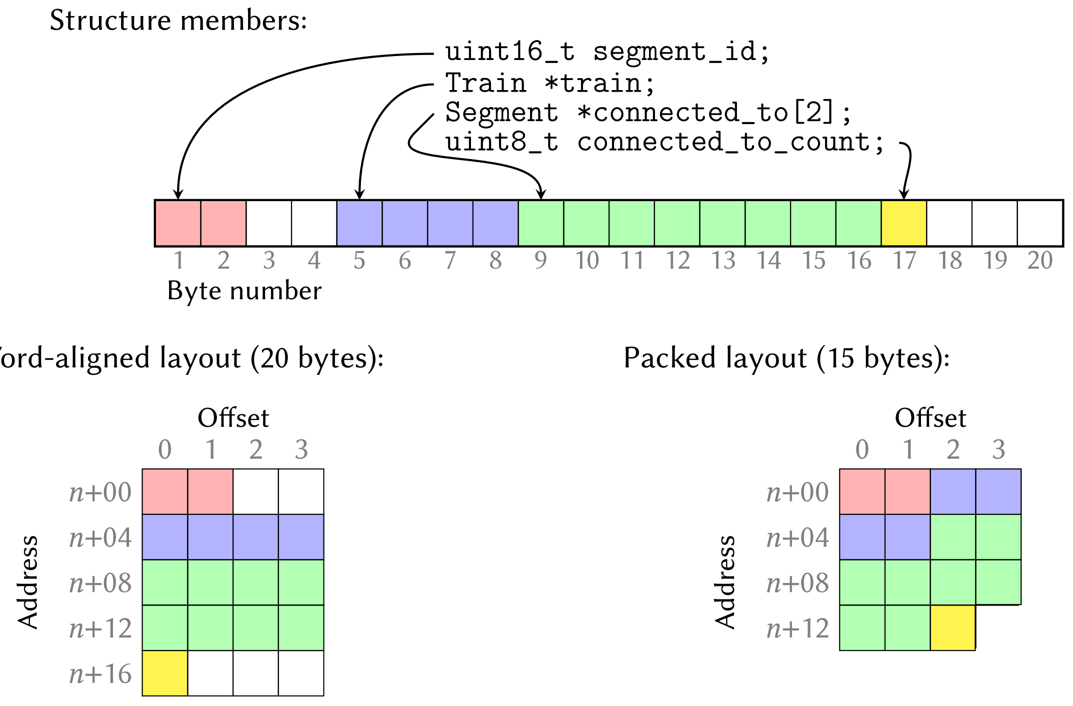


Figure 5.1: Memory layout options for the C structure Segment

Example 13. Figure 5.1 illustrates the memory alignment differences using the C structure `Segment` defined in Listing 5.1.1. The top part and lower left side of the figure demonstrates how the compiler aligns the data by default on a CPU with a word size of four bytes. The `Segment` structure would occupy a total of 20 bytes including padding, while the useful data is only 15 bytes. Furthermore, such a data structure can only be allocated to memory addresses of the form $n = 20 \cdot k, k \in \mathbb{N}$ to ensure consistent alignment.

On the other hand, when `__attribute__((packed))` is added to the definition of the structure, the data alignment will be the one shown in the lower right part of Figure 5.1. In this case, the data structure can be allocated to any memory address as well, i.e., $n \in \mathbb{N}$ addresses are all allowed.

5.2 Evaluation

We created a prototype software that implements the model operations described in Section 3.1.2. We deployed this prototype to different microcontroller units (MCUs) with limited computing and memory capacity to compare the different memory allocation strategies. In particular, we aim to find answers to the following questions:

RQ1 What is the average execution time of the different model update operations when using static vs. dynamic memory allocation?
RQ2 What is the memory usage of static and dynamic allocation strategies?
RQ3 What is the runtime effect of using packed memory alignment?

5.2.1 Evaluation Overview and Setup

The runtime graph model

To address our research questions, we manually designed an instance model inspired by the MoDeS3 CPS demonstrator [*c*6] introduced in Chapter 2. We implemented a program prototype which first creates and then deletes this realistic runtime model repeatedly, and we measure the average execution times of the update (create and write/delete) operations.

The model captures a detailed runtime model snapshot of MoDeS3 that is similar to the one presented in Figure 3.2b and counts a total of 156 objects. The model statistics are as follows:

- $stats_M$(`Segment`) $= 96$,

- $stats_M(\texttt{Turnout}) = 30$, and
- $stats_M(\texttt{Train}) = 30$.

Hardware setup

In this evaluation, the programs writing the runtime graph model execute on MUCs which have no other tasks (e.g., interrupts) running. Because these programs take tens of microseconds to execute, we apply the following measurement setup to obtain an estimate of the average object update execution times:

1. First we create an infinite loop and observed the loop execute n times under a fixed time duration T_{meas} (in our case $T_{meas} = 1s$).
2. Then, we add a selected set of model update operations targeting a single type $\texttt{C}_i$ (e.g., all deletions of Train objects) to the loop and counted m loop executions under T_{meas}.
3. Finally, we use the following formula to get the average run times of query evaluations on given input models (total run time divided by the number of objects of type $\texttt{C}_i$):

$$T_{obj\_upd} = \frac{\left(\frac{1}{m} - \frac{1}{n}\right) \cdot 10^6 \mu s}{stats_M(\texttt{C}_i)}.$$

Repeating the above measurement process 30 times for each set of operations show negligible difference in the average object update operation times (maximum difference from the measured average is $0.05\mu s$), which is unsurprising since it is the only task running in the system.

The programs used are compiled with the GCC compiler for ARM version 7.2.1 20170904 (release) and they are deployed to three different microcontroller evaluation boards:

- Adafruit Feather Express Board[1] has a low-power Microchip microcontroller ATSAMD-21G18A with ARM Cortex-M0+ core without any instruction or data cache. The CPU clock runs at 48MHz.
- Infineon Relax Lite Kit-V1 Board[2]. This board has an XMC4500 F100-K1024 microcontroller and it is driven by a 120MHz system clock. This microcontroller is considered to be a mature industrial MCU and has an ARM Cortex-M4 core with 4KB instruction cache and no data cache.

[1]`https://www.adafruit.com/product/3403`
[2]`https://www.infineon.com/xmc-dev`

- STM32 Nucleo-144 Development Board[3] features the STM32F767ZI high-performance microcontroller with ARM Cortex-M7 core. It has both instruction and data cache, and the system clock is 216MHz by default.

5.2.2 Measurement Results

The evaluation results are presented in Table 5.1. The average time of an operation projected to an object is presented in each row (measured in microseconds). The *Operation* column shows the operation type (creation or deletion), the *Allocation* column shows if the operation relies on upfront (static) or runtime (dynamic) memory allocation. The *Actions* column briefly summarizes what steps are taken during the operation. These possible actions are a combination of (i) allocating memory, (ii) writing the model object, (iii) adding the object to the index structure of the model, (iv) remove the object from the index, (v) scrubbing (i.e., filling with zeroes) the memory location where an object was previously stored, and (vi) freeing memory.

The last six columns in Table 5.1 present the measurement results on three different MCUs with the two different memory placement strategy (*Aligned* or *Packed*) on each. The used ARM MCUs, however, do not support unaligned memory accesses on hardware-level, i.e., such accesses lead to hard faults at runtime. Instead, the compiler generates machine code with respective individual byte accesses and bit shifting operators which are functionally equivalent to aligned data access, but inherently come with overhead in run time. With these devices, performing unaligned memory accesses require some extra caution from the programmer to refrain from certain language constructs, e.g., not to dereference pointers set to unaligned members of a C struct where compilers are unable to recognize unaligned data accesses and would generate code which leads to hard faults during execution.

Findings for RQ1. For *object creation* operations on all platforms, we observe approximately two times slower execution times with dynamic memory allocation when compared to the static one. This is in line with our expectations because the memory allocator function takes additional steps to record the assigned memory addresses which poses a significant overhead.

On the other hand, *object deletion* operations show bigger differences between platforms from the perspective of static and dynamic memory allocation. The time needed to call `free` on a dynamically allocated object vs. fill out the statically allocated memory location with zeroes

[3]`https://www.st.com/en/evaluation-tools/nucleo-f767zi.html`

Table 5.1: Runtime model object manipulation times on real-time units

				Microcontroller					
				ATSAMD21G18A		XMC4500		STM32F767ZI	
Operation	Allocation	Actions	Object Type	Aligned	Packed	Aligned	Packed	Aligned	Packed
Creation	Static	Add to Index and Write	Segment	8.04 μs	+204%	0.64 μs	+27%	0.26 μs	+38%
			Turnout	9.71 μs	+304%	0.86 μs	+23%	0.30 μs	+49%
			Train	7.46 μs	+197%	0.62 μs	+30%	0.26 μs	+56%
	Dynamic	Allocate Memory and Write	Segment	19.22 μs	+85%	1.18 μs	+12%	0.45 μs	+17%
			Turnout	25.06 μs	+117%	1.41 μs	+14%	0.57 μs	+19%
			Train	17.05 μs	+86%	1.07 μs	+16%	0.40 μs	+35%
Deletion	Static	Remove from Index	Segment	2.65 μs	+0%	0.21 μs	+0%	0.11 μs	+2%
			Turnout	2.69 μs	+0%	0.21 μs	+0%	0.11 μs	+5%
			Train	2.70 μs	+0%	0.21 μs	+0%	0.11 μs	+1%
		Remove from Index and Scrub Memory	Segment	11.31 μs	+122%	0.85 μs	+53%	0.41 μs	+64%
			Turnout	14.90 μs	+138%	1.12 μs	+58%	0.56 μs	+67%
			Train	8.41 μs	+148%	0.63 μs	+59%	0.33 μs	+62%
	Dynamic	Free Memory	Segment	11.83 μs	+0%	3.93 μs	+0%	0.21 μs	-2%
			Turnout	11.96 μs	+0%	8.14 μs	+0%	0.22 μs	+2%
			Train	11.72 μs	+0%	9.16 μs	+0%	0.21 μs	+3%

(i.e., scrub) and remove the index entry for the object show three different relationships on the three platforms. Dynamic memory allocation takes 50% less time on the more advanced STM32F767ZI, takes about the same time for ATSAMD21G18A, and takes 4–14× more time on XMC4500. This observation sheds light on the differences between the standard library implementations available for the three different MCUs.

Runtime models can be stored in a platform-independent way with rapid, lightweight, deterministic graph model storage.

Findings for RQ2. To compare the memory utilization of the two approaches, we focus purely on the memory consumed by the model objects themselves. In this comparison, we do not consider the additional space taken by the `malloc`/`free` standard library functions and the space required by the index structures which facilitate the lookup of model elements by their identifiers.

The model used for this evaluation has a total of 156 objects: 96 segments, 30 turnouts, and 30 trains. The runtime sizes of the structures (with only aligned accesses) are 20 bytes, 28 bytes, and 24 bytes, respectively. Furthermore, the `Object` union data structure formed from the three C structures occupies 32 bytes.

The total model size in the dynamic case equals to the sum of the sizes of the individual structures: $96 \cdot 20$ bytes+$30 \cdot 28$ bytes+$30 \cdot 28$ bytes = 3480 bytes. In case of static allocation, each graph object occupies 32 bytes, which yields $156 \cdot 32$ bytes = 4992 bytes. In this assessment, the dynamic memory allocation approach uses 30% less memory than our static allocation scheme.

Our comparison shows that the proposed static model allocation method requires 30% more memory compared to dynamic allocation to store the same graph, which is an acceptable price to pay in exchange for deterministic (and 50% faster) execution times.

Findings for RQ3. The biggest drawback of using unaligned memory accesses is the limited support from the hardware side and the added restrictions of allowed language constructs. Fortunately, in case of this evaluation, the compiler is able to recognize such cases and replace them by multiple instructions with equivalent behavior, thus our monitoring programs are able to successfully complete their executions.

Another negative consequence of packed memory layout is the significant increase in execution time needed for writing data to unaligned memory addresses, which we observed in three cases for all platforms: (i) writing a structure to statically allocated arrays, (ii) writing a structure to dynamically allocated memory space, and (iii) scrubbing the contents of a structure. However, the time increase is not the same for the used platforms. For example, in case of the ATSAMD21G18A MCU, operations writing objects to the statically allocated arrays results in an increase of 197%–304%, depending on the size of the structure being written. For the other two platforms, this increase for the same operations is at most 56%. The reason for this relatively moderate increase compared to that of ATSAMD21G18A is likely due to the instruction caching mechanism available for XMC4500 and STM32F767ZI, but not for the ATSAMD21G18A.

Furthermore, the time needed for dynamic memory handling in the aligned and packed cases show negligible differences. This is not a very surprising result because the allocators ultimately rely on the sizes of data types and structures expressed in bytes, and they do not deal with the data that is to be written to the memory space.

On a positive note, the size of the data structure decreases considerably when data alignment is not enforced. The benefits are presented below.

- `Segment`: size drops to 17 bytes from 20 bytes, 15% reduction.

- `Turnout`: size drops to 25 bytes from 28 bytes, 11% reduction.
- `Train`: size drops to 16 bytes from 24 bytes, 33% reduction.
- `Object`: size drops to 25 bytes from 32 bytes, 22% reduction.

Thus, the model used in the evaluation decreases in size as well: dynamically allocated model occupies 2862 bytes (18% less), while the static model size drops to 3900 bytes (22% less).

The packed memory layout is able to achieve up to 33% side reduction of structs in our experiments, however, it comes at the cost of execution time penalty up to 300% compared to the aligned case, and possible runtime faults by unaligned accesses.

5.2.3 Threats to Validity

Construct validity. In our experiments, we used a self-contained metamodel fragment from the MoDeS3 demonstrator that was selected and validated manually. Furthermore, we assessed all possible model update operations supported by our proposed framework.

Internal validity. To mitigate the impact of nondeterminism present in the C library functions for each platform, we measured average execution times of model operations. This allowed to reduce the fluctuations of run times caused by factors which are external to our implementation.

External validity. This evaluation was done using one metamodel from the MoDeS3 domain. Performing the same comparisons for multiple diverse domains would improve the generalizability of the results and findings. The ARM-based microcontrollers used for running our runtime model benchmarks are well-known and supported by the GCC compiler. In case of new or lesser-known hardware, one may face significant limitations on the software level (w.r.t available libraries and developer tools). Thus, performing the evaluation scenario may require a significantly different evaluation setup, where results may not be comparable with the ones presented here.

5.3 Summary

This chapter showed the challenges of adapting runtime graph models to real-time embedded systems, where ensuring predictable timing behavior and respecting resource limits are important. We provided a solution to improve timing predictability of programs which read and write such runtime models. We implemented a prototype in C, and evaluated the timing properties of the presented solution using three different microcontroller platforms. The obtained results suggest that the proposed method for managing the lightweight data structure of the runtime model is sufficiently fast to update ~100 objects per millisecond even on the simplest MCU, thus it is suitable for capturing dynamically changing runtime information of CPSs. Furthermore, by comparing results from aligned and packed memory layouts, a trade-off between saving memory space and sacrificing execution time is dissected. Despite its potential benefits, we concluded that using packed memory layout is not recommended because it can be a cause for hard faults at runtime on platforms where unaligned accesses are not supported in the hardware, and it potentially restricts the allowed set of programming language constructs. In particular, this chapter introduced the results related to the first contribution group (Co1.1, Co1.3 and Co1.4) for real-time embedded systems.

Publications related to this chapter. A conceptually similar C data structure design is introduced in article accepted at to the journal ACM Transactions on Embedded Computing Systems [*j*1], and it is my contribution. The rest of the results presented in this chapter are yet to be published. The contributions of the coauthors of the related journal article [*j*1] are detailed in Section 7.8 where the majority of the results from the paper are presented.

Chapter 6

Query-Based Runtime Monitors for Real-Time Systems

Runtime monitoring is a technique where the system's state during operation is observed in order to decide if certain properties hold or violated. It is often employed as a complementary approach for design time testing, and in this thesis we use it as an added layer of safety for smart and safe CPSs.

This chapter shows how to synthesize monitoring programs from high-level query specifications that are deployable to real-time embedded systems. In Section 6.1, we introduce data-driven safety monitors derived from high-level specifications that analyze aggregated changes triggered by complex sequences of atomic events by evaluating queries over a continuously evolving runtime model. Then, we show how to derive monitoring programs automatically from the query specifications in Section 6.2. Finally, we evaluate the performance of the generated monitoring programs in Section 6.3 to show the scalability of the approach.

6.1 Data-Driven Runtime Monitors by Graph Queries

Queries capture safety properties at a high level of abstraction by focusing on structural dependencies between system entities. Informally, we use graph queries to capture potentially unsafe situations that may occur at runtime. In this work, we use first-order logic (FOL) predicates to capture definitions of graph queries over instance models (see formal definition of graph queries in Definition 7). Similarly, the OCL standard has been used by Iqbal et al. [Iqb+15] for related purposes.

Graph queries of runtime monitors are *evaluated over a snapshot of the runtime model which reflects the current state of the monitored system*, e.g., data received from different sensors, the services allocated to computing units, or the health information of computing infrastructure. In accordance with the models@runtime paradigm [BBF09; SZ13], observable changes of the real system gets updated — either periodically with a certain frequency, or in an event-driven way upon certain triggers.

Classical *event-based runtime monitors* rely on some temporal logic formalism to detect sequences of events occurring in the system at different points in time, while the underlying data model used in such monitors is restricted to atomic propositions. On the other hand, *data-driven runtime monitors* defined by graph queries can check structural properties of a runtime model that represents a snapshot of the underlying system. In other words, they focus on the data available on the underlying system at a given point of time (rather than detecting the sequence of events that evolved the system into the particular state).

As such, event-based and data-driven monitors are complementary techniques. While graph queries can be extended to express temporal behavior [DRV18], the current work is restricted to (structural) safety properties where the violation of a property is expressible by graph queries.

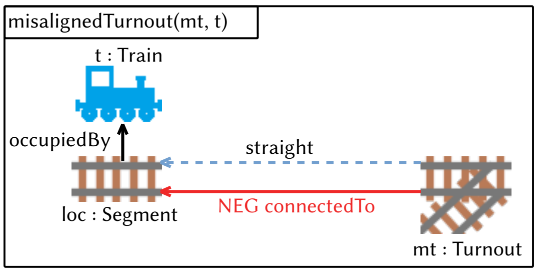
To capture the safety properties to be monitored, we rely on the VIATRA Query Language (VQL) [Ber+11]. VIATRA has been intensively used in various design tools of CPSs to provide scalable queries over large system models. This thesis aims to reuse this declarative graph query language for runtime verification purposes, which is a novel idea. The main benefit is that safety properties can be captured at a high level of abstraction over the runtime model, which eases the definition and comprehension of safety monitors for engineers, especially, when compared to monitors written in an imperative language. Moreover, this specification is free from any platform-specific or deployment details, and deployable monitoring programs can be automatically generated from such predicates (as we show this later in Section 6.2).

Technically, a graph query captures the erroneous case, when evaluating the query over a runtime model. Thus any match (result) of a query highlights a violation of the safety property at runtime. This language enables to specify a hierarchy of runtime monitors as a query may explicitly use results of other queries (along pattern calls).

Example 14. On a railway track, a *misaligned turnout* (MT) refers to a state where a turnout is set to a direction that differs from the direction of an incoming train. Trains

$$\varphi_{\mathrm{MT}}(mt, t) = \exists loc : \texttt{OccupiedBy}(loc, t) \land \texttt{Turnout}(mt) \land \texttt{Straight}(mt, loc) \land \neg\texttt{connected}(loc, mt)$$

(a) Graph query as logic predicate



(b) Graphical query presentation

```
pattern misalignedTurnout(mt, t) {
  Segment.occupiedBy(loc, t);
  Turnout(mt);
  Turnout.straight(mt, loc);
  neg find connected(loc, mt);
}
private pattern connected(s1, s2) {
  Segment.connectedTo(s1, s2);
}
```

(c) Description of a query and its subquery in VQL

Figure 6.1: Monitoring goal formulated as a graph query φ_{MT} for misalignedTurnout

passing through such misaligned turnouts can damage the railway equipment and can lead to derailment [MD15]. Query φ_{MT} shown in Figure 6.1a captures a (simplified) hazardous case and identifies violating situations. The query returns pairs of trains t and turnouts mt where the train is located on a segment loc that is the straight continuation of the turnout, but the turnout is currently not connected to this segment. Any match of this query highlights a train and a turnout where immediate action (stop the train or switch the direction of the turnout) is required. Figure 6.1b shows the same graph query in graphical presentation (used in modeling tools). Figure 6.1c shows the textual description of φ_{MT} using VIATRA Query Language (VQL) [Ber+11], which is a graph query language often used in CPS design tools.

6.2 From Declarative Queries to Executable Programs

Constructing effective plans for graph queries that can effectively evaluate a query over instance models is a complex challenge. It is outside of the scope of the current thesis and has been formerly extensively studied (see, e.g., [HVV07; Var+15] for possible solutions). Instead, we focus on the adaptation of graph queries to real-time resource-constrained environments, and we present a pseudo code that generates program code suitable for deployment to a CPS component from a search plan in Algorithm 6.2.1. The *CompileSearchPlan* function is parameterized with a search plan and a given search step index. Line 2 returns a code snippet to register a match if the provided index is beyond the index of the final search step. Otherwise, the search step is extracted (line 3) and the variable *matcherCode* to hold the generated code is

Algorithm 6.2.1: Code generation from search plans

```
1  Function CompileSearchPlan(sp, idx) is
2      if idx > sp.size() then return code for storing a match;
3      step = sp[idx]
4      matcherCode = ""
5      if step is extend then
6          for uv ∈ step.getFreeVariables() do
7              matcherCode +=
8                  AddLoopFor(uv, step.getConstraintFor(uv))
9          end
10     else if step is check then
11         matcherCode +=
12             AddIfFor(step.getAllVariables(), step.getConstraint())
13     end
14     return  matcherCode + CompileSearchPlan(sp, idx + 1)
15 end
```

Table 6.1: A possible search plan for query misalignedTurnout where free variables are underlined

Constraint	Step#	Op. type
Turnout($\underline{mt}$)	1	extend
Straight(mt, $\underline{loc}$)	2	extend
¬ConnectedTo(loc, mt)	3	check
OccupiedBy(loc, $\underline{t}$)	4	extend

initialized to an empty string (line 4). Then, if the current search step is an **extend**, it iterates over all free variables (line 6) and generates a series of embedded *for loops* to bind these to the respective candidate model objects selected by the constraint in the step (lines 7-8). Otherwise, the current step is a **check** (line 10) and an *if* condition (lines 11-12) is inserted. Finally, in line 14, the generation continues recursively appending the code generated from the subsequent steps to the result. The query code for the entire search plan *sp* can be generated by calling *CompileSearchPlan*(*sp*, 1).

Example 15. Table 6.1 shows a possible search plan for the φ_{MT} query. Each row represents a search operation. The first column (Constraint) shows which constraint is enforced by the given step, the second column is the assigned operation number (or step). The third column shows the search operation type (check or extend) which is based on the variable bindings prior to the execution of the search operation: if the constraint parameters are all bound (i.e., none of the variables are underlined in Table 6.1), then it is a check, otherwise, it is an extend.

Data-driven monitors aim to find matches of graph queries over the entire runtime graph model using a local search-based query evaluation strategy. When such graph queries are used in a real-time system, they need to retrieve all matches of a query in the model by a deadline. This is carried out by using a depth-first search graph traversal algorithm derived from the search plan of the query. This keeps the memory footprint of the algorithm constant, thus only the graph data may change over time as the model evolves.

```
 1 void mt_matcher(MTMatchSet *results) {
 2   MTVars vars = { mt = NULL, loc = NULL, t = NULL };
 3   int match_cntr = 0;
 4   // Constraint: Turnout(mt)
 5   for(int i0 = 0; i0 < model->turnout_cnt; i0++) {
 6     vars.mt = model->nodes[model->turnout_ids[i0]];
 7     // Constraint: Straight(mt, loc)
 8     vars.loc = mt->straight;
 9     if(vars.loc != NULL) {
10       // Constraint: ¬Connected(mt, loc)
11       int is_connected = 0;
12       is_connected |= vars.loc->connected_to[0] == vars.mt;
13       is_connected |= vars.loc->connected_to[1] == vars.mt;
14       if(is_connected == 0) {
15         // Constraint: OccupiedBy(loc, t)
16         vars.t = mt.loc->train;
17         if(vars.t != NULL){
18           // Register match
19           results->matches[match_cntr].mt = vars.mt;
20           results->matches[match_cntr++].t = vars.t;
21         }
22       }
23     }
24   }
25   results->size = match_cntr;
26 }
```

Listing 6.2.1: Source code generated for query misalignedTurnout

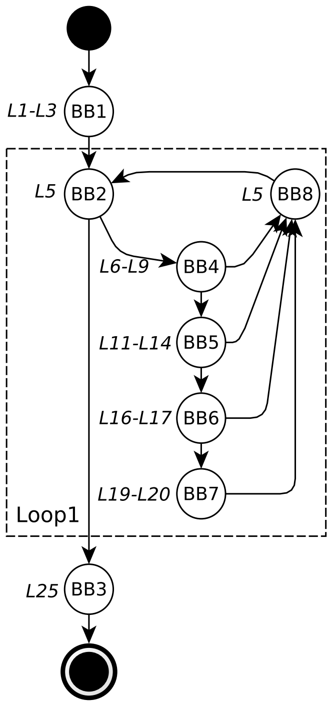


Figure 6.2: CFG of the function `mt_matcher`

The operations of the query search plan are translated to structured imperative code:

- Each **extend** operation is either a single assignment to a variable or a `for` loop iterating over a set of candidate variable bindings, depending on the multiplicity of the respective navigation edge (reference constraint).
- Each **check** operation is mapped to an `if` statement that checks whether the current variable binding satisfies a given condition created from the query constraint.

As a result, the source code contains a deep hierarchy of for-loops and if-statements embedded into each other along the ordering of predicates prescribed by the search plan. Such a template-based code generator for Java is presented by Varró et al. [VAS12].

We also need to estimate the number of matches of a query to allocate appropriate space in memory in advance to guarantee a predictable behavior. In the case of runtime monitors of safety properties, we can assume that only a few violating matches will be detected [Sem+20], thus the query result set is expected to be small and memory required for storing matches can be reserved at compile time.

Example 16. Listing 6.2.1 shows the C code generated from the query specification of misalignedTurnout. Assuming that a global variable `model` points to the root of the entire graph model including its up-to-date model statistics, calling the function `mt_matcher` with a pointer to the result set structure `results` will compute and store all matches over the model in `results`.

In the example, the initially bound variables are assumed to be empty, as indicated in Line 2 (L2 for short) with `NULL` values, because we aim to find all matches in the entire model. In L3, the size of the result set is initialized to `0`. The `for` loop in L5 represents step 1 from the search plan (see Table 6.1) and iterates over all turnouts in the model, binding the variable `vars.mt` to all possible objects in L6. Lines 8 and 9 together represent search step 2. In L8, the `vars.loc` is assigned the segment referred by `vars.mt` via the link `straight`. If such a segment exists in L9, execution continues with the third search operation that is mapped to L11–L14.

The generated code for ¬`ConnectedTo` (line 3 in Table 6.1) checks (as negative condition) if the `vars.loc->connected_to` array holds a pointer to the turnout `vars.mt`. The execution only continues if no such reference exists, i.e., ¬`ConnectedTo = 1` (see L14). The final step of the search plan is mapped to L16 and L17. Here the train occupying the segment stored in `vars.loc` is assigned to `vars.t`. If such a train exists, a match is found and registered by assigning the corresponding variable values to parameter variables in a new match (L19 and L20) and increasing the counter of found matches `match_cntr`. The execution concludes with saving the number of matches (L25).

6.3 Evaluation

We performed experiments to address the following research questions:

RQ1 How does the query evaluation scale with increasing model size?
RQ2 What is the difference in run times when measured on different embedded computers?
RQ3 What is the overhead of subquery calls in query programs?

6.3.1 Measurement Setup

Computation platform

We use devices present or very similar to the ones in the physical platform of MoDeS3 railway CPS demonstrator to answer our research questions (instead of setting up a virtual environment). One of these devices is a *BeagleBone Black* (BBB) device, which is frequently used in Edge computing applications. This device features an AM335x 1GHz ARM Cortex-A8 processor with 512MB available DDR3 RAM and runs an embedded Debian Jessie with PREEMPT-RT patch. In addition, we used three different microcontroller devices having limited computing capacities and available memory. These microcontrollers run bare-metal (aka super loop) monitoring programs, without any operating system, and they are representative to the hardware platforms used in CPSs.

