University of Isfahan

Faculty of Computer Engineering

Department of Software Engineering

PhD Thesis

Techniques to Compact Model Execution Traces in Model Driven Approach

Supervisor:

Dr. Bahman Zamani

Advisor:

Prof. Abdelwahab Hamou-Lhadj

By:

Fazilat Hojaji

July 2019

Techniques to Compact Model Execution Traces in Model Driven Approach

A Thesis

by

Fazilat Hojaji

Submitted to the Office of Graduate and Studies of
University of Isfahan
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

Committee Members,	Dr. Kamran Zamanifar
	Dr. Shekofeh Kolahdooz Rahimi
Head of Department,	Dr. Behroz Tork Ladani

July 2019

Faculty of Computer Engineering

ABSTRACT

Model-Driven Engineering (MDE) is a development paradigm that aims at coping with the complexity of systems by separating concerns using models. A model is a representation of a particular aspect of a system, and defined using a Domain-Specific Modeling Language (DSML). A subclass of DSMLs aim at supporting the execution of models, namely executable Domain-Specific Modeling Languages (xDSMLs). An xDSML includes execution semantics that manipulate the concepts of the considered domain. To ensure that an executable model is correct with regard to its intended behavior, dynamic Verification and Validation (V&V) techniques are used to verify the behavior of software systems early in the design process. Yet, existing V&V techniques mainly rely on *execution trace*s to model and analyze the behavior of executable models. An execution trace is a sequence containing all the relevant information about an execution over time. Traces, however, tend to be overwhelmingly large, making it difficult to analyze the recorded behavior. There exist *trace metamodels* to represent execution traces, but most of them suffer from scalability problems. Furthermore, existing model execution tracing approaches rely on their own custom trace formats, hindering interoperability and sharing of data among various trace analysis tools. The goal of this thesis is to fill this gap and provide a common trace exchange format for traces, designing scalable trace representations that enables the construction and manipulation of execution traces, obtained from executable models.

The *first contribution* of this thesis comprises a systematic mapping study on existing approaches for tracing executable models. With this study we aim at identifying and classifying the existing approaches, thereby assessing the state of the art in this area, as well as pointing to promising directions for further research in this area.

The *second contribution* consists in a generic compact trace representation format called Compact Trace Metamodel (CTM) that enables the construction and manipulation of execution traces, obtained from executable models. Compared to existing trace metamodels, the results show signif-

icant reduction around %59 in memory and %95 disk consumption. Also, the performance overhead required to the CTM trace construction is small enough around %10 that makes it practically applicable. Moreover, CTM offers a common structure, which allows interoperability between existing trace analysis tools.

Dedication

To my parents for their love, endless support and encouragement.

Acknowledgement

This work would not have been possible without the support and encouragement of the others.

I am extremely indebted to my supervisor, **Dr. Bahman Zamani**, for his teaching, supervision, and patience during doing this thesis. I could not have imagined having a better mentor for my Ph.D study.

I would like to express my sincere gratitude to my advisor **Prof. Abdelwahab Hamou-Lhadj**, the associate professor in the Department of Electrical and Computer Engineering at Concordia University of Montreal, for the continuous support of my PhD study and research.

Besides, I wish to thank **Dr. Tanja Mayerhofer** and **Dr.Erwan Bousse**, the members of Business Informatics Group at the Vienna University of Technology, for their continued support, encouragement, and insightful comments during my sabbatical leave and after that.

I would like to thank my family: my parents for giving birth to me at the first place, and supporting me spiritually throughout my life.

I would like thank my lab mates in our **MDSE Research Group** for their continued collaboration and support.