In the present setup, an instance model (i.e., runtime model snapshot) is preloaded in the (limited) memory of the device, and monitoring queries are evaluated over the model. We now only focus on query evaluation, model update was assessed in Section 5.2.

CPS monitoring benchmark

Graph models. The original snapshot of the runtime model of the CPS demonstrator only has a total of 24 objects. For the scalability evaluation, we replicate the original elements to reach sizes sizes of 49 – 43K objects and 114 – 109K links and pre-load the model into the memory of a BBB. These scaled-up models share structural properties with the original one.

To assess the query evaluation times on various devices common in CPSs, we use the same initial model with 24 objects and generated scaled-up models with up to 480 objects. This upper limit on model size was determined by the limited available memory in these devices.

Queries. To assess the query-based runtime monitors, we used multiple safety properties from MoDeS3 (see their definitions in Appendix A). They are all based on important aspects of the domain, and they have been integrated into the monitoring components. Our properties of interest are the queries described in Example 1. The monitoring query misalignedTurnout in this evaluation had a slightly simplified definition compared to Appendix A, and this version is the one presented in Example 14.

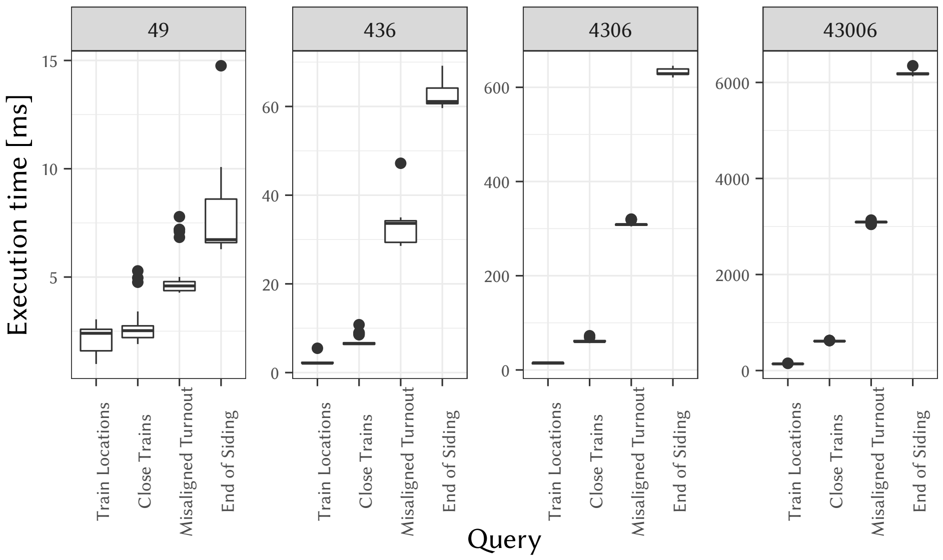


Figure 6.3: Scalability evaluation of query execution on a BeagleBone Black SBC

6.3.2 Measurement Results

Scalability evaluation

The query execution times over models deployed to a single BBB were measured to obtain a scalability of the evaluation for each monitoring goal. In Figure 6.3, each box captures the times of 29 consecutive evaluations of queries excluding the warm-up effect of an initial run which loads the model and creates necessary auxiliary objects. A measurement starts as query execution begins, and terminates when all matches from the entire model are collected. The programs deployed to the BBB were compiled with clang++ 4.0. Furthermore, Figure 6.4 shows the average run times for each query over different model sizes.

Findings for RQ1. In Figure 6.3 and Figure 6.4, queries show linear scaling with increasing model size up to 43k objects. In our experiments, queries with subquery calls in their definitions (**mt** and **eos**) take longer to execute than the ones (**ct** and **tl**) without such constructs. However, the length of a declarative query definition (i.e., the number of constraints in a query) cannot forecast the relationships between query execution times. The reason for this lies in the declarative nature of the high-level query definitions. These definitions, although easier to comprehend due to their conciseness, they hide the exact steps required for their evaluations.

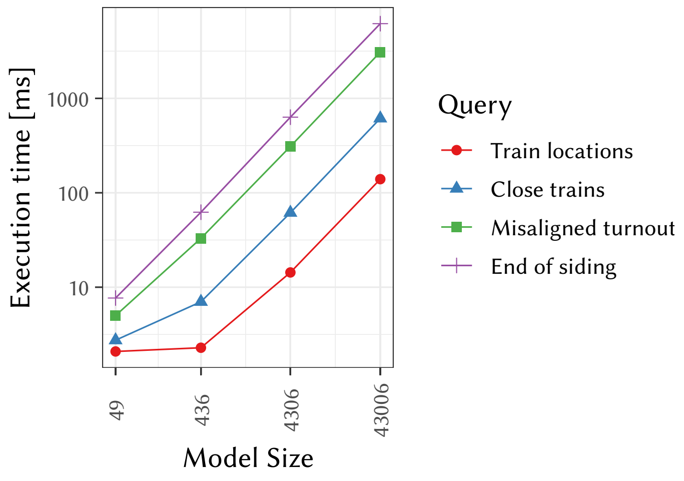


Figure 6.4: Query evaluation average times on a BBB SBC across models with different sizes

Up to medium-sized models, a 10× increase in model size yields approximately 10× increase in the average of the measured execution times for each query, which suggests linear scaling w.r.t model size.

Comparison of query run times over different microcontrollers

To see a comparison between query run times on different embedded devices, we picked three different well-establised microcontrollers with ARM Cortex-M0+, Cortex-m4, and Cortex-M7 cores: Atmel ATSAMD21G18A, Infineon XMC4500, and STM STM32F767ZI.

The bare-metal query programs were compiled with the default GCC compiler for ARM version 7.2.1 20170904 compile flags. We executed them on each microcontroller while no other tasks (e.g., interrupts) were scheduled. Because these programs take tens or hundreds of microseconds to execute, we applied the following measurement setup to obtain an estimate of average program run times:

1. First, we created an infinite loop and observed the loop execute n times under a fixed time duration T_{meas} (in our case $T_{meas} = 1s$).
2. Then, we added query execution to the loop and repeated it over the same graph, and counted m loop executions under T_{meas}.
3. Finally, we used the formula $T_{query} = (\frac{1}{m} - \frac{1}{n}) \cdot 10^6 \mu s$ to get the average run times of query evaluations on given input models.

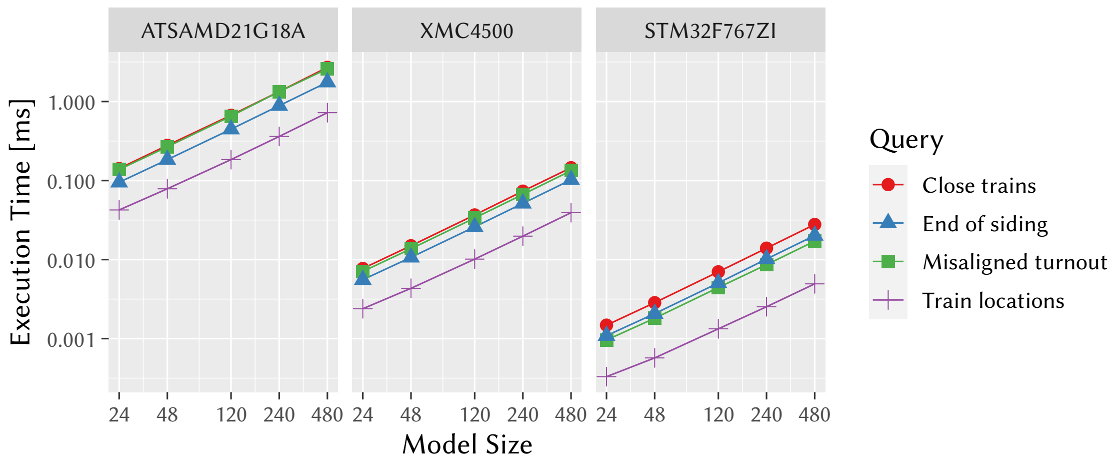


Figure 6.5: Query execution times on three different ARM-based microcontrollers

Repeated measurements of the average program run times obtained this way show negligible variation (in the order of a microsecond) for a given input graph, which is unsurprising since it is the only task running in the system. Precision of the measurement could be further improved, for example, by increasing the duration of the measurement which is currently set to one second. The results obtained by using the described method are presented in Figure 6.5.

Findings for RQ2. For each query execution on each platform, we can observe linear scaling of execution time with the size of the model. The biggest execution time differences between the used microcontrollers is attributed to the different CPU frequencies driving these devices and the presence (or absence) of instruction and data cache. For these particular query programs, instruction cache alone seems to greatly reduce execution time, as the difference in CPU frequency between ATSAMD21G18A and XMC4500 is only 2.5× (with the latter having a higher frequency), the measured run times are approximately 10× shorter.

The recurring control flow of query programs exploits the cache to a high degree, thus these programs have much faster execution times on devices with instruction cache.

Impact of optimized query evaluation code

We assess the impact of optimizations applied to the C microcontroller code generated from the query definitions. In the case of the two queries with subquery calls (**mt** and **eos**), we compare the execution times of the implementations generated directly from the query definitions with the ones where we inlined subquery calls to optimize the program. In practice,

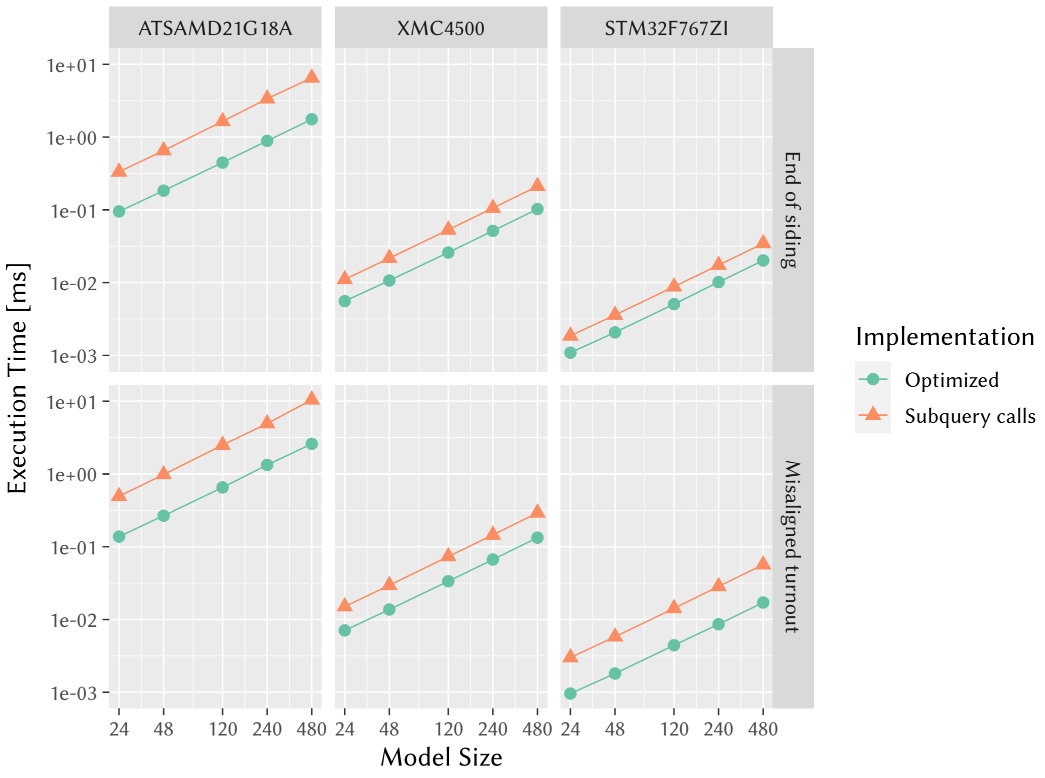


Figure 6.6: Comparison of initial and optimized query execution times on three different ARM-based microcontrollers

separate C functions are generated from each declarative query allowing query composition via function calls in the former case, while in the latter case, these functions are stand-alone implementations of the complete query with no code reuse between them. The results are presented in Figure 6.6.

Findings for RQ3. We can see a factor of 1.7–2.7 between the observed run times of the two versions of query programs with the optimized version being faster and no significant differences in characteristics on the three devices. This makes the overhead of explicit subquery calls apparent (independently from execution platform), and for applications that are not only real-time but also need fast execution times, one can achieve performance gains at the cost of using slightly more program memory in a device.

Optimizing query code by inlining C function calls is a favorable way to reduce run time.

6.3.3 Threats to Validity

Construct validity. The presented evaluation uses four different monitoring queries with different complexities from the MoDeS3 demonstrator. The imperative implementation synthesized from the high-level description of the queries follows the structure presented in Section 6.2, and each implementation was validated manually.

Internal validity. The software component used in the evaluation relies on no external software dependencies (e.g., operating system and standard library). As such, a well-controlled environment with precise measurement of query evaluation times is used. However, the presented evaluation relies on built-in debugger features of the hardware platform, which means that the precision of the presented results relies on the precision of such debugger features.

External validity. The presented evaluation relies on a single case study (MoDeS3). Performing the query benchmarks on additional case studies (other domains) would help improve the confidence in the results. Moreover, it would be useful to assess the execution times in an environment where other tasks are running in the system as well. This setup could provide a better insight into the impact of hardware caches with a more realistic workload.

6.4 Summary

This chapter showed how graph queries can capture safety monitoring objectives focusing on structural properties on a high-level of abstraction. We provided an algorithm to generate imperative code from the declarative query definitions. Furthermore, we deployed four different monitors to various devices used in the MoDeS3 demonstrator, and evaluated the scalability of the approach on them and compared the measured times across different devices. Our results show that query programs can handle up to 43k graph model objects. Furthermore, we also found that manual code optimization and available instruction cache can greatly speed up execution of query programs on microcontrollers. The results in this chapter are part of the second contribution group (Co2.1, Co2.4, and Co2.5).

Publications related to this chapter. The theoretical foundations of query-based runtime monitors, which is first presented in the proceedings of International Conference on Fundamental Approaches to Software Engineering [*c*5], is my contribution. Furthermore, in the paper [*c*5], the evaluation of the approach on the BBB hardware was carried out by Gábor Szilágyi, while András Vörös was providing continuous feedback on the work.

Chapter 7

Timing Analysis of Embedded Query Programs

In this chapter, we provide timeliness guarantees for runtime monitoring programs generated from high-level query specifications. Providing strict timeliness guarantees enables their use in a hard real-time setting. While it is possible to compute safe worst-case execution time (WCET) bounds by using existing static WCET estimation techniques, they may greatly overestimate the WCET of query-based monitors as they are unable to exploit domain-specific information about the input models. In this chapter, we introduce the challenges of computing WCET for query-based monitoring programs and provide timing analysis solutions to complement the current state-of-the-art WCET analysis approaches by automatically considering domain-specific constraints on the input data.

In Section 7.1, we discuss the challenges of the timing analysis of such query-based runtime monitors. Section 7.2 overviews the state-of-the-art analysis techniques and presents two complementary WCET analysis methods. The first static (design time) method relies on precise execution time estimation over a specific graph based on low-level analysis results from existing tools (Section 7.3). Then, in Section 7.4, this execution time calculation is used as the objective function of a modern graph solver, which allows the systematic generation of input graph models up to a specified size (referred to as *witness models*) for which the monitor is expected to take the most time to complete. Hence the estimated execution time of the monitors on these graphs can be considered as safe and tight WCET. In Section 7.5, we show a complementary solution to provide rapid recomputation of run time estimates over a given model to support cases when models are beyond the size of the witness model. To combine the static and dynamic WCET estimation methods, we propose hybrid WCET estimation in Section 7.6.

Finally, we perform experiments with query-based programs running on a real-time platform over a set of generated models to investigate the relationship between execution times and their estimates, and compare WCET estimates produced by our approach with results from two well-known timing analyzers, aiT and OTAWA.

7.1 Timing Analysis Challenges

Estimating the WCET of query-based monitors is a highly complex task which involves multiple classic challenges of timing analysis. The runtime model of the system is a continuously changing data structure that captures an up to date snapshot of the underlying running system. Hence, it is not sufficient to analyze execution time on a single input model, but all models possible at runtime must be considered.

However, the space of possible models is enormous. For example, in a metamodel with a single class and 3 reference types each having $0..*$ multiplicity bounds, there may be up to $2^{3 \cdot 25 \cdot 25} = 2^{1875}$ distinct models with 25 objects. In this simple case, the following explins this estimate: there may be three different types of edges pointing to each object (including itself) in the model. This results in $2^{3 \cdot 25}$ different configurations of outgoing edges for an object, and there are a total of 25 objects which maintain their outgoing edges independently from other objects in the model, yielding $2^{3 \cdot 25 \cdot 25}$ possibilities. Thus, explicit enumeration of graph models is intractable, which necessitates the use of abstractions.

Another major challenge is that *query execution time is heavily data-dependent*, i.e., the same control flow of a query program may have substantially different run times based upon the structural characteristics of the underlying graph model. Assuming some constraints on model size (e.g., capped by available memory) and some general restrictions on model scope (e.g., there are more segments than trains in any real model), a key open challenge is *how to provide a model where the execution time of a particular query program is maximal.* In this thesis, we propose systematic generation of *witness models* that maximize an estimate of the run time, which aids in WCET analysis and in identifying bottlenecks in query execution.

Moreover, is also important that the same graph model can be represented in memory in many ways, and *different placements of the same data can cause different run times.* For example, two memory image of the same graph may differ in the order the objects are stored in an array. For this reason, two different in-memory representations of the same graph may yield different run times, which must be considered when computing WCET of graph query programs. Thus, WCET estimation even for a single concrete input model must tackle the dependency of execution paths on the data representations. As a single model of n objects has $n!$ possible in-memory representations even when inserting the objects into a single continuous linear array of n elements, explicit enumeration is again intractable.

7.2 Graph Model-Based WCET Estimation

7.2.1 Existing WCET Analysis Methods

Static WCET analysis is traditionally divided into two major phases: *high-level* or *flow analysis* and *low-level analysis*. On the one hand, flow analysis aims at reconstructing the program flow and deriving control flow graphs (CFGs), which are then annotated with additional information called *flow facts* about the execution. The nodes of such CFGs are *basic program blocks* (BBs), which represent series of instructions in the program with one entry and one exit point. On the other hand, low-level analysis aims at computing hardware-specific timing parameters of BB executions. Note that our current work primarily focuses on flow analysis of WCET estimation while low-level WCET analysis is out of scope and we rely on low-level analysis results from other approaches.

A common flow analysis approach for static WCET computation is the implicit path enumeration technique (IPET) [LM95]. This method analyzes the control flow of the program to compute a sequence of instructions that yields the longest possible execution. IPET is based on solving an integer linear programming (ILP) problem constructed from the program CFG and flow facts.

The IPET method necessitates costly computations to solve the underlying ILP problem. For that reason, it is only applicable for design time WCET computation for real-time systems (but not for WCET recomputation at runtime). However, the symbolic method proposed by Ballabriga et al. [BFL17] is capable of providing parametric WCET formulae which are cheap to recompute in case the program flow facts change. The drawback of this approach is that the overestimation in the produced WCET can be slightly larger than the one produced by IPET. However, with more concrete information available at runtime about the model, this estimate computed at runtime (on-line) can provide lower bounds than the statically (off-line) computed IPET.

Table 7.1: WCET analysis approaches for data-driven runtime monitor programs

	Inputs	Outputs
CL	$\overbrace{\text{HW description + Query program}}^{\text{Low-level analysis inputs}}$	WCET estimate
VAL	HW description + $\overbrace{\text{Query program + Concrete model + Memory image}}^{\text{Value analysis inputs}}$	WCET estimate for single memory image
DS_M	HW description + $\overbrace{\text{Query program + Concrete model}}^{\textbf{Domain-specific high-level analyis inputs}}$	WCET est. for single model
DS_Σ	HW description + Query program + **Constraints**	WCET est. for all valid models + **Witness model**
OLS_M	HW description + Query program + **Condensed Model Statistics**	WCET est. for a single model + **Recomputable at Runtime**

7.2.2 Comparison of Timing Analysis Approaches

Table 7.1 illustrates the relationships between the existing and proposed approaches to the WCET analysis of query programs. *Classical* (CL) WCET analysis is based on the binary code of the query program and the characteristics of the hardware platform, but does not consider structure and the well-formedness of the input runtime models.

> **Example 17.** Static analysis of the code of the misalignedTurnout query itself in Listing 6.2.1 would not impose any restrictions on line 17 despite the fact that the domain-specific scope constrain prescribe that $\mathcal{S}(\texttt{Train}) = [0, 3]$. Therefore, the condition in this line can evaluate to true up to three times on well-formed models within the model scope, yielding a flow constraint that is not discoverable by analysis of the code only.

Value analysis (VAL) can derive more precise WCET estimates for executing a query on a single memory image (comprised of a single concrete model). However, it is unable to consider equivalent in-memory representations of the same concrete model, or to cover all possible consistent concrete models, thus it is unsuitable for the analysis of data-driven monitors (thus it is greyed out in Table 7.1).

To alleviate this issue, we propose two *domain-specific* (DS) static WCET analysis methods for data-driven monitors. We introduce the concept of *witness models*, which are consistent models that are *feasible inputs of the graph query program* and *maximize the WCET estimate* *for all models within the given scope.* They serve as representative data to calculate WCET for *any* model within the scope.

- First, we estimate WCET for a *single concrete model* (we refer to this method as DS_M). The estimate is valid for all in-memory representations of a given concrete model M.
- In the second case, the set of possible runtime snapshots is specified with *metamodel* $\langle \Sigma, \alpha \rangle$ along with well-formedness and scope constraints (DS_Σ). This WCET estimate is valid for all possible runtime snapshots within the memory limits of the system, i.e., for all consistent instances of the metamodel up to the size specified by the scope constraints.

Furthermore, we propose to adapt a symbolic WCET formula [BFL17] for graph query programs to allow fast recomputation of WCET at runtime using *on-line statistics* (OLS_M) of the runtime model snapshot M to provide support for cases when M no longer fits the model scope used for deriving witness models for a query. The parametric WCET formula relies on condensed model statistics about the graph that is available at runtime.

7.2.3 Overview of the Approach

We address the WCET estimation challenge for graph query programs in *two complementary ways*: the *static* DS_Σ approach (used at design time) and the *on-line* OLS_M approach (usable at runtime) methods. On the one hand, DS_Σ can provide upfront WCET bounds for models within the model scope by synthesizing and exploiting M^* witness models. This estimation achieves tighter WCET bounds (compared to other existing methods) by (1) precisely incorporating data flow information during WCET estimation and by (2) excluding unrealistic models that would often yield high WCET estimates.

On the other hand, when a runtime model snapshot ends up being outside all (statically determined) model scopes used for obtaining witness models, no design time WCET estimates would be available. Therefore, as a fallback strategy, we provide OLS_M to rapidly recalculate WCET at runtime by adapting parametric WCET formulae [BEL11; BFL17] and exploiting some aggregated model statistics.

To illustrate the relationship between possible input models, Figure 7.1 sketches the *model space* of runtime graph models (represented with dots), i.e., the set of all input models. Possible changes made to a model at runtime (depicted as arrows) result in a new model. To obtain a safe and tight WCET estimate for query programs, we make some assumptions about realistic

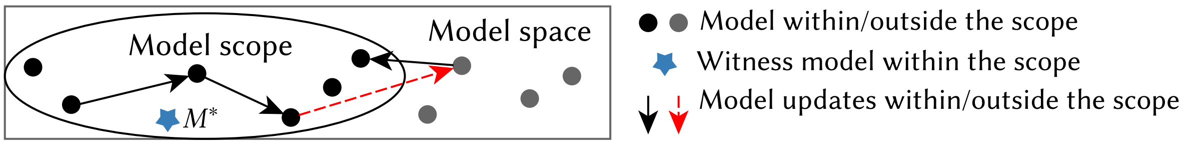

Model scope
Model space
M^*
● ● Model within/outside the scope
Witness model within the scope
Model updates within/outside the scope

Figure 7.1: Classification of query input models and model updates from the perspective of WCET analysis

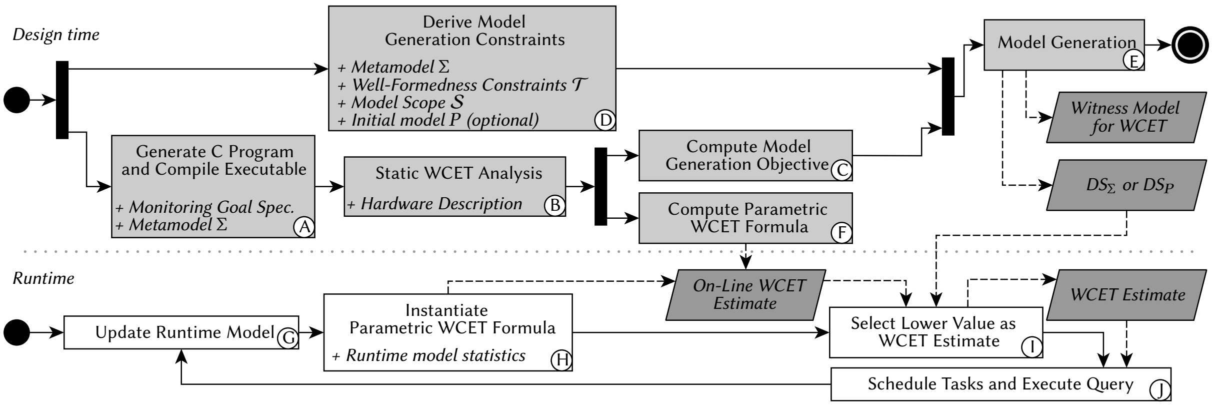

Design time
Derive Model Generation Constraints
+ Metamodel Σ
+ Well-Formedness Constraints 𝒯
+ Model Scope 𝒮
+ Initial model P (optional)
D
Model Generation
E
Witness Model for WCET
DS_Σ or DS_P
Generate C Program and Compile Executable
+ Monitoring Goal Spec.
+ Metamodel Σ
A
Static WCET Analysis
+ Hardware Description
B
Compute Model Generation Objective
C
Compute Parametric WCET Formula
F
Runtime
On-Line WCET Estimate
WCET Estimate
Update Runtime Model
G
Instantiate Parametric WCET Formula
+ Runtime model statistics
H
Select Lower Value as WCET Estimate
I
Schedule Tasks and Execute Query
J

Figure 7.2: Workflow of WCET estimation for query-based monitors

(and consistent) models captured in the form of a *model scope*. The witness model M^* for the consistent instances of the metamodel within the model scope is depicted as a blue star in Figure 7.1. If the runtime model snapshot lies outside the model scope (gray dots), then the result of OLS_M can be used as the WCET as a static WCET estimate based on a witness model is not available.

Figure 7.2 shows both design time and runtime aspects of our proposed approach. In the top part, the high-level workflow with design time tasks of obtaining two complementary WCET estimates is presented. The static WCET estimation DS_Σ relies on objective-guided generation of witness models, where the objective function is derived from the monitoring goal (i.e., the high-level query specification) and the low-level timing properties of the monitoring program. Any additional external input during this process is marked with + in the process overview. The process starts with the synthesis and compilation of the monitoring program (marked with A in Figure 7.2) where both monitoring goal specification and the metamodel are inputs from the user, and it is followed by classic WCET analysis, e.g., using the IPET method (B). Subsequently, the results of the WCET analysis are used to compute a model generation objective (C). This objective is to maximize the function that computes the execution time of the query program over a given instance model.

In parallel with these activities, constraints are derived for generating well-formed models in the given domain (D). Combining the results from activities C and D, the model generation step (E) uses a graph solver to systematically generate the model that maximizes the objective function, i.e., the WCET estimation. As a result, the workflow not only computes a *safe, static WCET value*, but generates a *witness model* where the WCET estimation is maximized. This thesis focuses on how to use the model generator to obtain witness models, but the technicalities of the underlying model generation process are out of scope [Mar+20].

The proposed on-line WCET estimation OLS_M starts with similar steps as DS_Σ by obtaining the source code and executable for the query program (A), then performing static WCET analysis (B). Using the results of (B), a parametric WCET formula is derived (F) using the algorithm proposed by Ballabriga et al. [BFL17]. While obtaining this formula happens at design time, the exact WCET bounds are obtained at runtime once the relevant underlying runtime model statistics are known.

The bottom part of Figure 7.2 highlights the stages of real-time monitoring program execution. Once updates to the runtime model are completed (G), the parametric WCET formula is instantiated and a *safe, on-line WCET bound* is obtained (H). Computing a formula at runtime requires minimal computational effort, thus it can be repeatedly recomputed during program execution. The on-line and static WCET bounds (if the latter is available) are then simply compared, and because they are both safe estimates for the current model, the lower value is selected as WCET estimate (I). If there is no static estimate, the value provided by OLS_M is used. This value (computed at runtime) is then used as the required time window while scheduling of tasks (J), and after completing query evaluation and the rest of the tasks, program execution will eventually continue processing model updates (G).