Last but not least, deepest thanks go to all people who took part in making this thesis real.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

Chapter 1

Introduction

1.1 Context

In software engineering, abstraction is a key enabler to deal with the complexity of software systems. Model Driven Development (MDD) is a software development paradigm that aims to decrease the complexity of software systems by raising the level of abstraction in the development process through the use of models and well-defined modeling languages [1]. In this paradigm, models are the key artifacts in the software development process, and are used to specify the structure and behavior of the system to be built. One of the main purposes of models is to analyze quality properties of complex systems, for instance to explore design alternatives or to identify potentials for improving systems. The analysis includes functional and non-functional properties, as well as structural and behavioral aspects of systems.

To ensure that behavioral models are correct concerning their intended behavior, early dynamic Verification and Validation (V&V) techniques are required. These techniques are based on the ability to *execute* models. To this end, efforts have been made to support the execution of models.

It can be achieved by defining execution semantics of Domain-Specific Modeling Languages (DSMLs) precisely. Such languages are called executable DSMLs (xDSMLs) that support the execution of models, and enable the use of dynamic V&V techniques. Moreover, providing executability at the model level also gives the possibility to directly deploy an executable model to run on a production system.

1.2 Problem

Defining execution semantics of modeling languages gives the possibility to execute models, and hence to use dynamic V&V to check the modeled behavior. Yet, many dynamic V&V techniques require an analysis of behavior over time, which requires capturing execution traces. Execution traces can be generated during the execution of a model, and provide information to help reason about the model's execution behavior. Traces can contain different kind of information depending on the defined structure for execution traces and the purpose of the trace.

To support dynamic V&V for xDSMLs, a data structure is required to capture, store, and analyze traces. However, the problem is that even with using an appropriate trace structure that adequately represents the execution behavior of a model, executing a model might lead to a very large execution trace, making it difficult to analyze the recorded behavior [2, 3, 4].

Furthermore, existing model execution tracing approaches rely on their own custom trace formats, hindering interoperability and sharing of data among various trace analysis tools. Consequently, there is a need to work towards a common format for exchanging model execution traces. A common format must be generic, to be able to support a wide range of xDSMLs, independent of the meta-programming approaches used for their implementation. It also must be scalable and expressive enough to capture the required runtime information.

The first requirement, genericity, can be partly addressed using existing generic trace metamodels such as the ones defined and presented by Hartmann et al. [5] and Langer et al. [6]. While these formats allow interoperability between existing trace analysis tools and simplify analyzing traces, they do not scale up to large traces efficiently. For example, the approach proposed by Langer et al. [6] relies on a generic clone-based execution trace metamodel, which defines a *trace* as a sequence of *step* and *state* elements. Such trace contains all the reached execution states as a sequence of complete model clones, which yields poor scalability in memory. Only a few trace structures, such as the ones proposed by Bousse et al. [7], consider scalability by providing some sort of trace compaction. However, these techniques still require substantial memory usage due to data redundancy. Also, they do not give a complete representation of a trace such as execution

states as well as input and output values, hindering expressiveness.

To summarize, the following are the two main inter-related challenges that should be considered for designing a new execution trace structure:

> **Ch#1**: The structure of trace should be *generic* so that it can be supported by any possible executable modeling language. Hence, it should capture all the necessary information during execution of a model.
>
> **Ch#2** : The exploration and understanding a large execution trace can be difficult due to the size of traces. This requirement consists of generating execution trace in a *scalable* format when manipulating traces.

We tackle the aforementioned challenges by developing a new trace metamodel called Compact Trace Metamodel (CTM) which 1) precisely captures the execution trace of models conforming to any possible modeling language and 2) represents traces in a compact form without lossing data.

1.3 Aim of the Research

To tackle the aforementioned challenges, we investigate two complementary directions. we first propose a generic trace metamodel which enables us to capture a complete trace of any xDSML. It is defined by identifying a set of key generic concepts needed to express traces generated from any xDSML. Such way of doing brings advantages regarding genericity (Ch#1), since the data structure of the execution trace is simple and appropriate for generic manipulations. Moreover, comparing to existing trace structures, a more expressive trace is generated, which captures the required runtime information for any executable modeling language. This means, that traces are expected to grow large, compromising the need for scalability in space (Ch#2). Thereby, scalability should be considered as a key requirement when defining a common trace structure. To cope with this problem, we propose a compact trace metamodel that is applicable to any executable modeling language while reducing the size of traces. The key idea is to compact repetitive parts of a trace.