7.3 Approximating Execution Time With Predicates

To characterize the data-dependent execution time of graph query programs, i.e., formulate DS_M, we derive an upper bound function f_q assigning approximate run times of the query q to model M. Formally, $RT_\text{q}(M) \leq f_\text{q}(M)$, where $RT_\text{q}(M)$ is the execution time of the query program q on instance model M.

For each basic block BB_i of the CFG of the query program q, we construct a graph predicate ψ_{BB_i}. The free variables $v_1, \ldots, v_k$ of ψ_{BB_i} correspond to the program variables within the program scope when BB_i is executed. When the query program runs on an input model M, each execution of BB_i corresponds to a match $Z\colon \{v_1, \dots, v_k\} \to O_M$ of ψ_{BB_i} ($Z \in \mathit{Matches}(M, \psi_{BB_i})$), where $Z(v_j)$ is the model object referenced by the variable v_j upon the execution of BB_i.

To achieve this, we set ψ_{BB_i} to the conjunction of **extend** and **check** constraints in effect on the variables in the program scope. **Extend** operations (evaluated by loops) introduce new free variables, while **check** operations (evaluated by `if` conditions) only restrict the possible binding of existing variables. As we have shown in Section 6.2, search-based query plans translate to a series of nested loops and `if` conditions. Thus, ψ_{BB_i} is the conjunction of **extend** constraints associated with loops and **check** constraints associated with `if` blocks that enclose BB_i. For enclosing `else` blocks, the negation of the **check** condition is taken instead.

Basics blocks BB^*_j of loop headers require special attention, since a loop variable v_k can be uninitialized or it may have a value from the previous iteration of the loop. Hence, in addition to the predicate $\psi_{BB^*_j}$ with free variables $v_1, \dots, v_{k-1}, v_k$, we also introduce a predicate $\psi'_{BB^*_j}$ with free variables $v_1, \dots, v_{k-1}$ to represent the first execution with v_k still uninitialized.

Definition 13. The upper bound for the execution time of `q` on a model M can be written using graph predicates by summing up the worst-case execution times of basic blocks weighted by the number of times each basic block is executed as follows:

$$\begin{aligned} f_{\mathtt{q}}(M) = &\sum_{i \in D} \big(T(BB_i) \cdot |\mathit{Matches}(M, \psi_{BB_i})|\big) \\ &+ \sum_{j \in L} \big[T(BB^*_j) \cdot \big(|\mathit{Matches}(M, \psi'_{BB^*_j})| + |\mathit{Matches}(M, \psi_{BB^*_j})|\big)\big], \end{aligned} \tag{7.1}$$

where

- T is a function that returns the WCET of a basic block in the CFG;
- D is the set of the indices of basic blocks that are not loop headers; and
- L is the set of indices of loops.

The function $f_{\mathtt{q}}$ is a linear function of the match counts of the graph predicates as defined in Section 3.3. Therefore it is not only an upper bound for the execution time of `q` on a given model M, but can also serve as an objective function in a model generation task created when using DS$_\Sigma$. In Figure 7.2, the function $f_{\mathtt{q}}$ is defined in activity C and used in activity E.

Example 18. We illustrate the execution time estimation method using the query misalignedTurnout. To construct the graph predicates ψ_{BB_i} and $\psi'_{BB^*_j}$ for the query program in Listing 6.2.1, we have to inspect the query plan in Table 6.1, its traceability to the generated code (shown as comments in Listing 6.2.1), and the CFG of the code (Figure 6.2). By tracing each basic block to the code lines and to the query constrains, we obtain

$$\psi_{BB_1} = 1, \quad \psi_{BB_2} = \texttt{Turnout}(mt), \quad \psi_{BB_3} = 1, \quad \psi_{BB_4} = \texttt{Turnout}(mt),$$
$$\psi_{BB_5} = \texttt{Turnout}(mt) \land \texttt{Straight}(mt, loc),$$
$$\psi_{BB_6} = \texttt{Turnout}(mt) \land \texttt{Straight}(mt, loc) \land \neg\texttt{Conntected}(mt, loc),$$
$$\psi_{BB_7} = \texttt{Turnout}(mt) \land \texttt{Straight}(mt, loc) \land \neg\texttt{Conntected}(mt, loc) \land \texttt{OccupiedBy}(loc, t),$$
$$\psi_{BB_8} = \texttt{Turnout}(mt).$$

Since BB_2 is a loop header, we also have $\psi'_{BB_2} = 1$ to account for the first, unconditional execution of BB_2 with the variable mt yet uninitialized. Therefore one can write the upper bound of the execution time on a model M as

$$f_{\texttt{mt}}(M) = \sum_{i=1}^{8} \big(T(BB_i) \cdot |Matches(M, \psi_{BB_i})|\big) + T(BB_2) \cdot |Matches(M, \psi'_{BB_2})|.$$

7.4 Witness Generation of Worst-Case Execution Time

In the static WCET analysis step of DS$_\Sigma$, we compute an upper bound of the execution time of a model query program given a set of constraints (defining the space of well-formed models) and the scope of the analysis at design time.

Given a theory $\mathcal{T}$ and type scopes $\mathcal{S}$, we derive the WCET estimate of a query program q of the set of models satisfying $\mathcal{T}$ and $\mathcal{S}$ by maximizing the upper bound function $f_{\texttt{q}}$.

Definition 14. This yields a *static* WCET estimate $WCET^s_{\texttt{q}}(\mathcal{T}, \mathcal{S})$ as the computation of this estimate necessitates the use of a model generator at design time. Formally,

$$WCET^s_{\texttt{q}}(\mathcal{T}, \mathcal{S}) = f_{\texttt{q}}(M^*), \text{ where } M^* \in optimal(\Sigma, \mathcal{T}, \mathcal{S}, f_{\texttt{q}}), \tag{7.2}$$

where M^* is a *witness model* of the maximum value of $f_{\texttt{q}}$.

Therefore, $RT_{\texttt{q}}(M) \leq WCET^s_{\texttt{q}}(\mathcal{T}, \mathcal{S})$ holds for all instance models $\mathcal{T}, \mathcal{S} \vDash M$.

We include the witness model M^* (with the used theory $\mathcal{T}$ and model scope $\mathcal{S}$) for illustration in Figure 7.3a which maximizes $f_{\mathtt{mt}}$ and yields the $WCET^{s}_{\mathtt{mt}}$ estimate for the query program `misalignedTurnout` from Listing 6.2.1. The theory $\mathcal{T}$ used in the generation process contained the multiplicity constraint $\varphi_{\mathtt{ct}-outDeg}$ that caps the out-degree of `connectedTo` references at 2.

The witness model M^* can be inspected to study the extreme execution time of the query program q and may aid in further query optimization. However, M^* is not necessarily an input where the real program WCET is exhibited: it may be the case that $RT_{\mathtt{q}}(M^*) < RT_{\mathtt{q}}(M_{worst})$ for some other model $\mathcal{T}, \mathcal{S} \vDash M_{worst}$, even though we still have $f_{\mathtt{q}}(M_{worst}) \leq f_{\mathtt{q}}(M^*)$ and $RT_{\mathtt{q}}(M_{worst}) \leq WCET^{s}_{\mathtt{q}}(\mathcal{T}, \mathcal{S})$.

Gradual refinement of the theory $\mathcal{T}$ and the scopes $\mathcal{S}$ can aid the designer in query program analysis. In particular, if the estimated $WCET^{s}_{\mathtt{q}}(\mathcal{T}, \mathcal{S})$ is too high, we may extend the set of constraints to $\mathcal{T}' \supsetneq \mathcal{T}$ to more precisely specify the space of well-formed models. Alternatively, if it is not feasible to further extend the theory of well-formedness constraints $\mathcal{T}$ and thus restrict the set of well-formed models, we may opt for constraining the model scope $\mathcal{S}$.

Proposition 1. For a query program q, theories $\mathcal{T}, \mathcal{T}'$, and model scopes $\mathcal{S}, \mathcal{S}'$ the following inequality holds (see proof sketch in Appendix B.):

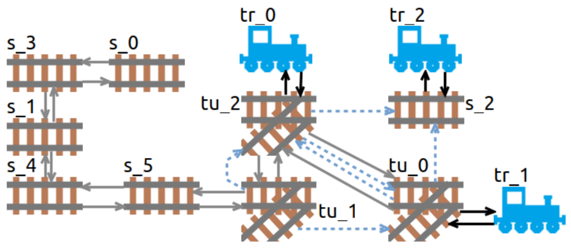
$$WCET^{s}_{\mathtt{q}}(\mathcal{T}', \mathcal{S}') \leq WCET^{s}_{\mathtt{q}}(\mathcal{T}, \mathcal{S}) \text{ if } \mathcal{T}' \supseteq \mathcal{T} \text{ and } \forall \mathtt{C}_i \in \Sigma \colon \mathcal{S}'(\mathtt{C}_i) \subseteq \mathcal{S}(\mathtt{C}_i). \qquad (7.3)$$

7.5 On-Line WCET Estimation for Graph Query Programs

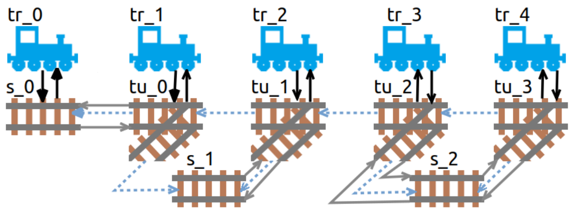
The primary goal of *on-line* WCET estimation OLS_M is to serve as a fallback to cover cases where the underlying runtime model M lies outside the model scope used for computing static WCET bounds or violates well-formedness constraints. Our idea is to exploit model statistics collected at runtime, such as (1) the number of nodes that are instances of a certain class or (2) the maximum out-degree of a node w.r.t a given reference type. As discussed in Section 5.1, these model statistics can be collected and maintained as part of the updates to the runtime model. As such, the current values of model statistics can be used as flow facts for loop bounds when instantiating a WCET formula of a specific query. The resulting WCET value can be used to reallocate execution time slots and reschedule tasks on-the-fly [CSB90].

$$\varphi_{\texttt{ct}-outDeg} = \texttt{ConnectedTo}(t, s_1) \wedge \texttt{ConnectedTo}(t, s_2) \wedge \texttt{ConnectedTo}(t, s_3) \wedge \neg(s_1 = s_2 \vee s_1 = s_3 \vee s_2 = s_3) \in \mathcal{T}$$

$$\mathcal{S}(\texttt{Segment}) = [0, 6], \qquad \mathcal{S}(\texttt{Turnout}) = [0, 3], \qquad \mathcal{S}(\texttt{Train}) = [0, 3]$$



(a) Witness model M^* for query misalignedTurnout with 12 objects and satisfying theory $\mathcal{T}$ and model scope $\mathcal{S}$



(b) Instance model M' with 12 objects and satisfying theory $\mathcal{T}$ but exceeding the model scope $\mathcal{S}$ (more Trains and Turnouts)

Figure 7.3: Illustrating model generation problems for witness models

In Section 6.2, we showed how search-based query plans can be translated to a series of embedded loops and if-conditions. Thus, the CFG of such a program has several cycles. We leverage the algorithm presented in [BFL17] that takes a program CFG and outputs a formula where the parameters are loop bounds, i.e., how many times a cycle in the CFG is executed.

Definition 15. A *parametric WCET estimation formula for a graph query program* q used to derive WCET bounds at runtime can be defined as follows:

$$WCET^{o}_{\mathrm{q}}(stats) = \sum_{i \in D_0} T(BB_{\mathrm{i}}) + \sum_{j \in L_0} \Big(T(BB_{\mathrm{h},j}) + l_j(stats) \cdot T(Loop_j, stats) \Big) \tag{7.4}$$

$$T(Loop_j, stats) = \sum_{k \in D_j} T(BB_{\mathrm{k}}) + \sum_{m \in L_j} \big(T(BB_{\mathrm{h},m}) + l_m(stats) \cdot T(Loop_m, stats) \big) \tag{7.5}$$

In these formulae

- *stats* is the model statistics related to the model scope of a given concrete model;
- l_k returns the loop bound of the k-th loop for a model statistics ($l_k(stats) \in \mathbb{N}$);
- T is a function that returns the WCET of a basic block or loop in the CFG;
- D_0 is the set of BB indices that are not contained in any loops but are part of the longest program execution path in the CFG of the query program q;

- D_j ($j > 0$) is the set of BB indices contained directly in $Loop_j$ (i.e., not part of other loops) that are part of the longest path within the loop;
- L_0 is the set of loop indices of loops that are not contained in any loop in the CFG of q;
- L_j ($j > 0$) is the set of loop indices of loops contained directly in $Loop_j$; and
- the BB for loop header of $Loop_j$ is denoted with $BB_{\mathrm{h},j}$.

Once $WCET^o_{\mathtt{q}}$ is formulated, it is easy to instantiate it because a multiplication is done for each parameter and then, the timing values are summed up. This computation is simple enough to instantly obtain a new WCET estimate when model statistics are available.

Example 19. Figure 6.2 shows the CFG built from the `mt_matcher` function with its corresponding BBs. The lines corresponding to BBs in Listing 6.2.1 are shown next to the nodes of the CFG. The WCET formula for the `mt_matcher` function built from the CFG shown in Figure 6.2 is the following:

$$\begin{aligned} WCET^o_{\mathtt{mt}}(stats) &= T(BB_1) + T(BB_3) + \sum_{j\in\{1\}} \Big(T(BB_{\mathrm{h},j}) + l_j(stats)\cdot T(Loop_j)\Big) \\ &= T(BB_1) + T(BB_3) + T(BB_2) + l_1(stats)\cdot\left(T(BB_2) + \sum_{k=4}^{8} T(BB_k)\right) \end{aligned}$$

Here $T(BB_i)$ is the WCET of a basic block BB_i and the value of $l_1(stats)$ is the flow fact for loop bound, which is in this case the number of turnouts in a given model.

To illustrate the impact of model statistics, we compare the model M^* presented in Figure 7.3a with model statistics $stats_{M^*}$ with the one obtained from a synthetic but still well-formed runtime model snapshot M' shown in Figure 7.3b with model statistics $stats_{M'}$. Both models have a total of 12 nodes but their model statistics (i.e., the number of instances of each class) are different; $stats_{M^*}(\mathtt{Turnout}) = 3$, while for the other model $stats_{M'}(\mathtt{Turnout}) = 4$. For this reason, the query program `mt_matcher` takes longer time to complete when evaluated over M'. The query plan starts with iterating over all turnout nodes, so the $WCET^o_{\mathtt{mt}}$ parameter is $l_1(stats_{M^*}) = 3$ for M^*, while $l_1(stats_{M'}) = 4$ for M'.

The static WCET estimate of a query program for some particular scope of models $\mathcal{S}$ may not be tighter than the on-line WCET computed for a model M in the scope. It may be the case that $WCET^o_{\mathtt{q}}(stats_M) \leq WCET^s_{\mathtt{q}}(\mathcal{T}, \mathcal{S})$ even if $\mathcal{T}, \mathcal{S} \vDash M$, especially when $stats_M$ is much smaller than the $stats_{M^*}$ belonging to the witness model M^* providing the static estimate.

However, for any (well-formed) model M, $WCET^s_{\mathtt{q}}$ computed by $f_{\mathtt{q}}(M)$ is always at least as accurate as $WCET^o_{\mathtt{q}}$. Compared to the $WCET^o_{\mathtt{q}}$ estimate, $WCET^s_{\mathtt{q}}$ may take into account the theory $\mathcal{T}$ in addition to the statistics $stats_M$, and $f_{\mathtt{q}}$ also has access to the whole model M.

Proposition 2. The following inequality holds between execution times and their estimates:

$$RT_{\mathtt{q}}(M) \leq f_{\mathtt{q}}(M) \leq WCET^s_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M}) \leq WCET^o_{\mathtt{q}}(stats_M), \tag{7.6}$$

where $\widehat{stats_M}(\mathtt{C}_i) = [stats_M(\mathtt{C}_i), stats_M(\mathtt{C}_i)]$ is the scope corresponding exactly to the model statistics $stats_M$, i.e., is the scope where the lower and upper bound is equal to the number of elements in the model statistics. See proof sketch in Appendix B.

7.6 Hybrid WCET Estimation

We propose a *hybrid* estimation method to leverage both results by DS_Σ and OLS_M. For models satisfying the type scopes $\mathcal{S}$ taken into account when calculating the static estimate, the lowest of the two estimates is taken. For models outside of $\mathcal{S}$, we fall back to the on-line estimates.

Definition 16. The *hybrid WCET estimate of a query* q over a well-formed runtime model with statistics *stats* is formally defined by the function

$$WCET^h_{\mathtt{q}}(stats) = \begin{cases} \min\{WCET^s_{\mathtt{q}}(\mathcal{T}, \mathcal{S}), WCET^o_{\mathtt{q}}(stats)\}, & \text{if } \mathcal{S} \vDash stats, \\ WCET^o_{\mathtt{q}}(stats), & \text{otherwise,} \end{cases} \tag{7.7}$$

where $\mathcal{T}$, $\mathcal{S}$, and the value of $WCET^s_{\mathtt{q}}(\mathcal{T}, \mathcal{S})$ is provided ahead of time.

Computing $WCET^h_{\mathtt{q}}$ only requires the type scope check $\mathcal{S} \vDash stats$ and the computation of the minimum in addition to the evaluation of the $WCET^o_{\mathtt{q}}$ estimate, both of which can be done in constant time. Thus, there is no significant overhead compared to the $WCET^o_{\mathtt{q}}$ estimate. We may avoid checking whether the current state of the runtime model satisfies $\mathcal{T}$, since, by assumption, $\mathcal{T}$ is chosen such that all possible runtime models are well-formed.

7.7 Evaluation

We conducted experiments to answer the following research questions related to the WCET of query programs:

RQ1 How difficult is it to find witness models?
RQ2 How tight are static estimates w.r.t. existing approaches and real execution times?
RQ3 How do static and on-line WCET estimates compare when applied to query programs?
RQ4 How does query program complexity impact the overestimation of computed WCET?

7.7.1 Evaluation Overview and Setup

Queries

To address these research questions, we use graph queries from the domain of the MoDeS3 CPS demonstrator (see Chapter 2). This demonstrator uses high-level runtime monitoring rules captured as graph queries, and showcases synthesized monitoring programs executing these queries over the runtime graph model of the underlying running system. Our experiments focus only on query evaluation, and updates to the runtime model are out of scope for the thesis. Therefore, we ran the query programs on various snapshots of runtime graph models. We evaluated the queries introduced in Example 1.

The calculation of query search plans is out of scope of the thesis, but they were created and optimized based on the typical model statistics of runtime model snapshots in the MoDeS3 system. Although search plans were shown to be highly efficient if they are updated as the properties of the undelying model changes [Var+15], the ones calculated for the realistic model were used throughout the entire evaluation. For example, the search plan presented in Table 6.1 is the one used by the program executing the query Misaligned turnout.

WCET algorithms and WCET tools

To compare the results produced by our WCET estimation approaches DS_{Σ} and DS_{P} with estimates produced by other tools, we used the commercial aiT [FH04] (version 20.10i) and the open-source OTAWA [Bal+10] (version V1.2.0) tools. While aiT comes with inbuilt platform model for XMC4500, the respective OTAWA script was taken from a public repository of an external research group specialized in the analysis of embedded systems[1]. For aiT, we used a *high precision* configuration with pipeline-level analysis and full loop unrolling, as well as a *low precision* configuration with only basic block-level analysis and no unrolling. To incorporate the results of low-level analysis into DS_Σ and DS_P, we extracted the IPET linear equations from the low-level configuration of aiT manually, as no facility was available for automatic export or accessing the high precision system of linear equations directly. We also extracted the IPET linear equations from OTAWA, which have BB execution context information (paths of length two).

Similarly to DS_Σ, we rely on the low-level analysis results from aiT and OTAWA to employ OLS_M and compute WCET^o for query programs. Using this low-level analysis information, we compute parametric WCET formulae. Due to the lack of available tool support, we used a semi-automated WCET formulae computation by applying the algorithm described by Ballabriga et al. [BFL17].

Graph models

In the following, we describe how we obtained a variety of models to assess the impact of models with different characteristics on query evaluation times.

Using the metamodel in the MoDeS3 case study, we generated *witness models* M^* for each monitoring query and for both low-level analyses (aiT, OTAWA) such that the query is estimated to have the longest possible execution time according to the low-level analysis. For all of these models, we used the same model scope inspired by the railway domain: up to 20% of the objects can be Trains and up to 20% of the objects can be Turnouts. The rest of the objects are Segments; we capped the maximum number of objects at 25. The resulting M^* models are syntactically valid and they can represent a realistic railway system thanks to the domain-specific well-formedness constraints. Furthermore, to obtain a *realistic model* M_{real}, we manually captured a detailed runtime model snapshot of MoDeS3 that is similar to the one presented in Figure 3.2b with 25 objects and respects the domain-specific model scope.

To assess the execution times of the query programs on random models, we generated models conforming to the MoDeS3 metamodel with up to a total of 25 objects. Due to the large space of possible graph models, representative sampling from the model space is an open question [JSS13; Sem+20]. Nevertheless, we generated 250 models with the open-source

[1]`https://github.com/uastw-sat/ARMv7t-WCET-Analysis`

EMF random model generator[2] (**Rand**) with up to 5 Turnouts and up to 5 Trains, but none of them represents a railway setting that can occur because they all violated well-formedness constraints due to the completely random construction.

We also generated 250 models with the VIATRA Generator (**VG**) without an optimization objective. These models satisfy all well-formedness and scope constraints we used for generating witness models. However, the state exploration heuristics of the generator may lead to a biased sample.

Hardware setup

We use the Infineon Relax Lite Kit-V1 Board[3] to execute the query programs. This board has an XMC4500 F100-K1024 microcontroller and it is driven by a 120MHz system clock. This microcontroller is considered to be a mature industrial microcontroller and has an ARM Cortex-M4 core. For the present evaluation, the instruction cache on the device is not used as our primary focus is on the impact of domain-specific information about high-level program flow rather than microarchitectural effects.

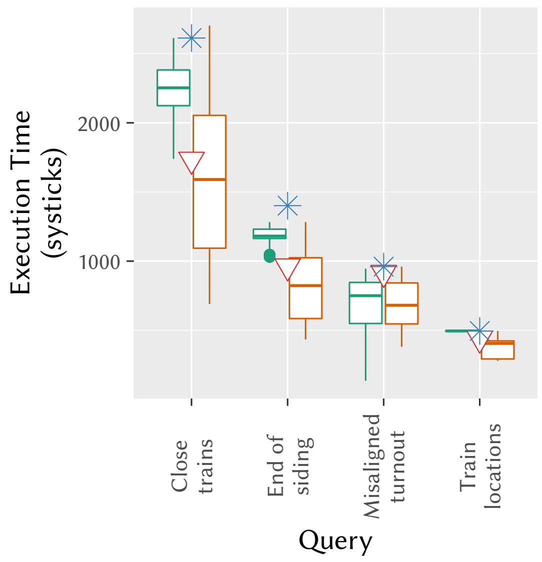
The bare-metal query programs are compiled with GCC compiler for ARM version 7.2.1 with `-O0` and `-g3` flags in debug mode. These programs run on the microcontroller while no other tasks (e.g., interrupts) are running. We rely on the cycle counter feature of the Data Watchpoint and Trace Unit in the device to extract the execution times of each query using a debugger. The embedded code used for the experiments as well as compiler and other configurations are available online[4].

7.7.2 Evaluation Results

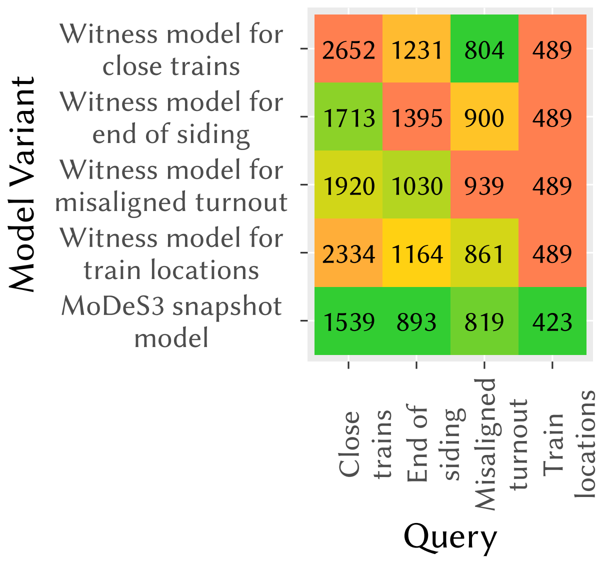
Measured query execution times

We investigate if a witness model for a query can be obtained from simpler graph generation approaches, and we do this by measuring the execution times of queries over various models. Our results are presented in Figure 7.4a. The run times over models by **VG** is captured by the green boxes, while the orange ones show the run times over models by **Rand**. Each query was evaluated on the same two sets of models. Additionally, the respective query execution time

[2]`https://github.com/atlanmod/mondo-atlzoo-benchmark`
[3]`http://www.infineon.com/xmc-dev`
[4]`https://imbur.github.io/cps-query/`



(a) Measured query execution times over consistent models (green box), random models (orange box), witness model (blue star), and realistic model (red triangle)



Model Variant	Close trains	End of siding	Misaligned turnout	Train locations
Witness model for close trains	2652	1231	804	489
Witness model for end of siding	1713	1395	900	489
Witness model for misaligned turnout	1920	1030	939	489
Witness model for train locations	2334	1164	861	489
MoDeS3 snapshot model	1539	893	819	423

(b) Cross-comparison of measured query run times over witness models and a model from the MoDeS3 demonstrator (systicks)

Figure 7.4: Query execution times on fully random models and realistic models

over each witness model M^* is added to these figures for comparison, where M^* is the witness model generated using the objective function built from the low-level analysis results of aiT. Moreover, the run time over the hand-crafted M_{real} model is also presented.

The heatmap in Figure 7.4b presents the obtained query run times for each query over all witness models generated by the solver (e.g., Witness model for close trains represents the model that maximizes the estimated run time of the query Close trains) and the realistic model taken from the MoDeS3 system (MoDeS3 snapshot model). The diagonal in this figure shows the measured execution times over models dedicated to maximize the WCET estimation of a corresponding query.

Findings for RQ1. For consistent models generated with **VG**, queries exhibit the longest observed execution times on their respective witness models. In fact, for two queries, eos and mt, the execution time on the respective M^* is longer than the maximum measured execution time over any other consistent model, which highlights the importance of our witness model generation technique.

Maximum run times over models generated by **Rand** can be both higher and lower than on witness models. For example, query ct takes 2% longer over a random model than over its witness model, but the random model does not represent a realistic railway. On the contrary, query eos takes at least 8% longer to complete on the witness model than on any model generated by **Rand**.

Another important observation is that execution time is highly sensitive to the structure and statistics of the runtime model. For example, Figure 7.4b shows that the query mt evaluated over the witness model for query ct takes the shortest time to complete compared to the execution times over the rest of the witness models and the realistic model. The main reason is that in case of mt, the query search plan (see Table 6.1) in step 3 enforces that the turnout is not switched in the straight direction. However, the query ct only relies on the connectedness of segments via their connectedTo references, and the witness model for ct has most turnouts switched in their straight directions which results in an early cut in the search.

Finally, we notice that query tl, which is a trivial query, exhibits the same execution time for each consistent model generated with **VG** and each witness model. This query simply iterates over Trains, and for all consistent models and witness models there are always 5 Trains.

Computing safe and tight WCET estimates of queries which execute over well-formed models (1) is infeasible by collecting run times over random models, and (2) is feasible by finding witness models by using sophisticated model generation approaches.

Assessment of Static WCET Estimates

Our goal is to compare the computed static WCETs obtained from different tools with our own technique. Table 7.2 shows the WCET esimtates in the DS_{Σ}, aiT, and OTAWA columns for the 4 queries along with measured execution time (expressed in systicks) over the respective witness model.

Findings for RQ2. In the case of ct, our WCET estimation approach produces estimates 14% tighter than the one by aiT (low precision analysis). It is also important to point out that even without context-sensitive BB timings, our low precision approach provides only 3% higher estimates than aiT's high precision mode, which indicates that it is able to automatically identify infeasible paths in the program based on high-level domain-specific information. For OTAWA, improvements of the WCET estimate are achieved in two cases: ct has a 23%, while

Table 7.2: Query code complexity, measured execution time, and WCET estimates in systicks

		Exec. time	$\mathbf{DS}_{\Sigma}$		aiT			$\mathbf{OLS}_{M}$	
Query	CC	over M^{*}	**w/aiT**	**w/OTAWA**	low pr.	high pr.	OTAWA	**w/aiT**	**w/OTAWA**
Close trains	7	2652	**3133**	**3430**	3563	3038	4210	**3563**	**4280**
End of siding	6	1395	**1757**	**1820**	1757	1477	1860	**1757**	**1880**
Misaligned t.	5	939	**1097**	**1370**	1097	987	1370	**1097**	**1390**
Train locations	3	489	**592**	**695**	592	507	695	**592**	**715**

eos has a 2% tighter estimate. For the rest of the queries, the analysis yields the same results as aiT low precision mode or OTAWA.