Therefore, to reduce vast size of traces, several compaction techniques are effectively applied to the generic trace metamodel, tailored to compact the respective parts of the trace, which lead to representation of the trace with minimal redundancy. The compaction is done so that the original trace can be fully constructed from the compact one. Note that what we mean by compaction is different from the common compression techniques found in the information theory. A compressed file needs to be uncompressed before usage. A compacted file should never be "uncompacted". In other words, we need to find a way to represent trace information by changing the structure of the data. An example of a trace format is CTF (Compact Trace Format) [4] which represents traces of routine calls as directed acyclic graphs. In this research, we propose CTM as a metamodel for representing lossless execution traces in a more scalable format.

1.4 Research Methodology

In this research, we have followed the *design science research methodology (DSRM)* presented by Peffers et al. [8] align with the guidelines for design science defined by Hevner et al. [9]. The DSRM approach consists of six main activities, which we present in each section of this thesis in detail. From a top level methodological perspective, we utilize different research techniques at each step, and perform some activities to appropriately support our overall objectives. The following are the activities we have followed in this research, which form the structure of this thesis.

Problem identification and motivation: We focused on the problem space, provided a background around the target domain, and made an overview of the techniques used for the trace compaction as well. We also did a literature review on the existing approaches to find their strengths and limitations.

Define the objectives of a new solution: We identified the main requirements for a trace metamodel, and defined our overall objective for the design of a new trace metamodel.

Design and development: We designed CTM for tackling aforementioned challenges, and based on the identified requirements.

Demonstration: We constructed a detailed implementation of CTM-enabling tools.

Evaluation: We evaluated the effectiveness of CTM to capture traces from models of five different xDMSLs. We measured the gain obtained by storing CTM traces compared to the use of the metamodel proposed by Bousse et al. [10, 7].

1.5 Structure of the thesis

Figure 1.1 shows an overview of the structure of the thesis. We present the different chapters thereafter.

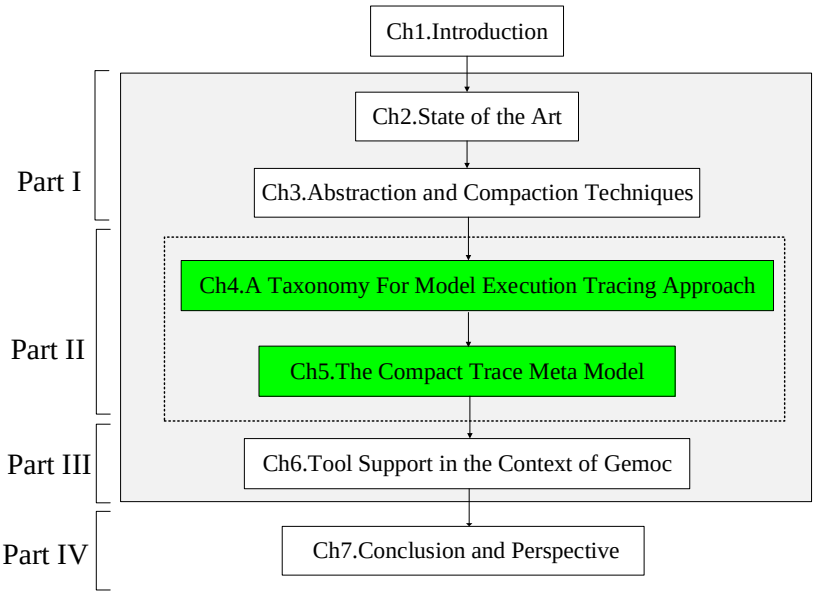

Figure 1.1: Graph of the outline of the thesis. Chapters in green contain the core of the scientific contributions.