Conceptually, it would be possible to formulate more precise DS_{Σ} estimates by incorporating low-level analysis results from the high precision mode of aiT as shown in Section 7.4, but such equations are not currently accessible in aiT.

WCET estimates with DS_{Σ} are at least as tight as those obtained by low-level IPET analysis in our experiments. Thus, domain-specific analysis can improve WCET estimates while simultaneously synthesizing witness models to study query program behavior.

Comparison of Static and On-Line WCET estimates

Similarly to DS_{Σ}, we computed one WCET estimate by using OLS_{M} for each tool we used to obtain low-level WCET analysis results. Furthermore, we instantiate all 4 formulae for the realistic model M_{real} as well and we get the following results:

- $\mathit{WCET}^{o}_{\mathtt{ct}}(M_{\mathrm{real}})$ is 2869 with aiT and 3441 with OTAWA timings
- $\mathit{WCET}^{o}_{\mathtt{eos}}(M_{\mathrm{real}})$ is 1424 with aiT and 1521 with OTAWA timings
- $\mathit{WCET}^{o}_{\mathtt{mt}}(M_{\mathrm{real}})$ is 1097 with aiT and 1390 with OTAWA timings
- $\mathit{WCET}^{o}_{\mathtt{tl}}(M_{\mathrm{real}})$ is 493 with aiT and 589 with OTAWA timings

Findings for RQ3. Based on the results presented in Table 7.2, the parametric (online) WCET formulae of monitor executions provide slightly higher estimates for each query investigated on witness models when compared to (static) IPET estimates in case of OTAWA, and provides the same results as aiT low precision analysis mode. However, the rapidly recomputable formula provides 15%–20% tighter estimates in three out of the four cases over the MoDeS3 snapshot realistic model. In case of mt, the estimate is the same as for the witness model of mt. The reason behind these differences is the runtime model statistics for the

MoDeS3 snapshot model has one train less than the maximum number allowed by the model scope, which is a key factor in the formulae of ct, eos, and tl, while the formula for mt does not depend on this number.

Static estimates complemented with on-line WCET estimates based on runtime model statistics provide safe and tight execution time bounds at runtime.

Impact of query program complexity

With RQ4, we look at the impact of query complexity on the computed WCET bounds, so that we can give recommendations on where our approach offers the greatest benefits. The execution times of queries over M^* in Table 7.2 provide a lower bound to the actual WCET (i.e., the longest possible execution time of the program over inputs which represent well-formed models in the model scope), while the CC columns shows query cyclomatic complexity. Since the actual WCET of the program is unknown (but it must lay between the measured execution time and the WCET estimates produced by the analyses), we use the measured execution time over witness models as the baseline when discussing overestimation in WCET estimates.

Findings for RQ4. The biggest visible advantage of DS_Σ is in the case of the most complex query ct: the overestimation is 18% with BB timings from aiT, while the aiT low precision analysis computes a 34% higher value than the measured execution time. In other cases, it produces the same result as aiT, with overestimates being between 16% (query mt) and 26% (query eos). We come to the same conclusion using BB timings from OTAWA, although these timings are slightly more conservative. The high precision analysis available in aiT is able to leverage the microarchitectural properties and thus provide the most precise estimates with the overestimation being 14% (observed for query ct). The overestimation increases with CC of the query code only in the case of high precision aiT analysis.

In general, DS_Σ computes a safe WCET bound and additionally provides a witness model. Moreover, it is able to discover additional infeasible paths for the WCET estimate for the most complex query, thus providing a tighter estimate.

For complex queries, the precise count of BB executions by DS_Σ gives tighter estimates than the IPET method which heavily relies on manually provided flow facts. Overall, this shows the strength of the proposed graph solver-based WCET estimation method.

7.7.3 Threats to Validity

Construct validity. In the assessment of the tightness of WCET, we compare the longest *measured* run time $RT_{\mathrm{q}}(M^*)$ and *estimated static* $f_{\mathrm{q}}(M^*)$ and *estimated on-line* $WCET^{o}_{\mathrm{q}}(stats_{M^*})$ run times. As mentioned in Section 7.4, this method can conservatively underestimate the tightness of the computed WCET in case $M^* \neq M_{worst}$, i.e., when the longest run time RT_{q} is exhibited on M_{worst}, and thus $RT_{\mathrm{q}}(M^*) \leq RT_{\mathrm{q}}(M_{worst}) \leq f_{\mathrm{q}}(M^*)$ and $RT_{\mathrm{q}}(M^*) \leq RT_{\mathrm{q}}(M_{worst}) \leq WCET^{o}_{\mathrm{q}}(stats_{M^*})$. Ideally, this assessment should use $RT_{\mathrm{q}}(M_{worst})$, however, this model is generally unknown.

Internal validity. Computed WCET values presented in this section are reasonable with respect to the measured longest execution times. However, the platform model of the microprocessor may not be completely accurate (especially, in the case of the model used with OTAWA), which can result in imprecision of computed WCET. The evaluation platform only ran the monitoring programs; no other tasks were running on the same device which excludes other external influencing factors. Finally, the algorithm for obtaining the parametric WCET formula [BFL17] supports contextual information for refining BB timings (e.g., to incorporate the effect of processor pipeline), but our formulae did not use this. For this reason, the formulae we used might provide less tight estimates, but the computed WCET bounds are still safe.

External validity. In addition to the hardware-specific considerations (i.e., replicating the presented evaluation using other hardware platforms), evaluation of the WCET estimation techniques along additional case studies with query-based runtime monitors from different domains could further improve the confidence in the evaluation results. Moreover, we assume that the presented results hold for larger models as well, however, the currently used model sizes are capped by the scalability limits of the current version of the underlying graph solver.

7.8 Summary

This chapter investigated worst-case execution time analysis methods for query-based runtime monitor programs to enable their use in hard real-time settings. To that end, we have provided a brief overview of the state-of-the-art, and we have combined low-level WCET analysis results with a cutting edge graph solver to provide both tight execution time bounds and input models where the longest execution is assumed. Furthermore, to support cases where a result from the graph solver-based approach is not available, we adopted a parametric WCET estimation method and proposed to parameterize it with condensed runtime model statistics to compute WCET. We compared our approaches with two existing analysis tools, aiT and OTAWA, and showed where we can tighten the results of the timing analysis. In particular, this chapter showed our results for the third contribution group (Co3.1–Co3.4).

Publications related to this chapter. The timing analysis of graph query-based runtime monitors was first included in the conference paper [*c*3]. This work is my contribution. The static WCET analysis method is submitted to a journal and is now under review [*j*1]. The concept of witness models and using them as means to estimate WCET is my contribution, whereas the specificities of how to find witness models is the contribution of Kristóf Marussy. Brett Meyer was helping the work as advisor and provided continuous feedback.

Part III

Distributed Runtime Graph Models and Queries

Chapter 8

Distributed Runtime Models

We propose a *distributed runtime graph model* in this chapter to capture the operational state and the context information of a smart CPS. While Chapter 5 focuses on suitable low-level data structures for runtime models in resource-constrained environments, the objective of the present chapter is to provide a runtime graph model management protocol for distributed systems. Our key assumption is that creating a centralized global view about the entire system is not realistic due to communication latency and the limited computing capacity available in the platform components or because of privacy reasons, data collected by the devices should not be directly sent to a cloud platform. Instead, we assume runtime model fragments are maintained by computing units of the platform to capture their local knowledge base, and only necessary changes are communicated between them.

Our work addresses decentralized mixed synchronous systems [Tei+94] where participants (1) communicate model updates to other participants in the first part of a time-triggered execution loop [KG93] (*update cycle*) and (2) then evaluate queries over a consistent snapshot of the system (*query cycle*). Later in Chapter 9, we focus on the query cycle while this chapter provides a detailed description of the model update cycle.

This chapter is structured as follows. Section 8.1 shows a solution to create self-descriptive models by incorporating execution platform information into a metamodel, and introduces the concept of distributed runtime graph models. Then, Section 8.2 discusses the details of our model management protocol that is designed to ensure consistency, i.e., it ensures that any given point in time, no computing units can provide contradictory information about the contents of the runtime model. Finally, in Section 8.4 we evaluate the performance of our model management protocol on both real and virtual CPS platforms and highlight the key findings.

8.1 Distributed Runtime Models

We now introduce the changes necessary to a metamodel to allow capturing information about the ownership (i.e., which computing unit has created it) of model objects, and we extend our formal notation to support this aspect.

8.1.1 Metamodel Features for Distributed Runtime Models

As discussed earlier in Section 3.1.1, a metamodel captures the concepts of the domain with their attributes, as well as possible relationships between them. In a distributed setting, we assume that this runtime model is self-descriptive in the sense that it contains information about the computation platform and the allocation of services to platform elements, which is a key enabler for self-adaptive systems [Che+11; VG14]. For this reason, we propose specific concepts that can be added to a metamodel to allow capturing this allocation aspect.

The changes are as follows: we introduce a new class named Participant with a unique hostId attribute that is used to represent computing units in the platform. The name of this class comes from the DDS standard, where each communicating entity is a participant. Furthermore, each class in the domain should realize a new DomainElement interface. This interface is introduced to avoid the addition of multiple references expressing the ownership between the instance of a specific class in the domain and a participant. Instead, there is a hostedObjects reference from Participant to DomainElement which is used to keep track of which participant hosts which objects. Finally, lower bounds of reference multiplicities in the metamodel are set to 0, while we allow the upper bounds to be 1 or unbounded. The example below illustrates the described additions.

> **Example 20.** Figure 8.1 shows all the elements of the initial metamodel for the MoDeS3 demonstrator presented in Figure 3.2a along with the proposed additions. Orange color shows the additions and changes to the metamodel.

We also consider bidirectional associations by adopting the concept of *opposite references* from EMF metamodels (*eOpposites*) where each reference type may have an opposite reference type and vice versa (such as location and occupiedBy in Figure 8.1). EMF maintains such pairs of opposite references consistently in case of (non-distributed) instance models, i.e., if a reference is created or deleted, its opposite reference is created or deleted automatically. However, maintaining such pairs of references is more complicated in a distributed setting,

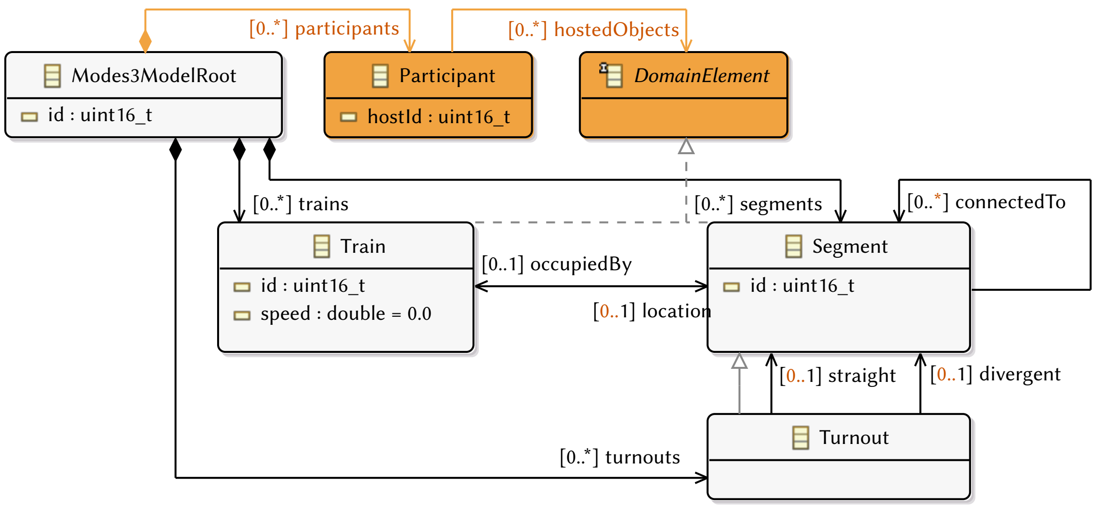


Figure 8.1: Extended MoDeS3 metamodel with platform information

as the objects that encapsulate the individual references might be hosted by two different participants, and thus it is necessary to communicate the changes. To provide a solution for this challenge, we propose a protocol in Section 8.2.2.

8.1.2 Distributed Runtime Graph Models

While a (regular) runtime model serves as a centralized knowledge base, this is not a realistic assumption in a distributed setting where data is collected and stored locally by participants in a decentralized manner. In our distributed runtime model, each participant only has up-to-date but incomplete knowledge about the distributed system. Moreover, we assume that each model object is exclusively managed by a single participant, referred to as the *host* (i.e., *owner*) of that element, which serves as the *single source of truth*. This way, each participant can make calculations based on its own view of the system (e.g., local evaluation of a query that reads only the part of the model hosted on the participant is possible), and it is able to modify the mutable properties of its hosted model elements. Any access, however, to an object hosted by a different participant requires communication to respect the single source of truth principle.

To extend the formal treatment of models to distributed runtime models, we mark which participant is responsible for storing the value of a particular predicate in its local knowledge base. For a predicate φ with parameters $v_1, \ldots, v_n$, $[\![\varphi(v_1, \ldots, v_n)]\!]^{M_{distr}}@p$ denotes its value over the distributed runtime model M_{distr} stored by host p.

Example 21. Figure 8.2 shows a snapshot of the distributed runtime model M_{distr} for the MoDeS3 system. Participants deployed to different physical computing units manage different (disjoint) parts of the system. The diagram in Figure 8.2 presents the three participants (P_1–P_3, depicted by dashed rectangles) deployed to three computing units, the domain elements (s_0–s_6, tu_0, tu_1, tr_0, tr_1, and tr_2), as well as the links between them. Each participant *hosts* model elements contained within them in the figure, e.g., P_2 is responsible for storing attributes and outgoing references of objects s_0, s_6, and tr_1. Formally, the following expressions all evaluate to 1 (`true`):

$$[\![\texttt{Train}(tr_1)]\!]^{M_{distr}}@P_2, \quad [\![\texttt{Segment}(s_0)]\!]^{M_{distr}}@P_2, \quad [\![\texttt{Segment}(s_6)]\!]^{M_{distr}}@P_2,$$
$$[\![\texttt{Location}(tr_1, s_0)]\!]^{M_{distr}}@P_2, \quad [\![\texttt{OccupiedBy}(s_0, tr_1)]\!]^{M_{distr}}@P_2,$$
$$[\![\texttt{ConnectedTo}(s_6, s_0)]\!]^{M_{distr}}@P_2, \quad [\![\texttt{ConnectedTo}(s_0, s_6)]\!]^{M_{distr}}@P_2,$$
$$[\![\texttt{ConnectedTo}(s_6, s_5)]\!]^{M_{distr}}@P_2, \quad \text{and} \quad [\![\texttt{ConnectedTo}(s_0, s_1)]\!]^{M_{distr}}@P_2.$$

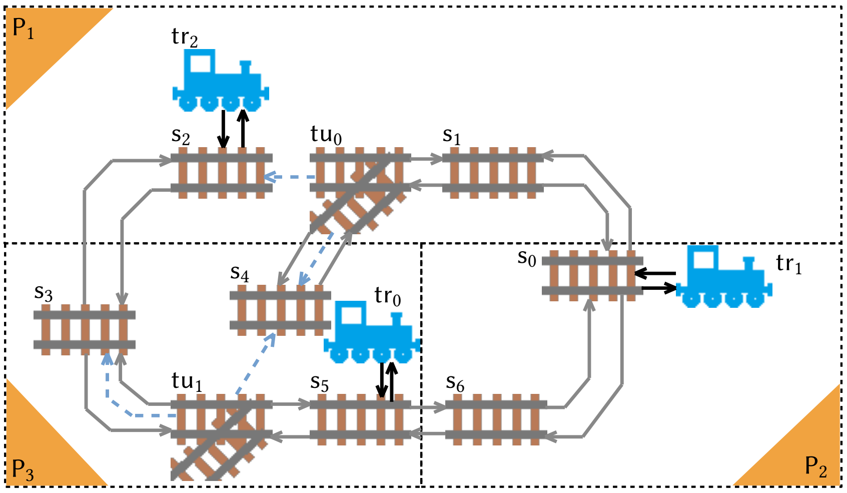


Figure 8.2: Distributed runtime model snapshot of MoDeS3

8.1.3 Distributed Model Update Operations

We assume that the following model manipulation operations are available for a distributed runtime model (see also Section 3.1.2):

- **Object operations**: In runtime models, objects can be *created* and *deleted*. Object update operations are implemented by broadcast messages.

- **Attribute operations**: Attribute values can be *updated* locally (i.e., no communication is needed) in a distributed runtime model since the values of attributes are always stored together with the object itself by the host participant.
- **Reference operations**: A link can be *added* or *deleted* between two objects. If both ends of a link are hosted by the same participant then such a reference update is a local operation (similar to an update of an attribute), otherwise it involves communication with other participants in a peer to peer manner.

8.2 A Model Management Protocol for Distributed Runtime Models

Next, a time-triggered protocol for distributed runtime model management operating over a reliable communication middleware of the DDS standard is proposed. This protocol is designed for CPSs where certain level of consistency is required, but can be adopted by other platforms that can run DDS-compliant software (e.g., smartphone applications). In any case, our discussion covers the design of the protocol to support distributed graph models, while privacy concerns are out of scope for the current thesis.

8.2.1 Overview of Assumptions

Below we summarize our main assumptions and considerations.

Assumptions on sensing. We assume that each participant can detect relevant information about its own model elements by local sensing, which triggers model updates to its local knowledge base (together with a timestamp). The lifecycle of any model element can be attached to sensor readings, i.e., a creation and deletion of a train object in the runtime model depends on whether a particular sensor is able to detect the train in the real system. Such sensor readings can be periodic (e.g., once in every 10 ms) or event-driven (e.g., when a new train is detected). Raw sensor readings are buffered until the next model update cycle, while the runtime model is updated in accordance with our protocol.

Assumptions on model updates. Conceptually, a participant may communicate relevant model changes to other participants either asynchronously or periodically. However, all model update requests are registered with a timestamp and buffered to be processed later in a time-triggered way by our distributed model management protocol. The real processing order of model update messages will not be time-ordered, but our protocol tolerates lost/delayed messages and handles common semantic corner cases (see later in Section 8.3.1) by the end of the model update cycle. As such, distributed graph query evaluation can return results from a consistent runtime model of the system during the query cycle.

Assumptions on timeliness. We assume approximate synchrony [Des+15] between the clocks of individual computing units, thus each participant can precisely detect the beginning and the end of each cycle w.r.t each other. In other terms, the discrepancy between the clocks of participants is negligible.

The time-triggered nature of our protocol provides an upper bound defined by the cycle time of model updates (t_u) and cycle time of query evaluation (t_q). Thus, if no messages arrive late, our runtime monitors will detect a relevant situation in at most $2 \cdot (t_u + t_q)$. However, a detailed evaluation of timeliness guarantees is out of scope for the thesis and it is left for future work.

Assumptions on communication middleware. In order to periodically communicate model changes between participants, our distributed model update protocol relies upon DDS, a standard reliable communication middleware to provide several important QoS guarantees.

1. Timely and reliable message delivery of model update messages is ensured by the DDS middleware.
2. If there is a violation of QoS guarantees, DDS notifies participants to allow them to recover from faults as part of the model update and query protocol. As such, the sender of the message will be aware of such communication fault.
3. The synchrony of physical clocks of participants is enforced by a clock synchronization protocol [Mar+04; SBK06], thus each participant receives messages with a timestamp denoting when the update action was initiated.
4. Participants can save update messages to a preallocated buffer with potentially limited size. This way, participants running under resource constraints will not be flooded by an excessive number of messages sent over the network, and they are able to selectively filter messages they want to keep based on their specific needs and preferences.

Assumptions on fault-tolerance guarantees. While DDS guarantees reliable message delivery (i.e., a sent message will eventually arrive), it may not enforce that messages would arrive within the timeframe of their phase. As such, our fault model considers messages that arrive outside the timeframe of their designated phase to be lost. Since DDS provides many QoS guarantees for participants, our fault model used in the paper is restricted to message loss or late arrival of a message.

Assumptions on properties of computing units. We assume that the computing units are capable of running a program containing the implementation of the DDS standard. Based on our initial assessment to be presented in Figure 8.4.2, this yields that the computing units need to have at least 15 MB heap available to successfully initialize the middleware in our case. Furthermore, we assume the devices can run the required TCP or UDP network stack.

8.2.2 A Multi-Phase Model Update Protocol

Time-triggered execution cycle. Our runtime monitoring approach is driven by a time-triggered execution loop which can be divided into five major conceptual phases. The first four phases constitute the model update protocol (discussed in this section) with (1) object create, (2) object delete, (3) link update request, (4) link update reply. The update phase is followed by a (5) query phase (discussed in Chapter 9).

Our model update protocol will be defined by complex statecharts containing multiple parallel regions. In this paper, we used Yakindu Statechart Tools[1] for the specification and simulation of the protocol.

Figure 8.3 shows a statechart model describing this execution cycle. Transitions are triggered by events coming from a master clock that is available to all participants, which is implemented using high precision clock synchronization across platform components.

Sensing. A statechart describing the behavior of the sensing services capable of detecting objects is displayed in Figure 8.4. The transitions are triggered by the events sensing.appear and sensing.disappear assumed to be raised by changes in the operational context of the system. When those transitions fire, the sensing.objectAppeared and sensing.objectDisappeared flags are set/cleared according to the events, allowing local buffering of events. These flags are cleared at the beginning of each object create cycle. Figure 8.4 only depicts the sensing model for objects, but an identical model employing a different pair of flags can be used to model the appearance/disappearance of links between two selected objects.

[1]https://www.itemis.com/en/yakindu-tools

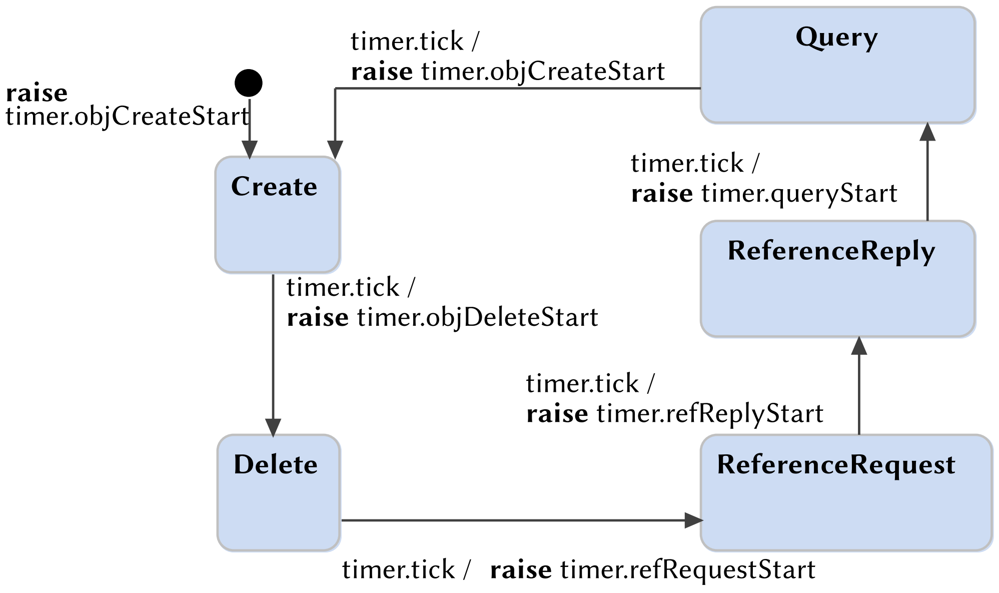

Query
timer.tick / raise timer.objCreateStart
raise timer.objCreateStart
Create
timer.tick / raise timer.queryStart
ReferenceReply
timer.tick / raise timer.objDeleteStart
timer.tick / raise timer.refReplyStart
Delete
ReferenceRequest
timer.tick / raise timer.refRequestStart

Figure 8.3: Runtime phases of model updates and queries (statechart *phases*)

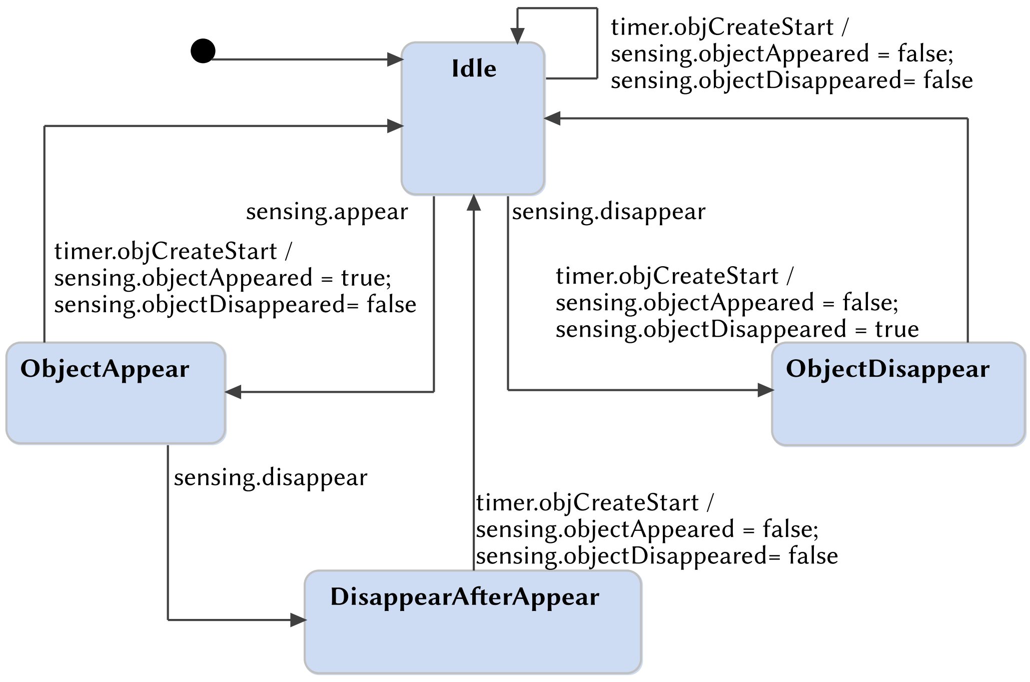

Idle
timer.objCreateStart / sensing.objectAppeared = false; sensing.objectDisappeared= false
sensing.appear
sensing.disappear
timer.objCreateStart / sensing.objectAppeared = true; sensing.objectDisappeared= false
timer.objCreateStart / sensing.objectAppeared = false; sensing.objectDisappeared = true
ObjectAppear
ObjectDisappear
sensing.disappear
timer.objCreateStart / sensing.objectAppeared = false; sensing.objectDisappeared= false
DisappearAfterAppear

Figure 8.4: High-level model of object sensing services local to a participant (statechart *sensing*)

Overview of messaging for model updates. In our distributed model update protocol, *object creation and deletion are communicated as broadcast messages* so that participants can register the existence of all objects. Such broadcast messages allow each participant to add or remove links to any model object, as well as to query object attribute values or links, even if

Table 8.1: Summary of actions when receiving object update messages

	Object update message	
Condition	$[\![\mathtt{C}(obj)]\!]^{M_{distr}}@p = 1$ (*obj* created at *p*)	$[\![\mathtt{C}(obj)]\!]^{M_{distr}}@p = 0$ (*obj* deleted at *p*)
obj is unknown	create proxy object *obj*	create proxy for *obj* and mark it as deleted
obj is present locally	no-op	mark proxy for *obj* as deleted

an object is not hosted by the participant. On the other hand, *messages for link addition and removal are sent in a peer to peer manner.* The precise protocol of sending and receiving such message will be defined by a series of statecharts, which control the model update behavior individually for each model element.

8.2.3 Object Create Phase

The first phase of the model update cycle addresses object creation. A participant that creates a new object must send a *broadcast message* with the identifier o_{create}, the type C, and its participant identifier p_{host}. Formally, the message has $[\![\mathtt{C}(o_{create})]\!]^{M_{distr}}@p_{host} = 1$ as content. It is necessary to notify other participants about the creation of a new object in order to allow them to create links pointing to the object (i.e., as a target end of an edge). Recipient participants will create a *proxy object* locally that represents the remote object in the model by having the same type C, but stores the object identifier o_{create} and the host participant identifier p_{host} as its only attributes. The middle column of Table 8.1 summarizes the actions a participant takes upon receiving an object create message.