Part I - Foundations

Chapter 2 introduces the state of the art of MDD, executable metamodeling, model execution, and execution trace. Finally, we focus more specifically on execution trace data structures, and

discuss the existing trace structures concentrating on compaction techniques.

Chapter 3 provides an overview of the techniques for dealing with the large size of data in three domains containing code-centric development, model driven development, and data management.

Part II - Contributions

Chapter 4 presents our first contribution, which is a taxonomy for model execution tracing approaches. We give a detail description about the systematic mapping study that we have done in the literature, and provided a classification on the existing approaches.

Chapter 5 deals with the second contribution. We discuss the approach for designing our new trace metamodels. Then, we introduce the generic trace metamodel supported by any xDSML. Continuing, we present the extension of the generic trace metamodel provided by using the compaction techniques incorporated in that. Lastly, we discuss related work.

Part III - Applications and Tooling

Chapter 6 deals with the implementation and evaluation of CTM in the context of Eclipse Gemoc Studio[1]. We present an overview of the software development endeavor that was done during this thesis, either to improve existing tools or to implement our approaches and applications. In particular, it is concerned with the integration of our work within the Gemoc Studio, which is a language and modeling workbench. We explain the generation of the trace metamodels in Eclipse Modeling Framework(EMF), and construction of traces by applying the compaction techniques. For each part of the trace, the corresponding techniques are applied. Continuing, we evaluate CTM by defining several research questions, measuring evaluation criteria, and comparing the results with the existing trace metamodels.

Part IV - Conclusion and Perspectives

Chapter 7 concludes the thesis by summarizing the advances that it brings to generate a more scalable execution trace for any xDSMLs. We end by discussing the perspectives of future research on the topic.

[1] http://gemoc.org/studio

Part I

Foundations

Chapter 2

State of the Art

This chapter introduces the basic background and related work that help the reader understand the problem and the solution that are described in this thesis. We present the state of the art in the different domains covered by our contributions and applications. In Section 2.1, we first introduce MDD by defining a number of its fundamental concepts. Then, in Section 2.2, we provide the definition of executable models, and focus more specifically on xDSMLs, and give an example of an xDSML. In Section 2.3, we define the concept of execution trace, and describe the application of model execution traces. Finally, in Section 2.4, we focus more particularly on execution trace data structures, and discus the existing trace structures concentrating on compaction techniques.

2.1 Model-Driven Development

In most of the engineering disciplines, models are necessary specially for designing a complex system. Nowadays, software systems are becoming more and more complex; hence, using models for developing software systems is unavoidable [11].

MDD is a development paradigm that uses models as the main artifacts of the development process [1]. For this purpose, two main kinds of modeling languages are used: General Purpose Modeling Languages (GPMLs), such as UML, that can be used for modeling systems regardless of the domain, and Domain-Specific Modeling Languages (DSMLs) that are each designed particularly for specific tasks in a given domain [1]. One main purpose of models is to analyze quality properties of complex systems, for instance to explore design alternatives or to identify potentials for improving systems. This includes checking both functional and non-functional properties,

which concerns both structural and behavioral aspects of systems. In the case of behavioral aspects, dynamic V&V techniques are used to check properties, which necessitate the ability to *execute* models. To this end, many efforts have been made to support the execution of models, such as methods to ease the development of executable DSMLs (xDSMLs)[1] [12, 13, 14, 15, 16], or to support the execution of UML models [17]. This endeavor includes both facilitating the definition of the execution semantics of modeling languages, and the development of dynamic V&V methods that can be used with these executable languages.

2.1.1 Metamodel

The standard way for specifying the syntax is by defining a *metamodel*. Many definitions of *metamodel* can be found in the literature: "a model to model modeling" [18], "a textual, graphical, and/or formal representation of the concepts and how they are linked" [1]. Therefore, a metamodel is essentially composed of classes, each being composed of properties. In addition, a metamodel possesses static semantics, which are additional structural constraints that must be satisfied by conforming models. We consider a *metamodel* to be an object-oriented model composed of classes, attributes, and relationships between these classes for defining the concepts of modeling [1].