The statechart in Figure 8.5 specifies the lifecycle of an object while a parallel region (depicted separately in Figure 8.6) shows the ownership of the object.

Statechart of object creation. The initial state for an object is NoObject (Figure 8.5) that represents when the object does not exist in the model, while its ownership is initially None (Figure 8.6). From this initial state, creation is triggered when the participant either (1) receives a data.create message or when (2) sensing signals sensing.objectAppeared at the beginning of the object creation phase triggered by a timer.objCreateStart event.

- In the first case, the corresponding (broadcast) message arrives in the object create phase. A proxy object is registered and enters the Created state, while ownership is set to Proxy after raising event ownership.releaseObject.

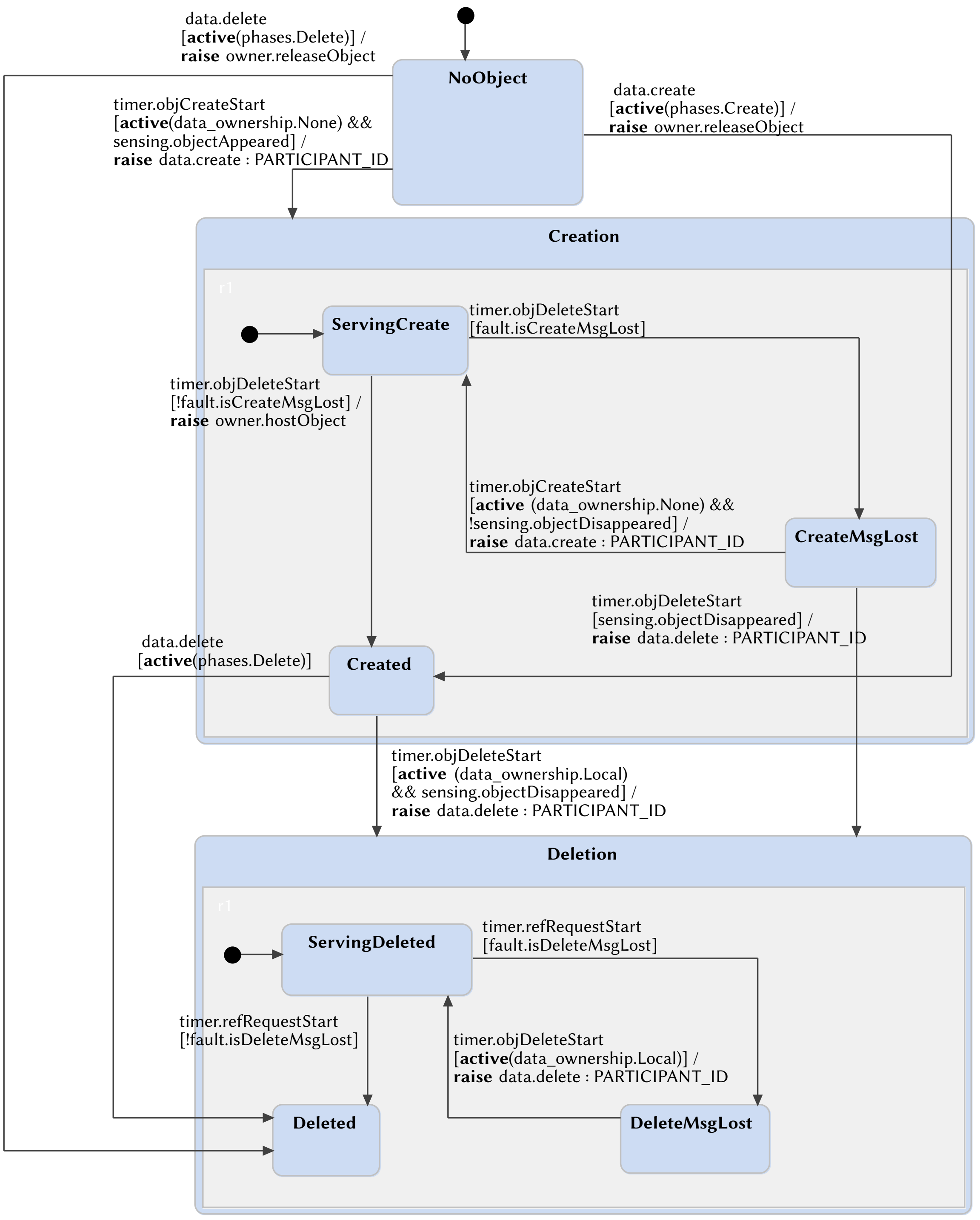

data.delete
[active(phases.Delete)] /
raise owner.releaseObject
NoObject
data.create
[active(phases.Create)] /
raise owner.releaseObject
timer.objCreateStart
[active(data_ownership.None) &&
sensing.objectAppeared] /
raise data.create : PARTICIPANT_ID
Creation
ServingCreate
timer.objDeleteStart
[fault.isCreateMsgLost]
timer.objDeleteStart
[!fault.isCreateMsgLost] /
raise owner.hostObject
timer.objCreateStart
[active (data_ownership.None) &&
!sensing.objectDisappeared] /
raise data.create : PARTICIPANT_ID
CreateMsgLost
timer.objDeleteStart
[sensing.objectDisappeared] /
raise data.delete : PARTICIPANT_ID
data.delete
[active(phases.Delete)]
Created
timer.objDeleteStart
[active (data_ownership.Local)
&& sensing.objectDisappeared] /
raise data.delete : PARTICIPANT_ID
Deletion
ServingDeleted
timer.refRequestStart
[fault.isDeleteMsgLost]
timer.refRequestStart
[!fault.isDeleteMsgLost]
timer.objDeleteStart
[active(data_ownership.Local)] /
raise data.delete : PARTICIPANT_ID
Deleted
DeleteMsgLost

Figure 8.5: States of an object in our model update protocol (statechart *object_consistency*)

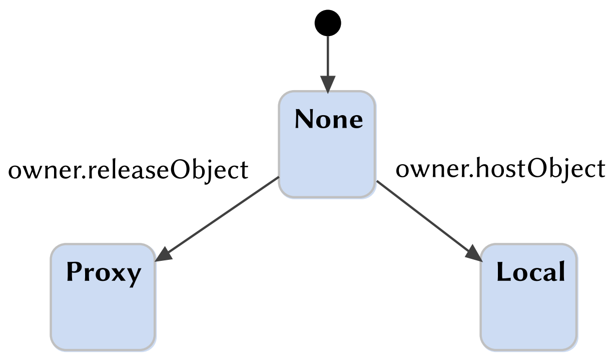


Figure 8.6: Object ownership states (statechart *data_ownership*)

- In the second case, local sensing services indicate that an object needs to be created locally by setting the sensing.objectAppeared flag and this flag is read at the start of the object create phase indicated by the timer.objCreateStart event. Then, a broadcast message is sent on the creation of a new object and the object is moved to the ServingCreate state. By the time the next phase starts (which is indicated by a timer.objDeleteStart event raised by the master clock), if no lost messages are reported, i.e., the broadcast message was successfully delivered to all other participants, the object enters the Created state, while the ownership is set to Local.

8.2.4 Object Delete Phase

In the second phase of model updates, objects of the runtime model can be deleted. The phase is similar to object creation: the identifier o_{delete}, the type $\texttt{C}$, and the host p_{host} of the deleted object is sent in a *broadcast message*. Formally, $[\![\texttt{C}(o_{delete})]\!]^{M_{distr}} @ p_{host} = 0$ is sent, which is also saved in the local knowledge base. It is necessary to notify other participants about the deletion of an existing model element to allow them to remove potential dangling edges originally pointing to the deleted object. Only the host participant of the object can initiate the deletion of the corresponding object, otherwise the object deletion message is ignored. Deleting an object is irreversible, i.e., once an object is deleted, it cannot be reverted. The last column of Table 8.1 summarizes the actions to be taken upon receiving a *delete object* message.

Statechart of object deletion. Deletion of an object that is in the Created state (Figure 8.5) depends on the ownership of the object (Figure 8.6).

- A hosted object (i.e., ownership.Local is active) is deleted by first entering the ServingDeleted state and sending a broadcast message to all participants in the system about the deletion. Then, if all participants have been successfully notified about the deletion of the object (i.e., fault.isDeleteMsgLost is `false`) at the end of the object delete phase, the object goes to Deleted state.

- A proxy object (i.e., ownership.Proxy is active, thus owned by another participant) is immediately brought to Deleted state upon receiving a data.delete event in the object delete phase.

Furthermore, if a participant receives a data.delete for an object that is not known, i.e., while the object is in the initial state NoObject, it transitions to the Deleted state to ensure that participants have a synchronized knowledge on the state of model objects. It must be noted that an object in the Deleted state is never included in any match during the query phase.

8.2.5 Link Update Request Phase

In this third phase of the model update protocol, link additions and removals are initiated (in arbitrary order) between objects. Unlike the case of object update, where broadcast messages are used to notify each participant, link updates are communicated in a peer to peer manner between participants that are the hosts of the two objects at the ends of the link being updated. Other participants, as they are not storing any information about links between objects they are not hosting, are not notified.

Link creation. Adding a link from object o_{src} to o_{trg} is done without sending any messages if either (i) both objects are hosted by the same participant p_{host} or (ii) their hosts are different, but the structural consistency checks can be done locally by the host of o_{src}.

Otherwise, link addition from object o_{src} to o_{trg} is initiated by the host of o_{src} (denoted as p_{src}). Formally, a message is sent with $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}}@p_{src} = 1$ as content. To maintain a consistent model, the local knowledge base keeps the $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}}@p_{src} = 0$ entry until receiving an acknowledgement message from the host of the target object containing $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}} @p_{trg} = 1$ in the subsequent link update reply phase. However, if the link cannot be added for some reason, e.g., a multiplicity constraint would be violated, the reply from p_{trg} will be $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}} @p_{trg} = 0$. From this information the host of o_{src} will also deduce that the link cannot be set, thus a consistent truth value is maintained by both parties.

Statechart of link creation. Figure 8.7 shows the possible states of a link, while a participant's role is modeled separately in Figure 8.8. The respective initial states are NoLink and

Server. Similarly to object creation, adding a link can start at a participant after (1) receiving a data.addRequest while in the reference request phase or when (2) the local sensing services signal sensing.linkAppeared at the beginning of the reference request phase indicated by the timer.refRequestStart event.

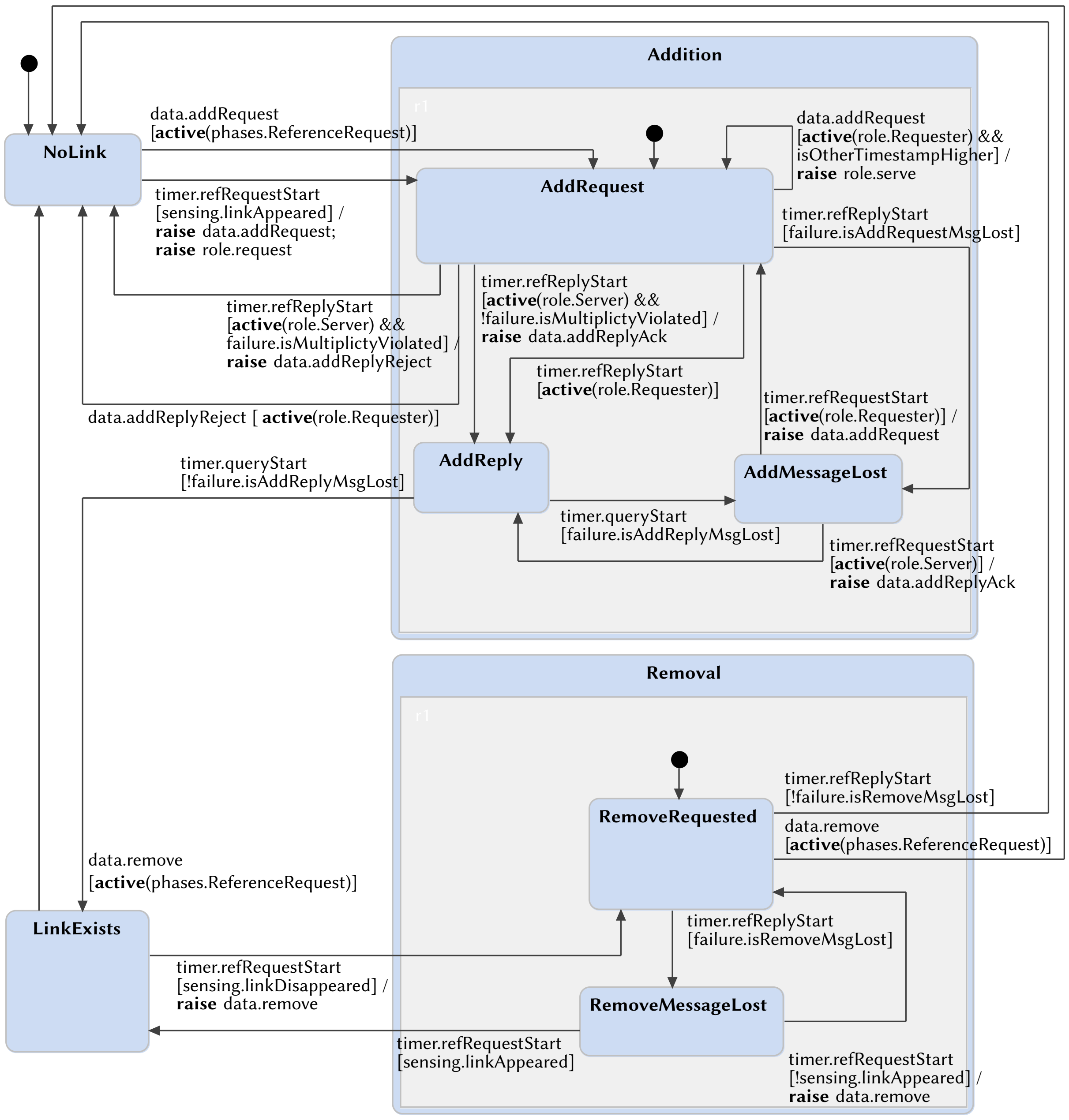

Addition
NoLink
data.addRequest [active(phases.ReferenceRequest)]
timer.refRequestStart [sensing.linkAppeared] / raise data.addRequest; raise role.request
AddRequest
data.addRequest [active(role.Requester) && isOtherTimestampHigher] / raise role.serve
timer.refReplyStart [failure.isAddRequestMsgLost]
timer.refReplyStart [active(role.Server) && failure.isMultiplictyViolated] / raise data.addReplyReject
timer.refReplyStart [active(role.Server) && !failure.isMultiplictyViolated] / raise data.addReplyAck
timer.refReplyStart [active(role.Requester)]
timer.refRequestStart [active(role.Requester)] / raise data.addRequest
data.addReplyReject [active(role.Requester)]
timer.queryStart [!failure.isAddReplyMsgLost]
AddReply
AddMessageLost
timer.queryStart [failure.isAddReplyMsgLost]
timer.refRequestStart [active(role.Server)] / raise data.addReplyAck
Removal
RemoveRequested
timer.refReplyStart [!failure.isRemoveMsgLost]
data.remove [active(phases.ReferenceRequest)]
data.remove [active(phases.ReferenceRequest)]
LinkExists
timer.refReplyStart [failure.isRemoveMsgLost]
timer.refRequestStart [sensing.linkDisappeared] / raise data.remove
RemoveMessageLost
timer.refRequestStart [sensing.linkAppeared]
timer.refRequestStart [!sensing.linkAppeared] / raise data.remove

Figure 8.7: States of a reference in our model update protocol (statechart *link_consistency*)

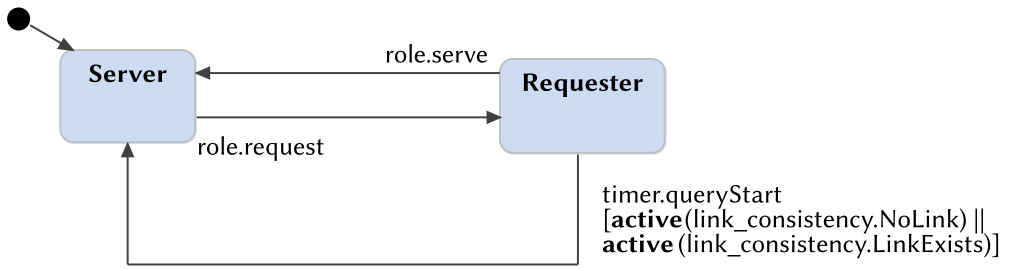

Server
Requester
role.serve
role.request
timer.queryStart
[active(link_consistency.NoLink) ||
active (link_consistency.LinkExists)]

Figure 8.8: Participant roles during reference update (statechart *role*)

- In case (1), the participant is serving the reference add request, so that it stays in the Server state, while the link transitions from NoLink to AddRequest. Then, at the start of the reference reply phase, it sends back an acknowledgement to the requester and enters the AddReply state. Upon successful delivery of the reply message, the reference is created and it enters the LinkExist state. While in AddRequest, if the new reference would violate a multiplicity constraint, a reject is sent back to the requester and the link is not created, its next state is NoLink.
- In case (2), the participant takes the Requester role and the reference moves to the AddRequest state. Once the request is successfully delivered, the reference's state changes to AddReply where it is waiting for the reply message. Once acknowledged, the link is added and it enters the LinkExist state.

Link removal. The removal of a directed link leading from object o_{src} to o_{trg} is similarly done without sending any messages if either (i) both objects are hosted by the same participant p_{host} or (ii) they are hosted by different participants, but structural consistency can be ensured locally by the host of o_{src}.

Otherwise, removing a link can be initiated by participant p_{src}, the host of the source object o_{src} by sending a request to participant p_{trg} hosting the target object o_{trg}. Formally, to initiate removing a reference of type R, the content of the messages is $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}} @p_{src} = 0$. Reference removal requests will not have corresponding reply messages, because we assumed lower multiplicity bounds for references to be 0, thus such requests would always be acknowledged.

Statechart of link removal. Table 8.2 briefly summarizes the actions to be taken upon receiving reference update request messages, while the right part of Figure 8.7 shows the states related to the removal of an existing reference. Similarly to deleting an object, the removal of a reference that is in the state LinkExist can be triggered in two ways: either receiving a message

Table 8.2: Summary of actions for link update request messages

	Reference update request message	
Condition	$[\![\mathtt{R}(src, trg)]\!]^{M_{distr}}@p = 1$ (p requests addition)	$[\![\mathtt{R}(src, trg)]\!]^{M_{distr}}@p = 0$ (p requests removal)
link exists	no-op	delete opposite link
link does not exist	if multiplicity constraints hold, add opposite and send acknowledgement, otherwise send reject to request	no-op

in the reference request phase initiating the removal, or via the local sensors. In the former case there is no extra condition, the link simply goes to state NoLink. If the removal is triggered by reading sensing.linkDisappeared = `true` at the beginning of the reference request phase, the reference enters the RemoveRequested state. Once the target participant is delivered the remove message, the link goes to NoLink.

8.2.6 Link Update Reply Phase

The only special attention is needed for handling the addition of inverse links (which need to be updated simultaneously) with [0..1] multiplicities due to the potential race condition between participants. In such a case, the target object of a link update request may reject the corresponding add request to ensure structural consistency, i.e., to respect the upper multiplicity bound. Thus, when a link with an opposite is to be added, the host of the target object needs to acknowledge the operation for the host of the source object in a subsequent *link update reply phase.*

- In case of success, both parties are consistently notified about the change by replying $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}} @p_{trg} = 1$, thus the opposite references can be set automatically at both participants without sending extra messages over the network.
- If a structural inconsistency is detected at the target object, the reference add request is rejected by sending $[\![\mathtt{R}(o_{src}, o_{trg})]\!]^{M_{distr}}@p_{trg} = 0$.

8.3 Fault Tolerance and Consistency

8.3.1 Fault Tolerance to Handle Message Loss

As model update messages sent by a participant might get delayed, thus a message will eventually arrive but possibly after its deadline (outside the respective phase). These cases are always detectable by the sender of the message, and our protocol conceptually handles such latecoming messages as message loss (i.e., the message is lost within the given cycle).

Message loss during object update. Nevertheless, our object update protocol can recover from faults eventually caused by message loss thanks to extra states introduced in Figure 8.5. For *object create*, if at least one message was not delivered while in ServingCreate state, fault.isCreateMsgLost is set to true based on notifications coming from the communication middleware. Then, the object enters the CreateMsgLost state and the broadcast message at the beginning of the next object create phase is repeated. This loop is iterated until eventually everyone is notified about the existence of the object.

Furthermore, the protocol is able to handle cases when a sensor reports that the object under creation should be immediately deleted (the sensing.objectDisappeared flag is set) while recovering from lost creation messages. When this happens, object enters the Deletion composite state and the deletion procedure will begin in the object delete phase.

Likewise, upon deleting an object, the DeleteMsgLost state is entered (from state ServingDelete) if the middleware detects issues with delivering the messages by the end of the object delete cycle. At the beginning of the object delete phase, the object returns to ServingDelete and the deletion broadcast messages are retransmitted. Again, this loop is iterated until eventually each remote participant is notified about the deletion.

Message loss for reference updates. Similarly to object update, our reference update protocol is prepared to tolerate message loss to avoid inconsistencies thanks to the extra states AddMessageLost and RemoveMessageLost introduced for fault tolerance purposes. When reference addition is requested (i.e., while having the role Server), the AddMessageLost is reached if the request message is lost. Then, at the beginning of the next reference request phase, the data.addRequest is resent and the state AddRequest is entered. If a reply message is lost as a server during a reference add, the same AddMessageLost is reached, but the reference in this case will return to the state AddReply, and will retransmit the previously lost answer (either data.addReplyAck or data.addReplyReject).

Tolerating a message loss in case of a reference remove request is a simpler task compared to reference add because a remove request that is sent when transitioning from LinkExist to RemoveRequested does not need to be acknowledged. Once such a message is delivered, the edge can be safely removed but until that the requester is looping between RemoveMessageLost and RemoveRequested states.

8.3.2 Semantic Aspects of Consistency

While providing a formal proof of consistency for our distributed model update protocol is outside the scope of this thesis, we highlight some aspects and corner cases which need to be tackled to establish desirable semantic properties like consistency or termination.

Termination. Our protocol avoids deadlocks (i.e., two participants are mutually waiting for each other) and livelocks (when they are continuously sending messages to each other). Deadlock avoidance is achieved by (1) restricting each cycle to messages of a particular type and (2) using a time-triggered execution which continuously progresses to the next phase regardless of the arrival of messages. Livelocks are avoided by ensuring that a bounded number of messages (requests and replies for each model element) are sent in each phase.

Local consistency w.r.t sensor readings. By local consistency, we mean that durable local events detected by sensors attached to a model element will eventually be reflected in the (local) runtime model of the participant. Since each sensor reading is recorded as a local event with a timestamp, causality of such sensing events (e.g., an object appearance or disappearance is observed by the owner participant) are easily established in the update cycle (e.g., a corresponding object is created or deleted in the runtime model), but events detected in cycle t are reflected in the runtime model in cycle $t + 1$. This gives a guarantee that the owner of a model element can make a decision based exclusively on the runtime model within at most two cycle periods $2 \cdot T$.

Global query consistency of runtime model. By global consistency of the (distributed) runtime model, we mean that by the time the query cycle starts, each participant has updated its own hosted model elements, and synchronized the changes with the rest of the platform participants. As such, a query initiated by two different participants will always provide the same result set within the query cycle.

The assumed single source of truth principle (i.e., each model element has a unique owner) ensures that no contradictory updates will ever be communicated. But in case of message loss during model update phase, some participants may have outdated information about some model elements. Nevertheless, the communication middleware notifies the owner of the model element about any lost messages, thus a query accessing such a model element will still use the previous (consistent) state of the object, and the new state will be reflected when all participants are successfully notified (see below).

A potential race condition may occur when two participants attempt to add a reference between a pair of objects, but this reference also has an inverse reference with at most one multiplicity, thus only one of the reference add operations can succeed. For a consistent model update, the one with the later timestamp should be enforced by introducing a self-loop transition in state AddRequest (of the link_consistency statechart) and one participant will act as a server while the other will act as a requester. Furthermore, we assume that by choosing the later timestamp to resolve such conflicts, the most recent change is reflected in the model.

This race condition is a consequence of the provided model manipulation interface, as it allows the host participant to initiate the update of any references that is an outgoing reference of a hosted object. An alternative solution to this race condition would be to change this interface and remove the possibility of setting certain references directly, and thus provide a similar asymmetric behavior as in EMF [Ste+08].

Eventual update consistency in case of message loss. While global query consistency prevents reading contradicting information in case of a message loss, such message loss may still delay the effects of a particular model update. In this case, according to our assumption on the communication middleware, notification is provided to the sender participant about the failure of delivering the message. This way the owner of a model element can prevent inconsistencies by tracking the last state surely known to all other participants as the consistent information, and will repeatedly re-send the message containing the change. For example, if some participants are not yet notified about the creation of an object then the object is considered to be non-existing (i.e., it is in the CreateMsgLost state of the object_consistency statechart). This way, *update consistency is eventually achieved* when the update is successfully communicated to all recipients. Therefore, all updates that require communication between participants will eventually take effect unless there is a more recent action which overrides its effect.

In our protocol, consistency takes precedence over availability of data in the runtime model. This means that lost messages can delay the update of the local knowledge base of a participant and thus delay the appearance of a match of a monitor. Similarly, if a participant in the platform becomes unavailable, and thus becomes unresponsive to messages, it can delay the appliation of updates to the runtime model until it becomes fully functional again. For this reason, systems sensitive to delays or have unreliable platform components can opt for a solution which aggressively updates the runtime model, but provides less strict consistency guarantees.

8.4 Evaluation

We conducted measurements to evaluate the scalability of our distributed runtime model to address the following research questions:

RQ1 How does the throughput of update operations change with increasing size of models?
RQ2 How does the distributed model update technique scale w.r.t the number of participants?

8.4.1 Measurement Setup

We carried out experiments on two different platforms to increase the representativeness of our measurements.

Real CPS platform. We use the real distributed (physical) platform of the CPS demonstrator which consists of 6 interconnected BeagleBone Black (BBB) devices (all running embedded Debian Jessie with PREEMPT-RT patch) connected to the railway track itself. This arrangement represents a distributed CPS with several computing units having only limited computation and communication resources. We use these units to maintain the distributed runtime model, and evaluate monitoring queries. This way we are able to provide a realistic evaluation, however, we cannot evaluate the scalability of the approach w.r.t the number of computing units due to the fixed number of devices in the platform.

Virtual CPS platform. To evaluate scalability w.r.t increasing number of participants, we deploy our framework over a virtual platform with Docker containers. This way, we can increase both the model size and dynamically add new participants. The containers are running Ubuntu Linux 18.04 LTS and they are all deployed to the same server machine with 32 cores and 240GB memory. A dedicated Docker network is created and assigned to the containers allowing them to communicate over a virtual local area network.

DDS middleware. We use a commercial DDS implementation provided by RTI[2] which supports the QoS settings included in the DDS specification. Furthermore, RTI provides additional options to fine-tune applications. We make minor modifications to the initial profile provided in *high_throughput.xml* to ensure timely message delivery. Namely, we increase the *max_samples* for the data writer to allow increased write throughput. Furthermore, we set the `max_flush_delay` to 100 ms to ensure periodic sending of buffered messages, and increased

[2] `https://www.rti.com/products/dds-standard`

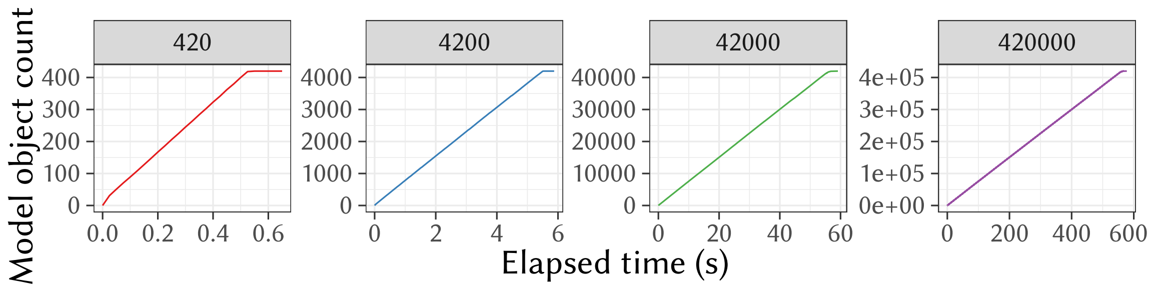


Figure 8.9: Number of model objects registered by a single participant

the `max_send_window_size` to allow larger batches of transport messages. These two parameters are both RTI's own extensions to the standard.

8.4.2 Benchmark Results on Real CPS Platform

Assessment of execution time. In the first set of experiments, we assessed how the model update throughput is affected by the size of the runtime model. Each BBB was running a single participant, while each participant was sending 70, 700, 7000 and 70000 broadcast update messages, while also listening to model updates sent by other participants.

Figure 8.9 shows our results. Each line represents a separate scenario where 420, 4.2k, 42k, and 420k objects in total were created by the participants, respectively. Furthermore, lines in the plot depict the median of how many objects a *single participant* registered over time during the experiment (both local and remote objects).

Assessment of memory footprint. We measured the heap memory consumption of our prototype maintaining a runtime model on a single BBB unit. As a baseline, we measured the total memory consumption (2nd column in Table 8.3) including memory allocated for the required DDS data structures without creating any model objects. Then, we created 420, ..., 420k model objects with their references, and checked the total memory consumption (3rd column in Table 8.3). Based on this, we calculated the average memory consumption of an object (4th column in Table 8.3).

Findings for RQ1. Figure 8.9 implies that the throughput of model updates is not affected by the actual size of the model or the number of participants. The average throughput measure is processing 797 object updates per second. Additionally, the results also point out that our approach *scales up to 420k model objects hosted across 6 participants.*

Table 8.3: Memory footprints observed in the prototype implementation

Model objects	Total memory	Model size	Avg. object footprint
420	14.77 MB	0.59 MB	1404.76B
4,200	15.89 MB	1.71 MB	407.14 B
42,000	27.76 MB	13.58 MB	323.33 B
420,000	146.50 MB	132.32 MB	315.05 B

Concerning the memory use of a single runtime model object (approx. 300-400 bytes for models with more than 4k objects), we consider our runtime model to be lightweight, which is very promising in terms of scalability w.r.t model size.

The evaluation results show that object update time does not depend on the size of the entire model, which is a key property for scalability, while the average memory footprint per model object decreases as the size of the model increases.

Limitation. The loaded libraries and initial DDS data structures (mainly DDS topics) in our setup prevents our prototype to be deployed on devices with less than 15 MB memory. Note that only around 3 MB of memory is dedicated to message buffers introduced by our middleware configuration (i.e., to send batch messages to increase throughput), the rest of memory consumption would be noticed for any DDS-based implementation using the same (industrial) library. The measured memory usage is in accordance with the memory benchmark results provided by RTI[3].

In fact, there is a newer standard called DDS-XRCE [Obj19] dedicated to support devices with very limited available memory. This standard is an extension of the initial DDS specification designed to support resource-constrained environments and could provide a much lower runtime overhead. However, by the time of performing the experiments included in this thesis, no implementation was available to us for evaluation.

8.4.3 Benchmark Results on a Virtual CPS Platform

We used the virtual CPS platform to allow the scalability assessment of the model update protocol w.r.t the number of participants in the platform. In this set of experiments, we assessed how the model update throughput is affected by the number of participants present in the

[3] `https://www.rti.com/products/benchmarks`

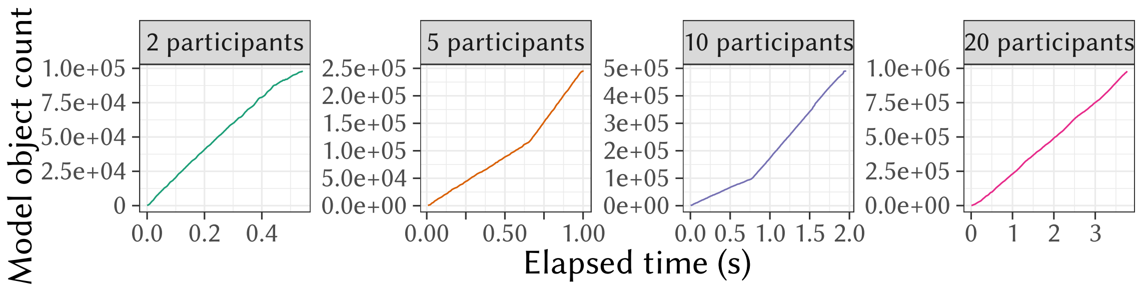


Figure 8.10: Registered objects over time by a single participant (median is shown)

system. Each participant was sending the same amount of broadcast update messages (each creating almost 50K new model objects), while also listening to model updates sent by other participants.

Findings for RQ2. Figure 8.10 shows our results. Each line represents a separate scenario where 2, 5, 10, and 20 participants were active, respectively. Furthermore, lines in the plot depict the median of how many objects a single participant registered over time during the experiment (both local and remote objects). This suggests that the throughput of model updates is unaffected both by the actual size of the model and the number of participants. Additionally, the results also point out that our approach scales up to 1M model objects hosted across 20 participants. Note that the charts in Figure 8.10 show the number of objects observed by a single participant over time, thus depending on the message dissemination provided by the underlying middleware, the arrival rate of object update messages may change, as observed in the case of 5 and 10 participants about 0.7 seconds after the start of the measurements.

We also assessed throughput for different model update types. Figure 8.11 shows the results for measuring the capability of a single participant to process various model update messages over the first 1 second of the benchmark with 10 participants. On average, 1708 objects are registered every 10 ms, while this value is only 286 and 462 for processed reference update requests and replies, respectively, which are promising performance indicators for a software prototype.

The distributed runtime model management protocol is able to support up to 20 participants in a virtual platform with a throughput of 250k object updates per second.

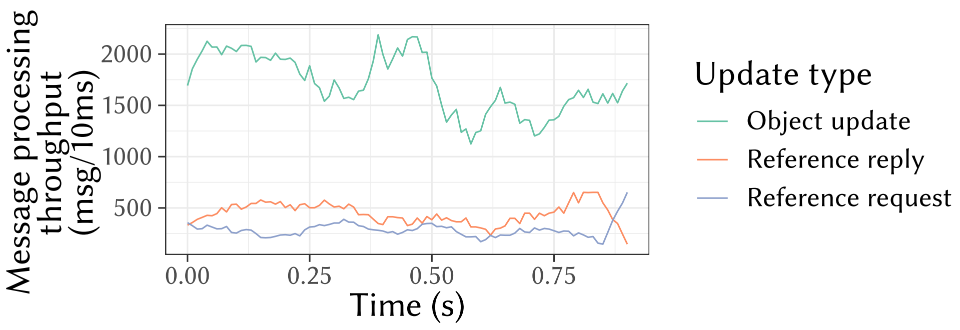


Figure 8.11: Throughput processing comparison by different model update messages on a single participant

8.4.4 Threats to Validity

Construct validity. In our assessment, we did not use fault injection to enforce message losses in order to investigate the fault tolerance capabilities of our protocol. Furthermore, update operations dealing with object deletion and reference removal in our protocol are more simple than the ones with object creation and reference addition. In our evaluations, we focussed on the latter two which represent the more complicated cases.

Internal validity. To measure the performance of our distributed model management approach, participants executed only model management tasks, thus other external factors can be excluded. In Section 8.4.3, we ran Linux containers on a remote server located in a cloud infrastructure. As such, we had very limited influence on the allocation of the machines and the potential workload that is present on the same physical host as our instances. As such, a non-cloud-based setup may yield different run times.

External validity. The generalizability of our experimental results is limited by the fact that our experiments were carried out on isolated local area networks, where the network latency is (most likely) significantly lower than on real networks with additional traffic.

8.5 Summary

In this chapter, we introduced a distributed runtime model management protocol for CPS. The presented protocol ensures that queries initiated by any participant in the system provide consistent results even in case of message loss during a model update. We implemented the proposed protocol on top of a cutting edge commercial DDS library, and evaluated the scalability of the approach. The evaluation was performed on both a real and virtual CPS platforms, and the results showed that the approach is able to scale up to 20 participants and 1M model objects, which are promising results for our software prototype. This chapter introduced the results related to the first contribution group (Co1.2, Co1.3, and Co1.4) for distributed resource-constraint CPS platforms.

Publications related to this chapter. The distributed runtime model management protocol is presented in the journal article [*j*2]. The development of the protocol is a joint effort by my supervisor and myself. The implementation of the software prototype and evaluation of the approach on the real CPS platform is in part the contribution of Gábor Szilágyi. I have performed the evaluation on the virtual platform. András Vörös was helping the work as advisor and provided continuous feedback.

Chapter 9

Distributed Graph Queries

This chapter extends query-based runtime monitors introduced in Chapter 6 to distributed systems. To evaluate graph queries of runtime monitors in a distributed setting, we propose to deploy queries to the platform in a way that is compliant with the distributed runtime model and the potential resource restrictions of computation units. If a graph query engine is deployed as a service on a computing unit, it can serve as a *local monitor* over the runtime model. However, such local monitors are usable only when all graph nodes traversed and retrieved during query evaluation are deployed on the same computing unit, which is generally not the case. Therefore, while the local evaluation of queries is still preferable for performance reasons, a *distributed monitor* needs to gather information from other model fragments and monitors stored at different computing units.

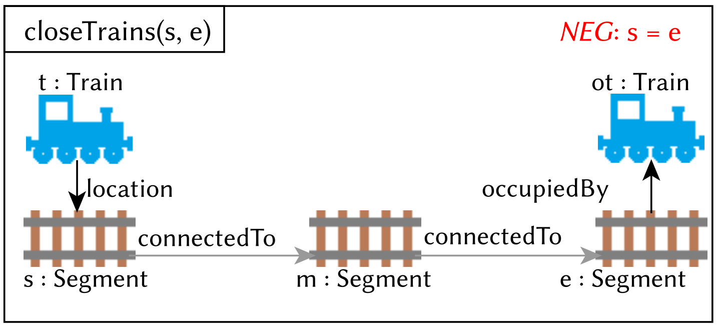
In this chapter, we present an approach for evaluating distributed queries. Section 9.1 discusses search-based query evaluation strategies. Then, in Section 9.2 we show in detail an adaptation of the evaluation algorithm presented in Algorithm 3.2.1, and discuss how 3-valued logic can be used to represent uncertain matches caused by message losses during query evaluation. Finally, in Section 9.3 we evaluate the proposed approach using the MoDeS3 demonstrator and the Train Benchmark [Szá+17] query benchmark framework and discuss our findings.

9.1 Strategies for Distributed Runtime Monitoring

As in Chapter 6, we rely on graph queries to capture the monitored properties. This declarative description allows to capture safety properties on a high level of abstraction over the distributed runtime model, which eases the definition and comprehension of safety monitors for engineers and avoids accidental complexity caused by additional platform-specific or de-

```
pattern closeTrains(s, e) {
  Train.location(_t, s);
  Segment.connectedTo(s, m);
  Segment.connectedTo(m, e);
  Segment.occupiedBy(e, _ot);
  s != e
}
```

(a) Query closeTrains captured in VQL



(b) Graphical query presentation

$$\varphi_{\mathrm{CT}}(s,e) = \exists t : \mathtt{Train}(t) \land \mathtt{Location}(t,s) \land \exists m : \mathtt{ConnectedTo}(s,m) \land \mathtt{ConnectedTo}(m,e) \land \neg(s=e) \land \exists ot : \mathtt{OccupiedBy}(e,ot)$$

(c) Graph query as logic predicate

Figure 9.1: Monitoring goal formulated as a graph query φ_{CT} for closeTrains

ployment details. The synthesized distributed monitoring programs exploit the knowledge about the ownership of data and ensure that matches of queries are collected from the entire runtime model rather than just the locally available model fragment.

Example 22. In the railway domain, safety standards prescribe a minimum distance between trains on track [Abr+08; Eme11]. The closeTrains monitor definition captures a (simplified) description of the minimum headway distance to identify violating situations where trains have only limited space between each other. Technically, one needs to detect if there are two trains on two different segments of the track, which are connected by a third segment. Any match of this pattern highlights track elements where passing trains need to be stopped immediately. Figure 9.1a shows the monitoring query closeTrains in textual VQL syntax, Figure 9.1b shows a graphical presentation, and Figure 9.1c displays the definition as a graph formula φ_{CT}.

Our system-level runtime monitoring framework is *hierarchical* and *distributed*. Monitors may observe the local runtime model of a participant, and they can collect information from runtime models of different participants. Moreover, one monitor may request information from other monitors, thus yielding a hierarchical network.

Similarly to evaluating queries on a single device, monitors compute matches of a graph query $\varphi(v_1, \ldots, v_n)$ along a *search plan* by assigning model objects to variables $v_1, \ldots, v_n$ and evaluating the predicate of the query. A search plan is an ordered list of search operations (e.g. checking type of objects, navigating along references) that traverses the runtime graph model in order to find all complete variable bindings satisfying the query condition. The search plan for $\varphi(v_1, \ldots, v_n)$ also depends on the initial binding information for the input parameters (provided by the caller) as a variable with a fixed value can greatly reduce the search space.

In distributed query-based monitoring, a significant challenge in executing search plans is to manage navigation along references and reading an attribute values of objects hosted by other participants. Additionally, we wish to allow that monitors initiated by an arbitrary participant can return all violations in the entire system. We have identified two possible approaches to support these requirements:

- **Single executor**: a single participant executes all steps of the search plan while accessing information stored at other participants, which can be achieved by a synchronous remote procedure call implementation over DDS. The request message conveys (i) what reference/attribute value of (ii) what object is requested and (iii) by which participant, while the reply message encapsulates the (i) object identifier and (ii) an array of values.
- **Multiple executors**: the participant which initiates the execution of a monitor will not directly request information from other participants. Instead, if a remote object is encountered and its reference/attribute value is queried, the partial variable bindings are asynchronously passed to the other participant, which continues the execution of the search plan. The request message needs to contain (i) an auto-generated unique request identifier, (ii) the partial variable binding, (iii) the next step in the search plan and (iv) the requester participant ID (i.e., the hostId attribute in Figure 8.1). Once the other participant receives this message, it continues the matching and finally, it returns all found matches to the requester in a reply message with (i) the set of matches and (ii) the request identifier.

As we have shown it in one of our former papers [*c*4], the multiple executors strategy turned out to be far superior to the single executor one. For this reason, in this thesis we are only using the multiple executors strategy when evaluating distributed queries.

9.2 Distributed Evaluation of Graph Queries

Query execution is the last step of our proposed time-triggered execution loop in Section 8.2.2. During this time, the underlying distributed runtime model is assumed to be fixed, i.e., queries are executing over a snapshot of the system.

9.2.1 A Query Cycle

Monitoring queries are evaluated during the so-called *query cycle*. We assume that the search plan for each monitoring query has been made available to all participants prior to this query cycle phase to support query evaluation at any participant. During query evaluation, each participant uses this same search plan.

When participants compute matches in a distributed way, they simultaneously evaluate predicates of the query on the values of the bound variables. However, in cases when a predicate evaluation cannot be computed based on the local knowledge of a participant, the matching should be delegated to the participant hosting the corresponding part of the distributed runtime model M_{distr}. The delegation is possible through proxies representing remote objects in the local runtime model, based on the object ownership discussed in Section 8.2.2.

We extend Algorithm 3.2.1 for a distributed platform in Algorithm 9.2.1. A monitor execution at a participant can be initiated by calling FindAllMatches (line 28 in Algorithm Algorithm 9.2.1). There are two key cases that require further consideration:

- *Delegating execution*: The distributed runtime model M_{distr} refers to the unified knowledge of multiple participants about the system, where each element of the model is owned by a single participant. This way, if the distributed query execution algorithm is finding matches over the complete runtime model, it needs to take into account matches formed by joining the locally stored parts of the complete model. To support the distributed execution, we added an extra condition for evaluating extend search operations to check if the value for the newly bound variable is part of the local knowledge base. If this is not the case, query execution is delegated to the owner of the data, i.e., receiver will execute the Continue procedure. This extension is shown in lines 9-10 in Algorithm 9.2.1.
- *Gathering matches*: Delegating a query execution to a remote participant can be done asynchronously in accordance with the actor model [HBS73] as in lines 12-14. This way, finding local matches can continue without waiting for replies from remote participants. However, the execution of neither FindAllMatches nor Continue cannot be completed before awaiting all matches from remote participants and fusing them with the local results (see lines 30-31 and lines 24-26 respectively).

Algorithm 9.2.1: Distributed query execution outline

```
Function ExecuteQuery($\varphi$, $idx$, $Z_p$) is
    $searchPlan \leftarrow$ LookupPlan($\varphi$)
    if size($searchPlan$) = $idx$ then return $\{Z_p\}$ ;
    $matches \leftarrow \emptyset$
    PRED $\leftarrow$ predicate evaluated by $searchPlan[idx]$
    if $searchPlan[idx]$ is extend then
        for $e$ in {all candidates in $M$} do
            $Z'_p \leftarrow Z_p \cup \{v_F \mapsto e\}$
            if $e$ is not owned by current participant then
                $future$←Continue($sender$,$\varphi$,$idx$,$Z'_p$) store $future$
            else if $[\![\text{PRED}]\!]^M_{Z'_p} = 1$ then
                $next \leftarrow idx + 1$
                $matches \leftarrow matches \cup$ExecuteQuery($\varphi$,$next$,$Z_p$)
            end
        end
    else if $[\![\text{PRED}]\!]^M_{Z_p} = 1$ then
        $next \leftarrow idx + 1$
        $matches \leftarrow matches \cup$ExecuteQuery($\varphi$,$next$,$Z_p$)
    end
    return $matches$
end
Procedure Continue($sender$,$\varphi$,$idx$,$Z_p$) is
    $matches \leftarrow$ExecuteQuery($\varphi$, $idx$, $Z_p$)
    await all futures stored in ExecuteQuery
    add remote results to $matches$
    send $matches$ to $sender$
end
Procedure FindAllMatches() is
    $allMatches \leftarrow$ ExecuteQuery($\varphi$, 0, $\emptyset$)
    await all futures stored in ExecuteQuery
    add remote results to $allMatches$
end
```

9.2.2 Semantics of Distributed Query Evaluation

Each query is initiated at a given computing unit which will be responsible for calculating query results by aggregating the partial results retrieved from its neighbors. This aggregation has two different dimensions: (1) adding new matches to the result set calculated by a different participant, and (2) making a potential match more precise. While the first case is a consequence of the distributed runtime model and query evaluation, the second case is caused by uncertain information caused by message loss/delay.

Fortunately, the definition of graph queries in Definition 7 can be used with 3-valued logic with logic values 1, 0, and $1/2$, where the latter represents uncertain or unknown information.

This can be used to provide semantic treatment for the first case: any match reported to the requester by any neighboring participant will be included in the query results if its truth evaluation is 1 or $1/2$. As such, any potential violation of a safety property will be detected, which may result in false positive alerts but critical situations would not be missed.

However, the second case necessitates extra care since query matches coming from different sources (e.g. local cache, reply messages from participants) need to be fused in a consistent way. This match fusion is carried out by participant p as follows:

- If a match is obtained exclusively from the local runtime model of p, then it is a certain match, formally $[\![\varphi(o_1, \ldots, o_n)]\!]@p = 1$.
- If a match is sent as a reply by multiple participants p_i that were sent a request to continue the evaluation of a query, (with $p_i \in cont(p)$), then we take the most certain result at p, formally, $[\![\varphi(o_1, \ldots, o_n)]\!]@p := \underline{\max}\{[\![\varphi(o_1, \ldots, o_n)]\!]@p_i | p_i \in cont(p)\}$.
- Otherwise, tuple $o_1, \ldots, o_n$ is surely not a match: $[\![\varphi(o_1, \ldots, o_n)]\!]@p = 0$.

Note that in the second case uses $\underline{\max}\{\}$ to assign a maximum of 3-valued logic values wrt. *information ordering* (which is different from the numerical maximum used in Definition 7). Information ordering is a partial order $(\{1/2, 0, 1\}, \sqsubseteq)$ with $1/2 \sqsubseteq 0$ and $1/2 \sqsubseteq 1$. It is worth pointing out that this distributed truth evaluation is also in line with Sobociński 3-valued logic axioms [Sob52].

Example 23. Figure 9.2 shows 6 messages from the beginning of a query evaluation sequence for monitor closeTrains initiated at participant P_1 over the runtime model depicted in Figure 8.2. Calls are asynchronous (cf. actor model), and numbers represent the order between timestamps of messages. In this example, only (the first few) requests are shown and replies are omitted to keep the illustrative example simple.

When the query is initiated (message 1 or <u>m1</u> for short), its query identifier is sent along with the initially empty list of variable bindings and the search operation index 0. Then, according to the first search operation, participant P_1 will look for appropriate variable values variable t potentially satisfying the predicate `Train`. Three objects are considered: tr_0, tr_1 and tr_2 out of which tr_1 is managed by the remote participant P_2, so that <u>m2</u> is sent from P_1 to P_2 delegating query execution by supplying the query identifier, the index of the next search operation, and the computed variable binding. Similarly, in <u>m3</u>, a message is sent to P_3 with the $t \mapsto \mathsf{tr}_0$ binding, as tr_0 is hosted by P_3. Once P_1 sent the messages in a non-blocking way, it proceeds with the execution of the search plan.

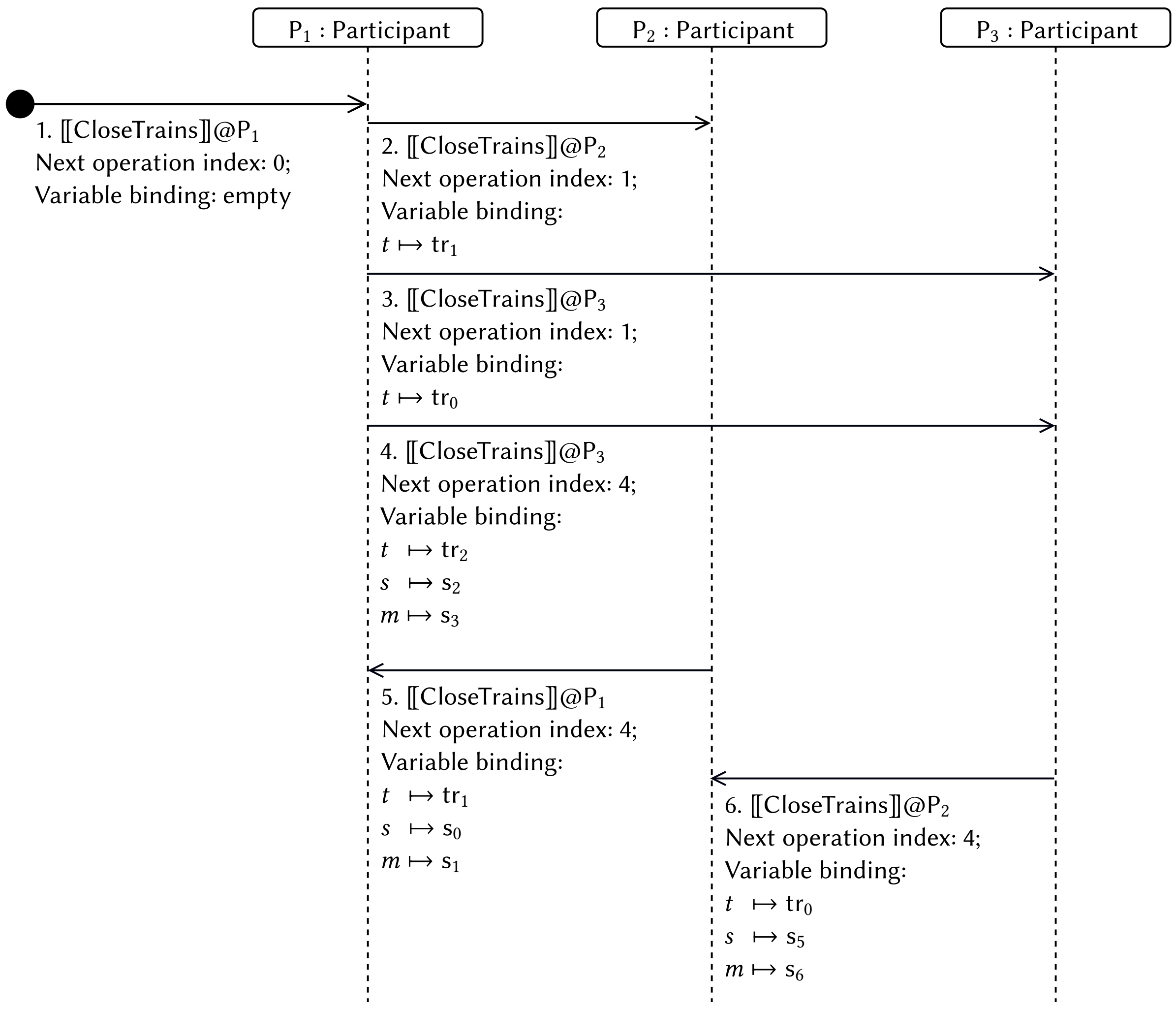


Figure 9.2: Query execution requests across participants while evaluating closeTrains

Next, when the binding with $t \mapsto \mathrm{tr}_2, s \mapsto \mathrm{s}_2,$ and $m \mapsto \mathrm{s}_3$ is computed by P$_1$ in search step #3, it is sent to P$_2$ in $\underline{\text{m4}}$ along with the next operation index, 4. The next message $\underline{\text{m5}}$ sent by P$_2$ is a follow-up request to $\underline{\text{m2}}$ with the $t \mapsto \mathrm{tr}_1, s \mapsto \mathrm{s}_0,$ and $m \mapsto \mathrm{s}_1$ binding is sent to P$_1$, the host of s$_1$. The last message depicted in Figure 9.2, $\underline{\text{m6}}$ is also a follow-up request by P$_3$. It is based on $\underline{\text{m3}}$ and sent to P$_2$, the host of s$_6$.

9.2.3 Performance Optimizations

Each match sent as a reply to a participant during distributed query evaluation can be cached locally to speed up the re-evaluation of the same query within the query cycle. This *caching of query results* is analogous to *memoing* in logic programming [War92].

Currently, cache invalidation is triggered at the end of each query cycle by the local physical clock, which we assume to be (quasi-)synchronous with high precision across the platform.

This memoing approach also enables units to selectively store messages in the local cache depending on their specific needs. Furthermore, this can incorporate to deploy query services to computing units with limited amount of memory and prevent memory overflow due to the several messages sent over the network.

9.2.4 Semantic Guarantees and Limitations

Consistency. Our approach ensures that query execution initiated at any two participants will not yield contradicting query results. This is achieved by the single source of truth principle when only an owner of an object can serve a read request during query execution.

Furthermore, in case of a communication failure, the results may contain incomplete or uncertain matches by the end of the query cycle. However, these will (1) overestimate the complete set of query results and (2) two result sets obtained by two different platform units will still not contradict each other.

Termination. We can guarantee that query evaluation terminates despite potential message losses, i.e., there is no deadlock or livelock in the distributed query protocol. To show this property, it is enough to see that the evaluation of queries (1) is a monotonous process in terms of search operation execution and (2) a search operation cannot halt the execution. Condition (1) holds because whenever a participant is executing an operation that incurs query delegation, the delegation will start from the next operation in the plan. This way execution will never go back to a previous operation. Condition (2) holds because the model size is bounded thus all model elements can be traversed.

Assumptions and limitations. There are also several assumptions and limitations of our approach. We only assumed delay/loss of messages, but not the failures of computing units. We also excluded the case when participants maliciously send false information. Instead of refreshing local caches in each cycle, the runtime model could incorporate information aging which may enable to handle other sources of uncertainty (which is currently limited to consequences of message loss). Finally, in case of longer cycles, the runtime model may no longer provide up-to-date information at query evaluation time. We believe that some of these limitations can be handled in future work by various adaptations of the query evaluation protocol.

9.3 Evaluation

We conducted measurements to evaluate the scalability of our distributed query evaluation technique to address the following research question:

RQ1 How does the distributed graph query execution technique scale w.r.t model size?
RQ2 How does the distributed graph query execution performance change with increasing number of participants in the execution platform?

9.3.1 Benchmark Setup

Similarly to the evaluation presented in Section 8.4, in order to increase the representativeness of our measurements, we have used a physical and a virtual platform, and the prototype implementation relies on the same commercial DDS library for network communication. In addition, we have implemented distributed runtime monitors using two different case studies for the two different evaluation platforms.

Real CPS benchmark. For the measurements over the real CPS platform, We rely on the MoDeS3 railway CPS demonstrator as the domain of our experiments to synthesize various distributed runtime models. Since the original runtime graph model of MoDeS3 has only a total of less than 100 objects and a total of 6 participants, we scale up this initial model. To ensure that structurally consistent models are generated, we follow a template-based method, which is a simplified version of [He+17]. Altogether, we use the same model generator and queries as in our former paper [*c*5] to be able to compare our results with the ones obtained at an earlier stage of the development. We executed the queries introduced in Example 1 over these scaled-up models.