Definition 1. *A metamodel is compose of :*

- *A set of metaclasses that are used to define the concepts of a specific domain and the relationships between them.*
- *Static semantics that define the structural meaning of a language by using a set of OCL rules and containment references.*

As an example, the metamodel of the Petri net language is depicted in Figure 2.1. It consists of three classes: Net, Transition, and Place. The metaclass Net is composed of a set of places (metaclass Place) and transitions (metaclass Transition) by using containment references. The

[1]Note that techniques to develop executable DSMLs are applicable to executable modeling language in general, including executable GPMLs. In the remainder of this report, we will use the term xDSML including both executable modeling languages and executable GPMLs.

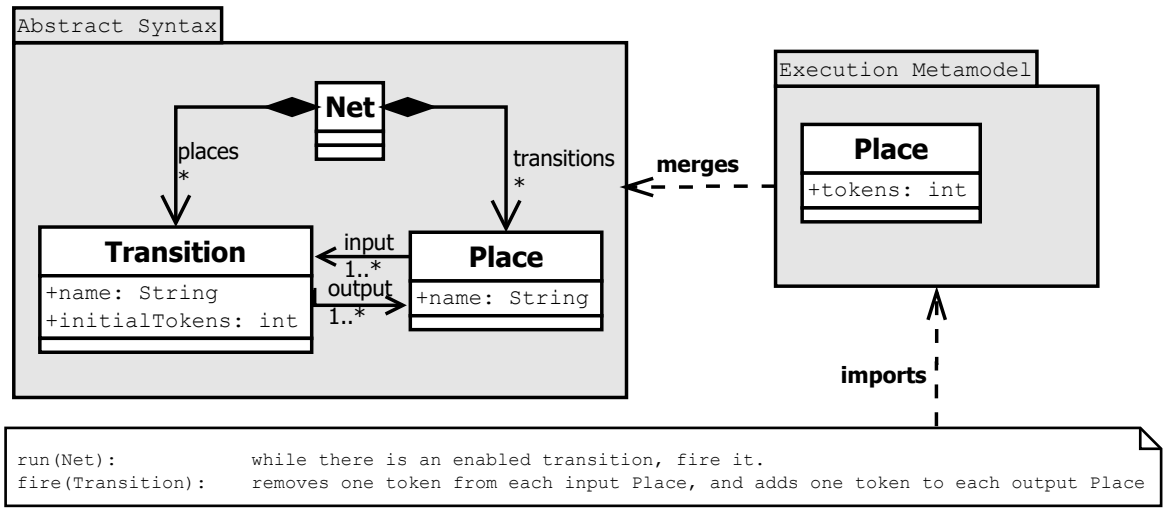

Figure 2.1: Petri net abstract syntax

class Transition has two references *input* and *output* pointing to the class Place and an attribute `name`. A place has an attribute `name` and an attribute `tokens` for specifying the number of tokens.

2.1.2 Model

We consider a model as a set of objects (i.e., instances of metamodel classes) that represents a system. The objects are instances of the classes defined in the metamodel, pointing to the conformity relationship between a model and its metamodel. A model is also static semantics of the metamodel. Each object has a set of attributes that represent the values of the properties of the corresponding class.

Figure 2.2 shows a concrete syntax of the model represented by the Petri net notation, which is conformed to the metamodel shown in Figure 2.1. The model consists of one instance of the Net class, four instances of the Place class, and two instances of the Transition class. The `initialTokens` field of *p1* and *p2* is one and for *p3*, and *p4* is zero, meaning that at the beginning of the execution, there exists one token for each one of *p1* and *p2*, and no token for *p3* and *p4*. Figure 2.3 illustrates the object diagram that shows all objects of the model and their relationships.