Virtual CPS benchmark. For the measurements over the virtual platform, we use the model generator and graph queries of the open Train Benchmark [Szá+17] by making only the necessary technological adaptations for a DDS-compatible execution platform. We did not implement all queries of the benchmark, but selected three with different complexities after consulting with the authors of the benchmark.

9.3.2 Benchmark Results Over Real CPS Platform

Query execution times.

The query execution times over models deployed to a single BBB were first measured to obtain a *baseline evaluation time of monitoring* for each rule (referred to as *local* evaluation). Then the execution times of system-level distributed queries were measured over the platform with 6 BBBs, evaluating two different allocations of objects (*standard* and *alternative* evaluations).

In Figure 9.3 each result captures the times of 29 consecutive evaluations of queries, while Figure 9.4 shows the average run times of each query over models with different sizes. Prior to each benchmark, an initial warm-up evaluation is done and the results of this step are discarded. A query execution starts when a participant initiates evaluation, and terminates when all participants have finished collecting matches and sent back their results to the initiator.

Overhead of distributed evaluation. On the positive side, the performance of graph query evaluation on a single unit is comparable to other graph query techniques reported in [Szá+17] for models with approximately 0.5M objects, which shows a certain level of maturity of our prototype. Furthermore, the CPS demonstrator showed that distributed query evaluation yields run times, which are comparable with run times yielded by local evaluation on models over 4200 objects in 3 out of the 4 cases, which is promising. In case of the most complex *Misaligned turnout* query, which uses multiple query calls, the distributed evaluation takes up to 13.77× longer compared to local evaluation, which shows the negative impact of query calls on the execution time.

Finally, distributed query evaluation on larger models had performance problems with *Train locations*, which is a simple query (2.92× slower execution compared to local evaluation on the largest model). The reason is this query has a large result set that roughly equals to 20% of the complete model size, thus communication of results imposes intense network traffic.

Altogether, our measurements results in Figure 9.3 indicate one order of magnitude better scalability for query execution compared to results reported in our former paper [*c*5].

Impact of allocation on query evaluation. We synthesized different allocations of model elements to computing units to investigate the impact of allocation of model objects on query evaluation. With the real CPS benchmark model in particular, we chose to allocate all Trains

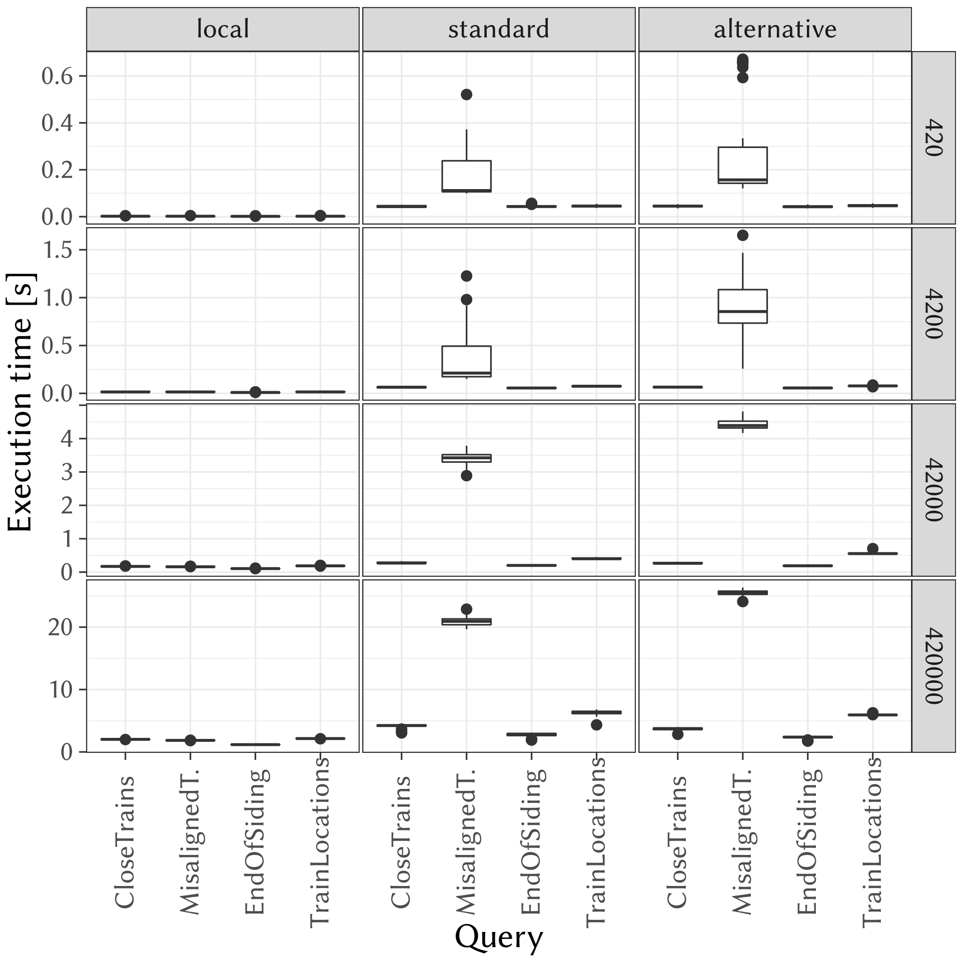


Figure 9.3: Query execution times in MoDeS3 case study

to a dedicated BBB, and assigned every other node stored previously on this BBB to the rest of the participants.

Interestingly, we see no major difference in run times of individual queries between the two different allocation scenarios (standard and alternative), while previous results [*c*5] showed 19.92× slowdown for extreme cases using the alternative allocation. However, since that initial prototype, we managed to significantly improve the distributed query evaluation algorithm, and exploit the high data throughput of the DDS communication middleware.

We find that distributed graph query evaluation performance is comparable to that of local evaluation, but subquery calls and large query match sets mean performance bottlenecks.

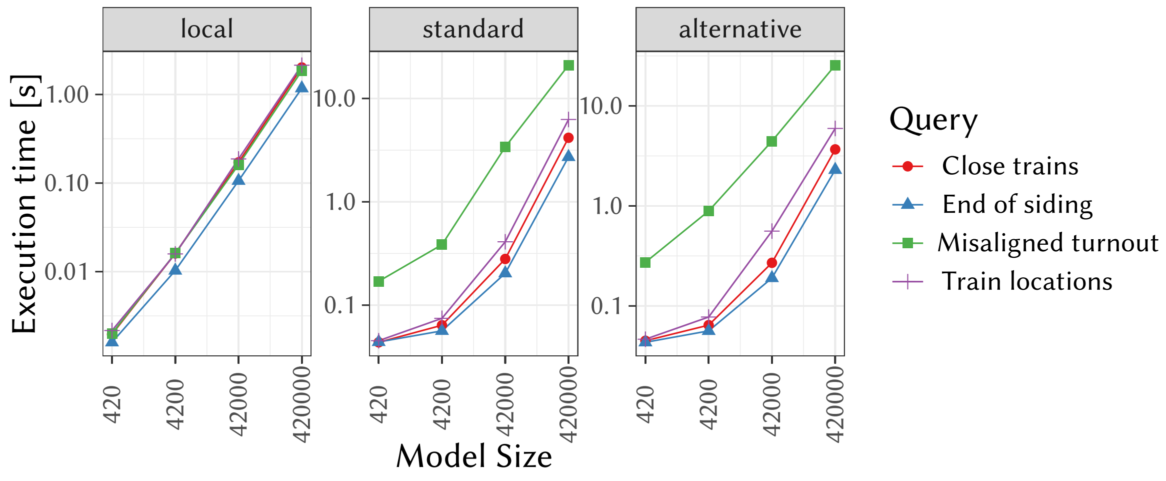


Figure 9.4: Average query execution times in MoDeS3 case study

9.3.3 Virtual CPS Benchmark Results

Scalability of query evaluation over a virtual platform. With the virtual CPS platform we aimed at assessing how our query-based runtime monitoring approach performs w.r.t the number of participants in the platform. To achieve this, we adapted the model generator component of Train Benchmark [Szá+17] to also supply the generated models with allocation information. Then, we generated models with objects 1.3k – 250k for four different allocation to 2, 5, 10, and 20 participants, respectively. Figure 9.5 shows the run times of 30 subsequent query evaluations over a virtual CPS platform consisting of multiple Docker containers, while Figure 9.6 shows the average of query run times over different models.

The results show that initially, query execution times are approximately the same for all allocations. Then, starting from 64k objects, execution times over the same model size gradually start decreasing as the number of participants increases. The biggest gain on average is for the query SwitchSet query: evaluation on a platform with 20 participants over a model with 250k elements is 2.28× faster than on a platform with only 2 participants. This means that increasing the degree of distribution in the system yields lower execution times for queries if the models are larger than a certain size.

A higher degree of distribution in the system results in lower query evaluation time for models over 126k objects, while it has negligible performance impact for smaller models.

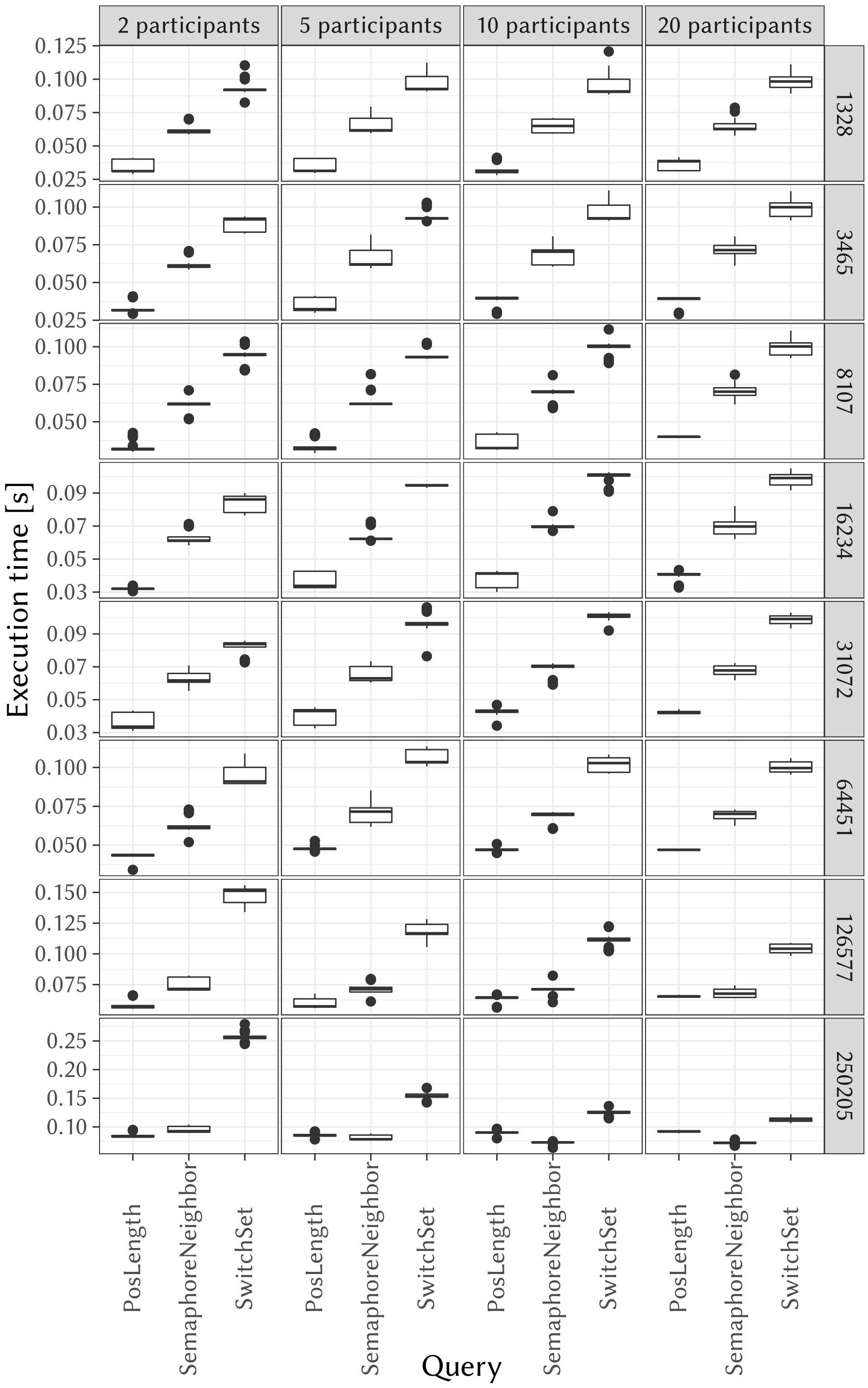


Figure 9.5: Train Benchmark scalability evaluation results (individual run times)

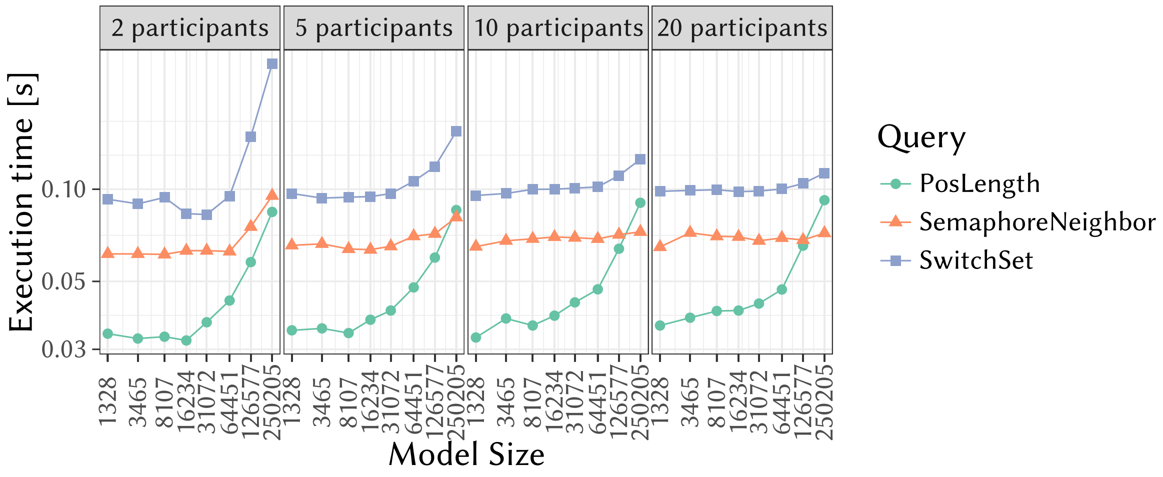


Figure 9.6: Train Benchmark scalability evaluation results (average run times)

9.3.4 Threats to Validity

Construct validity. We assessed query execution performance using a single query plan synthesized automatically by the VIATRA framework that uses heuristics for query execution a single computation unit insead of considering a distributed platform. We believe that execution times of distributed queries could be further decreased with a carefully constructed search plan. In our assessment, we did not use fault injection to enforce message losses in order to investigate the fault tolerance capabilities of our protocol.

Internal validity. First, to measure the performance of our approach, the platform devices executed only query services. We ran Linux containers on a remote server located in a cloud infrastructure, so that we had very limited influence on the allocation of the machines and the existing workload that is present on the same physical host as our instances may impact the results.

External validity. Host machines running distributed query programs are connected to an isolated local area network: in Section 9.3.2, the physical platform units were connected via local Ethernet connection, while in Section 9.3.3, a virtual Ethernet network was used. Performance on a real network with a busy channel would likely have longer delays and more message losses thus increasing overall execution time.

9.4 Summary

In this chapter, we presented a distributed graph query evaluation for runtime monitoring of CPSs. We adapted the local search-based query evaluation algorithm to a distributed setting, and showed how 3-valued logic can be used to represent uncertain information caused by network errors in query results. We performed scalability assessment of the proposed query evaluation approach using two different platforms and two different case studies. In particular, this chapter showed our results for the second contribution group (Co2.2–Co2.5).

Publications related to this chapter. The distributed query evaluation was presented at several international conferences [*c*5; *c*4]. The formal definition of graph query evaluation using 3-valued logic, and the definition of multiple distributed query evaluation strategies is my contribution. The development of the software prototype is a joint effort of Gábor Szilágyi and myself. The evaluation of the approach on a real CPS platform is the contribution of Gábor Szilágyi, while the evaluation on the virtual platform is my contribution. András Vörös was helping the work as advisor and provided continuous feedback.

Chapter 10
Final Conclusion & Future Work

This chapter provides a summary and concluding remarks regarding the contributions of this thesis in Section 10.1, while Section 10.2 discusses future work.

10.1 Thesis Summary

Runtime monitoring of smart and safe cyber-physical systems has become an important topic in the last decade. This thesis investigated novel runtime monitoring approaches which are predicated on well-established graph-based techniques typically used in the design tools of such systems. However, the adaptation of these techniques faces several serious challenges which we addressed in this work.

Runtime monitoring of CPS by graph queries In this thesis, we proposed a runtime verification technique for smart and safe CPSs by using a high-level graph query language to capture safety properties for runtime monitoring and runtime models as a rich knowledge representation to capture the current state of the running system. The approach was implemented and evaluated on the physical system for multiple resource-constrained environments typical in CPS applications. Our first results show that it scales for medium-size runtime models even on platforms where the available memory space is limited.

Timing analysis of query-based monitors In this paper, we presented a method to *provide safe and practical WCET bounds for runtime monitoring programs derived from graph queries* to enable their use in real-time systems. On the one hand, we provide a static WCET estimate by incorporating low-level analysis results from traditional IPET-based tools and high-level domain-specific constraints into the objective function of an advanced graph solver. In addition to a tight WCET estimate, the result also entails a witness graph model where the query-based monitoring program execution time is expected to be the longest. On the other hand, we combine state-of-the-art parametric WCET computation with runtime graph statistics to allow online WCET recomputation at runtime upon relevant model changes to enable to reallocate time slots to a tighter bound.

We carried out extensive evaluation of our approach on an industry-grade hardware platform using a variety of graph models as inputs for query programs, and assessed the tightness of computed WCET using three different algorithms. We managed to construct witness models for highest estimated execution times of queries as well as random graph models as inputs for graph query programs as an attempt to showcase high execution times. While we have no formal guarantee that worst-case timing behavior is exhibited on witness models as inputs, in all our experiments, the longest run times were always measured on such witness models.

Distributed runtime graph models We showed how our runtime monitoring approach can be extended for distributed CPS. The solution is a time-triggered runtime model management approach, that keeps the information in the model close to the data sources. Models and high-level graph queries provide an expressive language to capture correctness requirements during runtime. Our solution is built on top of the standard DDS reliable communication middleware that is widely used in self-adaptive and resource constrained CPS applications.

Our approach introduces an efficient handling of a distributed knowledge base stored as a graph over a heterogeneous computing platform. Consistent manipulation and update of the knowledge base is defined as a distributed and time-triggered model management protocol and implemented by exploiting the fine-grained QoS guarantees provided by the DDS communication middleware.

The scalability of our approach was evaluated in the context of the physical system of MoDeS3 CPS demonstrator with promising results such as high throughput for model updates and good scalability with increasing change sizes and number of participants.

Query-based runtime monitoring in distributed systems We proposed a query evaluation protocol for monitors in smart and safe CPSs with distributed components. The monitor specification language is execution platform-agnostic and captures safety properties for runtime monitoring on a high level of abstraction. A distributed query evaluation technique was introduced where none of the computing units has a global view of the complete system. The approach was implemented and evaluated on the physical system of MoDeS3 CPS demonstrator as well as on a virtual CPS environment to assess scalability. In our results, distributed monitor evaluation in most cases comparable to local evaluation times on a single device, which is a positive sign regarding the viability of the solution.

Finally, a summary of research questions, objectives, and contributions of this thesis is shown in Table 10.1.

Table 10.1: Summary of contributions

				Ob1.	Ob2.	Ob3.	Ob4.	Ob5.
			Section related to the contribution	Precise semantics	Predictable execution time	Resource-constrained platform	Scalable execution	Integration with existing frameworks
RQ1.	Real-Time Platform	Concept: *Graph Data Structures for ES*	5.1	Co1.1				
		Prototype Implementation	5.1			Co1.3		
		Scalability Evaluation	5.2				Co1.4	
	Distributed Platform	Concept: *Distributed Runtime Model*	8.1	Co1.2				
		Prototype Implementation	8.2			Co1.3		Co1.4
		Scalability Evaluation	8.4				Co1.4	
RQ2.	Real-Time Platform	Concept: *Graph Query at Runtime*	6.1	Co2.1				
		Prototype Implementation	6.2			Co2.4		
		Scalability Evaluation	6.3				Co2.5	
	Distributed Platform	Concept: *Distributed Query at Runtime*	9.1	Co2.2-3				
		Prototype Implementation	9.2			Co2.4		Co2.4
		Scalability Evaluation	9.3				Co2.5	
RQ3.	Real-Time Platform	Concept: *Static WCET Estimation*	7.3, 7.4		Co3.1			
		Assessment of WCET Estimate	7.7			Co3.4		Co3.2
		Concept: *On-line WCET Estimation*	7.5, 7.6		Co3.3			
		Assessment of WCET Estimate	7.7			Co3.4	Co3.4	

10.2 Future Work

We summarize the research directions in three general areas: runtime graph models, query-based runtime monitoring, and timing analysis of query-based monitors.

10.2.1 Runtime Graph Models

A short-term research goal is to provide and evaluate a fully-fledged hard real-time model management framework based on the presented requirements in Chapter 5 for resource-constrained embedded systems. Currently, we provide a prototype implementation, but it will be interesting to derive safe and tight WCET bounds of certain model update operations.

In terms of the distributed runtime model update protocol presented in Chapter 8, a remaining research task is to investigate in details what general properties does the proposed distributed runtime model protocol guarantee (e.g., global consistency, fairness, and liveness). Although we have tested several corner cases using the simulator features of the statechart modeling tool that we used for capturing the update protocol, theorems with formal proofs would greatly help to increase confidence in the protocol.

10.2.2 Distributed Query Evaluation

The details of search plan generation for distributed query evaluation is neglected in Chapter 9, and we rely on former results in the field of graph pattern matching. For this reason, the query search plans used in our approach are optimized for models stored in a central location. In the future, it will be useful to investigate how to characterize effective search plans with graph model allocations in the context of distributed queries used for runtime monitoring.

Furthermore, a promising research direction is to experiment with dynamic search plans for query evaluation [Var+15], where significant changes to the runtime model structure and model allocation can trigger changes to the query search plan at runtime which can improve performance. In addition, more efficient query evaluation algorithms can be incorporated into the system to provide near real-time analysis capabilities.

As a part of a long-term future research agenda, the presented WCET estimation approach for query-based runtime monitors could be extended to a distributed setting to provide timing guarantees for distributed monitoring queries as well. In addition to the characteristics of the graph model, query execution time estimates need to take into account the latency of the network and the allocation of data.

Moreover, as a long-term goal, the graph query based approach could be integrated with temporal logic languages to support an even wider range of specifications. As we pointed out in Section 6.1, graph queries can be extended to express temporal behavior [DRV18] but the current work is restricted to structural safety rules.

10.2.3 Timing Analysis

In the short run, the proposed WCET estimation approach for graph query programs in Chapter 7 can be improved by incorporating hardware platform-specific constraints in the objective function of the witness model generation task directly rather than relying on timing analysis results from an IPET analysis. Currently, the results from IPET provide basic block timings which are represented as constants in the objective function, which increase the overestimation. Capturing the hardware-specific constraints directly in the model generator enables more precise WCET calculation by adding execution context sensitivity to basic block timings.

Additionally, the evaluation of the presented WCET estimation approach should be done on further hardware platforms to demonstrate the generalizability of the approach. Ideally, devices that have advanced hardware features (e.g., multicore) or use different core designs should be used as well.

As a part of a long-term future research agenda, the presented WCET estimation approach could be extended to provide witness models with specific data placement in memory where the execution time equals to the WCET of the program. It will be interesting to connect the memory-specific features of the underlying hardware with the abstract features of the graph models to determine the longest possible execution times.

Bibliography

[Abe+15] Jaume Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. *10th IEEE International Symposium on Industrial Embedded Systems - Proceedings*, 2015, pp. 39–48. DOI: 10.1109/SIES.2015.7185039.

[Abr+08] Montserrat Abril, Federico Barber, Laura Ingolotti, Miguel A. Salido, Pilar Tormos, and Antonio Lova. An assessment of railway capacity. *Transportation Research Part E: Logistics and Transportation Review* 44(5), 2008, pp. 774–806.

[Aer11a] Radio Technical Commission for Aeronautics. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. 2011.

[Aer11b] Radio Technical Commission for Aeronautics. DO-330: Software Tool Qualification and Considerations. 2011.

[Afz+17] Wasif Afzal et al. The MegaM@Rt2 ECSEL project: megamodelling at runtime - scalable model-based framework for continuous development and runtime validation of complex systems. *Proceedings - 20th Euromicro Conference on Digital System Design, DSD 2017*, 2017, pp. 494–501. DOI: 10.1109/DSD.2017.50.

[ANR17] Cesare Alippi, Stavros Ntalampiras, and Manuel Roveri. Model-Free Fault Detection and Isolation in Large-Scale Cyber-Physical Systems. *IEEE Trans. Emereg. Topics Comput. Intell.* 1(1), 2017, pp. 61–71. DOI: 10.1109/TETCI.2016.2641452.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive WCET analysis. In: *Software Technologies for Embedded and Ubiquitous Systems*, vol. 6399, 2010. DOI: 10.1007/978-3-642-16256-5_6.

[Bar+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E Rydeheard. Quantified event automata: towards expressive and efficient runtime monitors. In: *FM*, pp. 68–84. 2012. DOI: 10.1007/978-3-642-32759-9_9.

[BBF09] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer* 42(10), 2009, pp. 22–27. DOI: 10.1109/MC.2009.326.

[Beh+06] Gerd Behrmann, Alexandre David, Kim G Larsen, John Håkansson, Paul Pettersson, Yi Wang, and Martijn Hendriks. UPPAAL 4.0. In: *Third International Conference on the Quantitative Evaluation of Systems*, pp. 125–126. IEEE, 2006. DOI: 10.1109/QEST.2006.59.

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An efficient algorithm for parametric WCET calculation. *Journal of Systems Architecture* 57(6), 2011, pp. 614–624. DOI: 10.1016/j.sysarc.2010.06.009.

[Ber+11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A graph query language for EMF models. In: *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pp. 167–182. 2011. DOI: 10.1007/978-3-642-21732-6_12.

[BF16] Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods in System Design* 48(1-2), 2016, pp. 46–93. DOI: 10.1007/s10703-016-0253-8. arXiv: 1111.5133.

[BFL17] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. Symbolic WCET computation. *ACM Trans. Embedded Comput. Syst.* 17(2), 2017. DOI: 10.1145/3147413.

[Bla+10] Régis Blanc, Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: algebraic bound computation for loops. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 103–118. 2010.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20(4), 2011, p. 14. DOI: 10.1145/2000799.2000800.

[BP05] Francois Bodin and Isabelle Puaut. A wcet-oriented static branch prediction scheme for real time systems. In: *17th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 33–40. 2005.

[Bro+06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In: *2006 17th International Symposium on Software Reliability Engineering*, pp. 85–94. 2006.

[Buc+14] Christian Buckl, Michael Geisinger, Dhiraj Gulati, Fran J Ruiz-Bertol, and Alois Knoll. Chromosome: a run-time environment for plug & play-capable embedded real-time systems. *ACM SIGBED Review* 11(3), 2014, pp. 36–39.

[Bur+04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with uml/rt in the fujaba real-time tool suite. In: *Proceedings of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML*, 2004.

[Bur+05] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. Worst-case execution time optimization of story patterns for hard real-time systems. In: *3rd International Fujaba Days*, pp. 71–78. 2005.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pp. 50–59. 2002.

[CDH00] Olivier Corby, Rose Dieng, and Cédric Hébert. A conceptual graph model for w3c resource description framework. In: Bernhard Ganter and Guy W. Mineau (eds.), *Conceptual Structures: Logical, Linguistic, and Computational Issues*, pp. 468–482. Springer Berlin Heidelberg, 2000.

[Che+11] Betty H. C. Cheng et al. Using models at runtime to address assurance for self-adaptive systems. In: *Models@run.time*, pp. 101–136. 2011. DOI: 10.1007/978-3-319-08915-7_4.

[Cic+17] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Romina Spalazzese. Model-Driven Engineering for Mission-Critical IoT Systems. *IEEE Software* 34(1), 2017, pp. 46–53. DOI: 10.1109/MS.2017.1.

[CJ11] Duc Hiep Chu and Joxan Jaffar. Symbolic simulation on complicated loops for WCET path analysis. *Embedded Systems Week 2011, ESWEEK 2011 - Proceedings of the 9th ACM International Conference on Embedded Software, EMSOFT'11*, 2011, pp. 319–328.

[CK96] Kong-Rim Choi and Kyung-Chang Kim. T*-tree: a main memory database index structure for real time applications. In: *3rd International Workshop on Real-Time Computing Systems and Applications*, pp. 81–88. 1996. DOI: 10.1109/RTCSA.1996.554964.

[Col+12] Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J Pace. poly-Larva: runtime verification with configurable resource-aware monitoring boundaries. In: *International Conference on Software Engineering and Formal Methods*, pp. 218–232. 2012.

[Cru+20] Jesus Gorroñogoitia Cruz, Andrey Sadovykh, Dragos Truscan, Hugo Bruneliere, Pierluigi Pierini, and Lara Lopez Muñiz. MegaM@Rt2 EU Project: Open Source Tools for Mega-Modelling at Runtime of CPSs. In: *Open Source Systems*, pp. 183–189. Springer International Publishing, 2020.

[CS06] Hugues Cassé and Pascal Sainrat. OTAWA, a framework for experimenting WCET computations. *3rd European Congress on Embedded Real-Time* (January), 2006, pp. 1–8.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems* 2(3), 1990, pp. 181–194. DOI: 10.1007/BF00365326.

[Cse+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. Viatra-visual automated transformations for formal verification and validation of uml models. In: *Proceedings 17th IEEE International Conference on Automated Software Engineering*, pp. 267–270. 2002.

[Des+15] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. Approximate synchrony: an abstraction for distributed almost-synchronous systems. In: *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II.* Springer, 2015, pp. 429–448. DOI: 10.1007/978-3-319-21668-3_25.

[DLT15] Normann Decker, Martin Leucker, and Daniela Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transfer*, 2015, pp. 1–21. DOI: 10.1007/s10009-015-0380-3.

[Dru00] Doron Drusinsky. The temporal rover and the ATG rover. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 1885, 2000, pp. 323–330.

[DRV18] István Dávid, István Ráth, and Dániel Varró. Foundations for Streaming Model Transformations by Complex Event Processing. *Software & System Modeling* 17(1), 2018, pp. 135–162. DOI: 10.1007/s10270-016-0533-1.

[Dug+12] P. S. Duggirala, T. T. Johnson, A. Zimmerman, and S. Mitra. Static and dynamic analysis of timed distributed traces. In: *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 173–182. 2012. DOI: 10.1109/RTSS.2012.69.

[EF17] Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pp. 125–135. ACM, 2017. DOI: 10.1145/3092703.3092723.

[Ehr+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamentals of algebraic graph transformation (monographs in theoretical computer science. an eatcs series). secaucus. 2006.

[ELK08] Leon Evers, Maria Eva Lijding, and Jan Kuper. Generic multi–packet communication through object serialization. In: *Proceedings of the 3rd international workshop on Middleware for sensor networks - MidSens '08*, ACM Press, 2008. DOI: 10.1145/1462698.1462703.

[Eme11] D Emery. Headways on high speed lines. In: *9th World Congress on Railway Research*, pp. 22–26. 2011.

[Erm+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, 2007.

[Fan+14] Shifeng Fang, Li Da Xu, Yunqiang Zhu, Jiaerheng Ahati, Huan Pei, Jianwu Yan, Zhihui Liu, et al. An integrated system for regional environmental monitoring and management based on internet of things. *IEEE Trans. Industrial Informatics* 10(2), 2014, pp. 1596–1605.

[Fer+08] Christian Ferdinand et al. Combining a high-level design tool for safety-critical systems with a tool for wcet analysis of executables. In: *Proc. of the 4th European Congress on Embedded Real Time Software (ERTS)*, 2008.

[FH04] Christian Ferdinand and Reinhold Heckmann. aiT: worst-case execution time prediction by static program analysis. In: Renè Jacquart (ed.), *Building the Information Society*, pp. 377–383. Springer US, 2004.

[Fis+98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: *International Workshop on Theory and Application of Graph Transformations*, pp. 296–309. 1998.

[For82] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1), 1982, pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0.

[Fou+12] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In: Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (eds.), *Model Driven Engineering Languages and Systems*, pp. 87–101. Springer Berlin Heidelberg, 2012.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering: testing model transformations. In: *Proceedings. 2004 First International Workshop on Model, Design and Validation, 2004.* Pp. 29–40. 2004.

[Gal06] Brian Gallagher. Matching structure and semantics: a survey on graph-based pattern matching. *AAAI FS* 6, 2006, pp. 45–53.

[Gar96] Vijay K. Garg. Observation of global properties in distributed systems. *Eighth IEEE International Conference on Software and Knowledge Engineering*, 1996, pp. 418–425.

[Gho+06] Arkadeb Ghosal et al. A hierarchical coordination language for interacting real-time tasks. In: *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pp. 132–141. 2006.

[GHT09] John C Georgas, André van der Hoek, and Richard N Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer* 42(10), 2009, pp. 52–60.

[Gie+03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time UML designs. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003, pp. 38–47. DOI: 10.1145/940071.940078.

[Gön+16] László Gönczy, István Majzik, Szilárd Bozóki, and András Pataricza. MDD-based design, configuration, and monitoring of resilient cyber-physical systems. *Trustworthy Cyber-Physical Systems Engineering*, 2016, p. 395.

[Got+15] Sebastian Gotz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. Adaptive exchange of distributed partial Models@run.time for highly dynamic systems. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, 2015, pp. 64–70. DOI: 10.1109/SEAMS.2015.25.

[Gub+13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (IoT): a vision, architectural elements, and future directions. *Future generation computer systems* 29(7), 2013, pp. 1645–1660.

[Gus+06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. *Proceedings - Real-Time Systems Symposium*, 2006, pp. 57–66. DOI: 10.1109/RTSS.2006.12.

[Gut+16] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Lukas Reinfurt. Comparison of iot platform architectures: a field study based on a reference architecture. In: *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1–6. 2016.

[HA16] Jiewen Huang and Daniel J Abadi. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment* 9(7), 2016, pp. 540–551.

[Har+15] Thomas Hartmann, Assaad Moawad, Francois Fouquet, Gregory Nain, Jacques Klein, and Yves Le Traon. Stream my models: reactive peer-to-peer distributed models@run.time. In: *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 80–89. IEEE, 2015. DOI: 10.1109/MODELS.2015.7338238.

[Har+17] Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. In: *The 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17)*, p. 6. 2017.

[Har+19] Thomas Hartmann, Francois Fouquet, Assaad Moawad, Romain Rouvoy, and Yves Le Traon. GreyCat: Efficient what-if analytics for data in motion at scale. *Information Systems* 83, 2019, pp. 101–117. DOI: `10.1016/j.is.2019.03.004`.

[Has+15] M. Hassanalieragh, A. Page, T. Soyata, G. Sharma, M. Aktas, G. Mateos, B. Kantarci, and S. Andreescu. Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: opportunities and challenges. In: *2015 IEEE International Conference on Services Computing*, pp. 285–292. 2015. DOI: `10.1109/SCC.2015.47`.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *Int. J. Software Tools Technol. Trans.* 17(2), 2015, pp. 143–170. DOI: `10.1007/s10009-014-0309-2`.

[HBS73] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In: *International Joint Conference on Artificial Intelligence*, pp. 235–245. 1973.

[He+17] Xiao He, Tian Zhang, Minxue Pan, Zhiyi Ma, and Chang-Jun Hu. Template-based model generation. *Software & Systems Modeling*, 2017, pp. 1–42.

[HMM13] Gergő Horányi, Zoltán Micskei, and István Majzik. Scenario-based automated evaluation of test traces of autonomous systems. In: *DECS workshop at SAFECOMP*, 2013.

[HOT89] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Processing aggregate relational queries with hard time constraints. *SIGMOD Rec.*, 1989. DOI: `10.1145/66926.66933`.

[HR09] Jörg Herter and Jan Reineke. Making dynamic memory allocation static to support wcet analysis. In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, 2009.

[HRW08] Jörg Herter, Jan Reineke, and Reinhard Wilhelm. Cama: cache-aware memory allocation for wcet analysis. In: *Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[Hu+20] Tingting Hu, Ivan Cibrario Bertolotti, Nicolas Navet, and Lionel Havet. Automated fault tolerance augmentation in model-driven engineering for cps. *Computer Standards and Interfaces* 70, 2020, p. 103424. DOI: `https://doi.org/10.1016/j.csi.2020.103424`.

[HVV07] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic search plans for matching advanced graph patterns. *Electronic Communications of the EASST* 6, 2007.

[Iqb+15] Muhammad Zohaib Iqbal, Shaukat Ali, Tao Yue, and Lionel Briand. Applying UML/MARTE on industrial projects: challenges, experiences, and guidelines. *Software and Systems Modeling* 14(4), 2015, pp. 1367–1385. DOI: 10.1007/s10270-014-0405-5.

[JSS13] Ethan K. Jackson, Gabor Simko, and Janos Sztipanovits. Diversely enumerating system-level architectures. In: *EMSOFT*, IEEE, 2013.

[JTF17] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. Runtime verification of LTL on lossy traces. In: *Proceedings of the Symposium on Applied Computing - SAC '17*, pp. 1379–1386. ACM Press, 2017. DOI: 10.1145/3019612.3019827.

[Jür03] Jan Jürjens. Developing safety-critical systems with UML. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2863, 2003, pp. 360–372. DOI: 10.1007/978-3-540-45221-8_31.

[KG93] H. Kopetz and G. Grunsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. In: *FTCS-23*, pp. 524–533. 1993. DOI: 10.1109/FTCS.1993.627355.

[KH10] Maximilian Koegel and Jonas Helming. Emfstore: a model repository for emf models. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pp. 307–308. Association for Computing Machinery, 2010. DOI: 10.1145/1810295.1810364.

[KKS12] Woochul Kang, Krasimira Kapitanova, and Sh Son. Rdds: a real-time data distribution service for cyber-physical systems. *IEEE Trans. Ind. Informat.* 8(2), 2012, pp. 393–405. DOI: 10.1109/TII.2012.2183878.

[KKZ13] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. WCET squeezing, 2013, p. 161. DOI: 10.1145/2516821.2516847.

[Koz16] V. P. Kozyrev. Estimation of the execution time in real-time systems. *Programming and Computer Software* 42(1), 2016, pp. 41–48. DOI: 10.1134/S0361768816010059.

[Kru+15] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Perv. Mob. Comput.* 17, 2015, pp. 184–206. DOI: 10.1016/j.pmcj.2014.09.009.

[KRV18] Pradeeban Kathiravelu, Peter Van Roy, and Luís Veiga. SD-CPS: software-defined cyber-physical systems. taming the challenges of CPS with workflows at the edge. *Cluster Computing* 22(3), 2018, pp. 661–677. DOI: 10.1007/s10586-018-2874-8.

[KTG14] Christian Krause, Matthias Tichy, and Holger Giese. Implementing graph transformations in the bulk synchronous parallel model. In: *Fundamental Approaches to Software Engineering*, pp. 325–339. 2014.

[Li+07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: a timing analyzer for embedded software. *Science of Computer Programming* 69(1-3), 2007, pp. 56–67.

[Lim+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE transactions on software engineering* 21(7), 1995, pp. 593–604.

[Lis14] Björn Lisper. Sweet–a tool for wcet flow analysis. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pp. 482–485. 2014.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In: *ACM SIGPLAN Notices*, vol. 30, pp. 88–98. 1995. DOI: 10.1109/43.664229.

[LS03] George Logothetis and Klaus Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. *Proceedings -Design, Automation and Test in Europe, DATE*, 2003, pp. 196–203. DOI: 10.1109/DATE.2003.1186386.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.* 78(5), 2009, pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

[Ma+12] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In: *Proceedings of the 21st international conference on World Wide Web*, pp. 949–958. 2012.

[Maj+19] 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)*, pp. 89–94. 2019.

[Mar+04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In: *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 39–49. 2004.

[Mar+20] Kristóf Marussy, Oszkár Semeráth, Aren A Babikian, and Dániel Varró. A specification language for consistent model generation based on partial models. *Journal of Object Technology*, 2020. Accepted.

[Mas+04] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. TLSF: a new dynamic memory allocator for real-time systems. In: *16th Euromicro Conference on Real-Time Systems*, pp. 79–88. 2004.

[MB15] Menna Mostafa and Borzoo Bonakdarpour. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In: *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 494–503. 2015. DOI: 10.1109/IPDPS.2015.95.

[MD15] Joseph D McDonald and Francis T Durso. A behavioral intervention for reducing postcompletion errors in a safety-critical system. *Human factors* 57(6), 2015, pp. 917–929.

[Med+15] Ramy Medhat, Borzoo Bonakdarpour, Deepak Kumar, and Sebastian Fischmeister. Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM Transactions on Embedded Computing Systems (TECS)* 14(4), 2015, pp. 1–29.

[Mer+12] Patrick O’Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *Int. J. Softw. Tools Technol. Transfer* 14(3), 2012, pp. 249–289. DOI: 10.1007/s10009-011-0198-6.

[Mit+14] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. i3QL: language-integrated live data views. *ACM SIGPLAN Notices* 49(10), 2014, pp. 417–432. DOI: 10.1145/2714064.2660242.

[Mol+18] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489.

[Mor+14] Brice Morin et al. *Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use.* Tech. rep. University of Luxembourg, 2014, p. 25.

[MP14] Stefan Mitsch and André Platzer. ModelPlex: verified runtime validation of verified cyber-physical system models. In: *Intl. Conference on Runtime Verification*, pp. 199–214. 2014. DOI: 10.1007/978-3-319-11164-3_17.

[MSV18] Kristóf Marussy, Oszkár Semeráth, and Dániel Varró. Incremental view model synchronization using partial models. *Proceedings - 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, 2018, pp. 323–333. DOI: 10.1145/3239372.3239412.

[MSW18] Mithun Mukherjee, Lei Shu, and Di Wang. Survey of fog computing: Fundamental, network applications, and research challenges. *IEEE Communications Surveys and Tutorials* 20(3), 2018, pp. 1826–1857. DOI: 10.1109/COMST.2018.2814571.

[MW16] Alexandra Mazak and Manuel Wimmer. Towards liquid models: an evolutionary modeling approach. *Proceedings - CBI 2016: 18th IEEE Conference on Business Informatics* 1, 2016, pp. 104–112. DOI: 10.1109/CBI.2016.20.

[Nen+15] Laura Nenzi, Luca Bortolussi, Vincenzo Ciancia, Michele Loreti, and Mieke Massink. Qualitative and quantitative monitoring of spatio-temporal properties. In: Ezio Bartocci and Rupak Majumdar (eds.), *Runtime Verification*, pp. 21–37. Springer International Publishing, 2015.

[Nie+15] Claus Ballegaard Nielsen, Peter Gorm Larsen, John S Fitzgerald, Jim Woodcock, and Jan Peleska. Systems of systems engineering: basic concepts, model-based techniques, and research directions. *ACM Comput. Surv.* 48(2), 2015, p. 18.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In: *ICSE 2000*, pp. 742–745. 2000. DOI: 10.1145/337180.337620.

[Obj19] Object Management Group. DDS for eXtreamly Resource-Constrained Environments (DDS-XRCE). 2019. URL: `https://www.omg.org/spec/DDS-XRCE/`.

[OS95] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: a survey. *IEEE Trans. Knowl. Data Eng.* 7(4), 1995.

[Par03] Gerardo Pardo-Castellote. OMG Data-Distribution Service: architectural overview. In: *Proc. 23rd Int. Conf Distrib. Comput. Syst. Workshops*, 2003.

[Pet+14] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Scaling parallel rule-based reasoning. In: *ESWC*, pp. 270–285. 2014. DOI: `10.1007/978-3-319-07443-6_19`.

[Pik+10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: a hard real-time runtime monitor. In: *LNCS*, vol. 6418, pp. 345–359. 2010. DOI: `10.1007/978-3-642-16612-9_26`.

[Pua06] Isabelle Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. *Proceedings - Euromicro Conference on Real-Time Systems* 2006, 2006, pp. 217–226.

[Qui+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In: *ICRA workshop on open-source software*, vol. 3, p. 5. 2009.

[Rie17] Leanna Rierson. *Developing Safety-Critical Software*. CRC Press, 2017, pp. 22–27. DOI: `10.1201/9781315218168`.

[Rus01] John Rushby. Bus architectures for safety-critical embedded systems. In: Thomas A. Henzinger and Christoph M. Kirsch (eds.), *Embedded Software*, pp. 306–323. Springer Berlin Heidelberg, 2001.

[Rus08] John Rushby. Runtime certification. In: *International Workshop on Runtime Verification*, pp. 21–35. 2008.

[SBK06] Roberto Solis, Vivek Borkar, and PR Kumar. A new distributed time synchronization protocol for multihop wireless networks. In: *Proceedings of the 45th IEEE Conference on Decision and Control*, pp. 2734–2739. 2006.

[Sem+20] 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), 2020, pp. 57–78.

[Sha+10] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A runtime model based monitoring approach for cloud. In: *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 313–320. 2010.

[SL18] Bin Shao and Yatao Li. Parallel processing of graphs. In: *Graph Data Management*, pp. 143–162. Springer, 2018.

[SNV18] Oszkár Semeráth, András Szabolcs Nagy, and Dániel Varró. A graph solver for the automated generation of consistent domain-specific models. In: *40th International Conference on Software Engineering*, pp. 969–980. 2018.

[Sob52] Bolesław Sobociński. *Axiomatization of a partial system of three-value calculus of propositions.* Institute of Applied Logic, 1952.

[Som+13] Stephan Sommer, Alexander Camek, Klaus Becker, Christian Buckl, Andreas Zirkler, Ludger Fiege, Michael Armbruster, Gernot Spiegelberg, and Alois Knoll. Race: a centralized platform computer based architecture for automotive applications. In: *2013 IEEE International Electric Vehicle Conference (IEVC)*, pp. 1–6. 2013.

[SS07] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In: *Proceedings of the 7th ACM & IEEE EMSOFT '07*, p. 203. 2007.

[Sta18] International Organization for Standardization. ISO 26262: Road Vehicles – Functional Safety. Second edition. 2018.

[Ste+08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[SV17] Oszkár Semeráth and Dániel Varró. Graph constraint evaluation over partial models by constraint rewriting. In: *ICMT 2017*, pp. 138–154. 2017. DOI: `10.1007/978-3-319-61473-1_10`.

[SVV16] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. In: *International Conference on Fundamental Approaches to Software Engineering*, pp. 87–103. 2016.

[SZ13] Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling* 15(1), 2013, pp. 31–69. DOI: `10.1007/s10270-013-0394-9`.

[Szá+14] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. IncQuery-D: a distributed incremental model query framework in the cloud. In: *International Conference on Model Driven Engineering Languages and Systems*, pp. 653–669. 2014.

[Szá+17] Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The Train Benchmark: cross-technology performance evaluation of continuous model queries. *Software and Systems Modeling*, 2017, pp. 1–29. DOI: 10.1007/s10270-016-0571-8.

[Szt+12] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, B. Goodwine, and J. Baras. Toward a science of cyber-physical system integration. *Proceedings of the IEEE* 100(1), 2012, pp. 29–44. DOI: 10.1109/JPROC.2011.2161529.

[Szt+14] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. Openmeta: a model- and component-based design tool chain for cyber-physical systems. In: *From Programs to Systems. The Systems perspective in Computing: ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings.* Springer Berlin Heidelberg, 2014, pp. 235–248. DOI: 10.1007/978-3-642-54848-2_16.

[Tei+94] Juergen Teich, Shruva Sriram, Lothar Thiele, and Michael Martin. Performance analysis of mixed asynchronous synchronous systems. In: *Proceedings of 1994 IEEE Workshop on VLSI Signal Processing*, pp. 103–112. 1994.

[The14] The Object Management Group. Object Constraint Language, v2.4. 2014. URL: https://www.omg.org/spec/OCL/2.4.

[Tót+14] Tamás Tóth et al. Verification of a real-time safety-critical protocol using a modelling language with formal data and behaviour semantics. In: *Computer Safety, Reliability, and Security*. 2014, pp. 207–218.

[TR96] Juha Taina and Kimmo Raatikainen. Rodain: A real-time object-oriented database system for telecommunications. In: *ACM International Conference on Information and Knowledge Management*, vol. Part F129290, pp. 10–14. 1996. DOI: 10.1145/352302.352306.

[Uet+16] Kosei Ueta, Xiaoyong Xue, Yukikazu Nakamoto, and Sena Murakami. A distributed graph database for the data management of iot systems. In: *Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE*

Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on, pp. 299–304. 2016.

[Ujh+15] Zoltán Ujhelyi et al. EMF-IncQuery: an integrated development environment for live model queries. *Sci. Comput. Program.* 98, 2015, pp. 80–99.

[Var+15] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An algorithm for generating model-sensitive search plans for pattern matching on EMF models. *Software & Systems Modeling*, 2015, pp. 597–621. DOI: 10.1007/s10270-013-0372-2.

[Var+16] Dániel Varró et al. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & System Modeling*, 2016. DOI: 10.1007/s10270-016-0530-4.

[Var+18] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In: *Graph Transformation, Specifications, and Nets (In Memory of Hartmut Ehrig).* 10800. 2018.

[VAS12] Gergely Varró, Anthony Anjorin, and Andy Schürr. Unification of compiled and interpreter-based pattern matching techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7349 LNCS, 2012, pp. 368–383.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3), 2007, pp. 214–234. DOI: 10.1016/j.scico.2007.05.004.

[VG14] Thomas Vogel and Holger Giese. Model-driven engineering of self-adaptive software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* 8(4), 2014, p. 18. DOI: 10.1145/2555612.

[Vie+16] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. Reminds: A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software* 112, 2016, pp. 123–136.

[War92] David S. Warren. Memoing for logic programs. *Commun. ACM* 35(3), 1992, pp. 93–111. DOI: 10.1145/131295.131299.

[Wil+08] Reinhard Wilhelm et al. The determination of worst-case execution times: overview of the methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7(3), 2008, 36:1–36:53.

[Wil+10] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In: *Verification, Model Checking, and Abstract Interpretation*, 2010.

[ZDG09] Haitao Zhu, Matthew B. Dwyer, and Steve Goddard. Predictable runtime monitoring. *Proceedings - Euromicro Conference on Real-Time Systems* (2), 2009, pp. 173–183.

[Zha+15] Ben Zhang et al. The cloud is not enough: saving IoT from the cloud. In: *7th USENIX Workshop on Hot Topics in Cloud Computing*, 2015.

[Zhe+16] Xi Zheng, Christine Julien, Rodion Podorozhny, Franck Cassez, and Thierry Rakotoarivelo. Efficient and Scalable Runtime Monitoring for Cyber–Physical System. *IEEE Systems Journal*, 2016, pp. 1–12. DOI: 10.1109/JSYST.2016.2614599.

Appendices

Appendix A

Definitions of Safety Properties in VQL

The safety monitoring goals presented in Listing A.1 use the concepts presented in Figure 3.2a (the MoDeS3 metamodel).

```
pattern trainLocations(train : Train, location : Segment) {
  Train.location(train, location);
}

pattern endOfSiding(train : Train, location : Segment) {
  Segment.occupiedBy(location, train);
  Segment.connectedTo(end, location);
  1 == count find connected(end, _);
}

pattern closeTrains(start : Segment, end : Segment) {
  Train.location(_train, start);
  Segment.connectedTo(start, middle);
  Segment.connectedTo(middle, end);
  start != end;
  Segment.occupiedBy(end, _otherTrain);
}

pattern misalignedTurnout(location : Segment, train : Train) {
  Turnout(turnout);
  Segment.occupiedBy(location, train);
  Turnout.straight(turnout, location);
  neg find connected(location, turnout);
} or {
  Turnout(turnout);
  Segment.occupiedBy(location, train);
  Turnout.divergent(turnout, location);
  neg find connected(location, turnout);
}

private pattern connected(a : Segment, b : Segment) {
  Segment.connectedTo(a, b);
}
```

Listing A.1: The definitions of queries taken from the MoDeS3 domain using VQL syntax

Appendix B

Proof Sketches

Proposition 1. For a query program q, theories $\mathcal{T}, \mathcal{T}'$, and model scopes $\mathcal{S}, \mathcal{S}'$ the following inequality holds:

$$\text{If } \mathcal{T}' \supseteq \mathcal{T} \text{ and } \forall \mathtt{C}_i \in \Sigma\colon \mathcal{S}'(\mathtt{C}_i) \subseteq \mathcal{S}(\mathtt{C}_i) \text{ then } \mathit{WCET}^{s}_{\mathtt{q}}(\mathcal{T}', \mathcal{S}') \leq \mathit{WCET}^{s}_{\mathtt{q}}(\mathcal{T}, \mathcal{S}).$$

Proof. (Sketch.) Let $\mathcal{M} = \{M : \mathcal{T}, \mathcal{S} \models M\}$ be the set of well-formed models in the model scope. It is sufficient to show that $\mathcal{M}' = \{M : \mathcal{T}', \mathcal{S}' \models M\} \subseteq \mathcal{M}$ since the witness model $M^* \in \mathcal{M}$ provides the longes estimated execution. Therefore, we have to consider the following two cases: (1) $\mathcal{T}' \supsetneq \mathcal{T}, \mathcal{S}' = \mathcal{S}$ and (2) $\mathcal{T}' = \mathcal{T}, \mathcal{S}' \subsetneq \mathcal{S}$

1. Assume $\mathcal{T}' \supsetneq \mathcal{T}, \mathcal{S}' = \mathcal{S}$, i.e., there is at least one additional WF constraint added to the theory of WF constraints, but the scope remains the same. The addition of a new WF constraint cannot invalidate existing constraints, i.e., $\forall M : \mathcal{T}', \mathcal{S} \models M \rightarrow \mathcal{T}, \mathcal{S} \models M$.
2. Assume $\mathcal{T}' = \mathcal{T}, \mathcal{S}' \subsetneq \mathcal{S}$, i.e., there is at least one $\mathtt{C}_i \in \Sigma$ such that $\mathcal{S}'(\mathtt{C}_i) \subsetneq \mathcal{S}(\mathtt{C}_i)$ and the theory of WF constraints remains the same. For the witness model $\mathcal{T}, \mathcal{S} \models M^*$, it is true that $\mathit{stats}_{M^*}(\mathtt{C}_i) \in \mathcal{S}(\mathtt{C}_i)$. If $\mathit{stats}_{M^*}(\mathtt{C}_i) \notin \mathcal{S}'(\mathtt{C}_i)$, the witness model $\mathcal{T}, \mathcal{S}' \models M^{**}$ need to yield a lower WCET estimate, otherwise it would have been included in the optimal solution using $\mathcal{T}, \mathcal{S}$.

□

Proposition 2. The following inequality holds between execution times and their estimates:

$$\mathit{RT}_{\mathtt{q}}(M) \leq f_{\mathtt{q}}(M) \leq \mathit{WCET}^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{\mathit{stats}_M}) \leq \mathit{WCET}^{o}_{\mathtt{q}}(\mathit{stats}_M),$$

where $\widehat{stats_M}(\mathtt{C}_i) = [stats_M(\mathtt{C}_i), stats_M(\mathtt{C}_i)]$ is the scope corresponding exactly to the model statistics $stats_M$.

Proof. (Sketch.) We show that the following three inequalities hold:

1. $RT_{\mathtt{q}}(M) \leq f_{\mathtt{q}}(M)$
2. $f_{\mathtt{q}}(M) \leq WCET^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M})$
3. $WCET^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M}) \leq WCET^{o}_{\mathtt{q}}(stats_M)$

1. The function $f_{\mathtt{q}}$ precisely counts the BB executions of the query program q over model M, and multiplies this number by the execution time of the BB. Furthermore, we use the longest possible estimated execution times of BBs when defining $f_{\mathtt{q}}$. Therefore, $RT_{\mathtt{q}}(M) \leq f_{\mathtt{q}}(M)$ holds.

2. The definition of $WCET^{s}_{\mathtt{q}}$ is to compute the value of $f_{\mathtt{q}}$ for the witness model M^*, which maximizes the value returned by this function. This means that for any model with the same statistics as M, $f_{\mathtt{q}}(M) \leq WCET^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M})$ holds.

3. The formula which defines $WCET^{o}_{\mathtt{q}}(stats_M)$ sums BB execution times based on program control flow, and $stats_M$ provides the flow facts for setting maximum loop bounds. These flow facts inherently overestimate execution counts in cases where there is a variation in actual loop repetitions since it will assume the number of maximum repetitions. On the contrary, $WCET^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M})$ will precisely count how many times a BB is executed if a query is evaluated over model M. This proves that the inequality $WCET^{s}_{\mathtt{q}}(\mathcal{T}, \widehat{stats_M}) \leq WCET^{o}_{\mathtt{q}}(stats_M)$ holds.

□