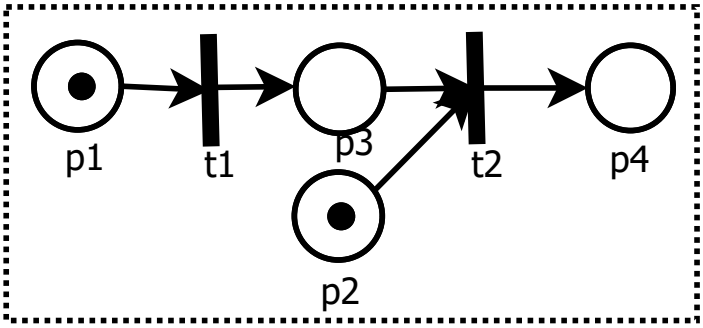

Figure 2.2: Example of Petri net model represented with concrete syntax

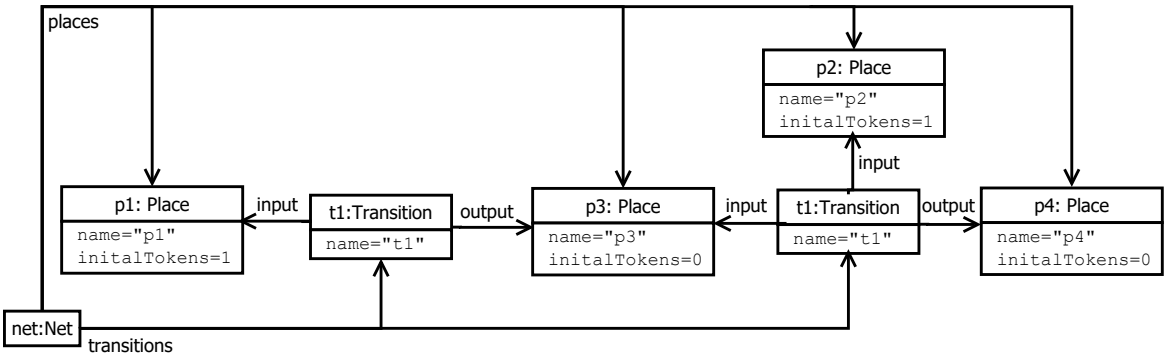

Figure 2.3: Example of Petri net model represented as an object diagram

2.1.3 Model Transformation

In MDD, models are not only used for describing a system, but also for analyzing static inconsistencies or defects. In addition, there are several activities that can be performed in an automated way to change or create a model. This is accomplished by applying model transformations, which are at the core of MDD [19]. Model transformation is the process of converting one or more source models into one or more target models. Model transformations can be defined using many paradigms, such as declarative programming (e.g., ATL [20]), imperative programming (e.g., Xtend/EMF, Kermeta [21]) or triple graph grammars (e.g., [22]). They are also used for defining

operational semantics. Model transformations are used for several purposes such as refactoring a model, reverse engineering, and generation of a new model based on an existing one.

A model transformation is composed of transformation rules, each defining a subset of the changes on the target model that provide the execution state of the model. Furthermore, there exist several types of model transformations. If both source and target models are expressed in the same metamodel, it is called *Endogenous* model transformation. Otherwise, it is named *Exogenous*. A specific kind of *Endogenous* model transformation is in-place transformation that directly changes source models without creating new target models.

An example of an in-place model transformation is the transformation rule *fire* defined by using Kermeta depicted in Listing 2.1.1. For this transformation rule, the input is a specific Transition object (line 1). It checks if there are sufficient tokens in all of its input Places, it removes one token from all input Places (lines 4-5), and adds one token in its output Places (lines 7-8).

```
1 def void fire() {
2   if (_self.isEnabled) {
3     // Removes a token from each input place
4     for (Place input : _self.input)
5        input.tokens = input.tokens - 1
6     // Adds a token to each output place
7     for (Place output : _self.output)
8        output.tokens = output.tokens + 1
9    }
10 }
```

Listing 2.1.1: Definition of the execution metamodel of Petri net through a Kermeta aspect

2.2 Model Execution

2.2.1 Execution semantics

Execution semantics specifies the execution behavior of models. In MDE, there exist two different approaches for defining the execution semantics of modeling languages: the translational semantics approach and the operational semantics approach. These two approaches have similarity with the denotational semantics approach, which is defined in the field of programming language design for algebraic/mathematical terms. In the following, we define both approaches and explain

their advantages and disadvantages.

Operational approach [23, 24]. In this approach, the execution behavior of models conforming to an executable modeling language is defined by an interpreter (a.k.a. virtual machine). Such an interpreter is an endogenous in-place model transformation in which models are modified directly to carry forward their execution by performing transitions from one execution state to the next one.

Translational semantics [25, 26]. In this approach, the model is translated into another executable language for execution. In the translation, the concepts of the source language are translated into the concepts of the target language. Thereby, the translation from the source language to the target language composes of the semantic mapping of the semantics definition. This can be done through exogenous model transformation or through code generation if the target language possesses a grammar.

In summary, the translational approach proposes the implementation of compilers while the operational approach proposes interpreters for modeling languages. In this thesis, we only consider operational semantics for the definition of the execution semantics of xDSMLs. Therefore, the term xDSML only refers to xDSMLs defined using operational semantics.

2.2.2 xDSML

xDSMLs are a specific kind of DSMLs that support the execution of models, and enable the use of dynamic V&V techniques, which involve controlling the execution of a model [10]. We call *executable model* a model conforming to an executable modeling language, and we define such models as follows.

Definition 2. *An* executable model *is a model conforming to an xDSML. It defines an aspect of the behavior of a system in sufficient detail to be executed.*

To support the execution of models, an xDSML must provide *execution semantics*, which is defined in two different ways: the translational semantics approach and the operational semantics approach. Both of these approaches were defined in Section 2.2.1.

The execution state of a model is defined by extending the abstract syntax of the xDSML and adding new properties and classes, which are so-called run-time concepts of the xDSML. The result of this extension is called execution metamodel [27]. We consider the additional properties and classes included in the execution metamodel dynamic as they can be changed during the execution of the model. The dynamic properties can be changed and new dynamic metaclasses can be instantiated. These modified values represent the execution state of the model. The execution state of a model changes over time by the definition of an in-place transformation, whose input and output is a model conforming to the execution metamodel. The transformation is accomplished through a set of transformation rules, each defining a subset of the changes performed on the execution state.

Finally, in order to execute a model, each object of the model should be translated into an executable object in the execution metamodel. For this work, the initialization function transforms the model to a model conforming to the execution metamodel.

Definition 3. *An xDSML is defined by:*

- *An abstract syntax, which is a metamodel.*
- *An execution metamodel, which is an extension of the abstract syntax with additional classes and properties by using package merge. This metamodel defines the execution state of executed models.*
- *Operational semantics, which includes an execution transformation that modifies a model conforming to the execution metamodel by changing values of dynamic fields and by creating/destroying instances of classes introduced in the execution metamodel.*
- *An initialization function, which is an in-line model transformation that transforms a model conforming to the abstract syntax into a model conforming to the execution metamodel.*

2.2.3 Execution Metamodel

The first part of an xDSML is the execution metamodel that defines the execution state of a model conforming to the xDSML. “An execution state is the set of the values of all dynamic fields

of a model at a certain point in time of the execution" [27]. The definition of execution state is related to the abstract syntax. For example, when an xDSML includes a variable as a concept in its abstract syntax, an execution state can include the values of all variables of the model. In this case, we can define a link from the variable of the abstract syntax to the value. In the literature, there are several approaches which define execution states of an xDSML through the extension of the abstract syntax. It can be accomplished by adding new properties and/or new classes to the abstract syntax. The result is the execution metamodel. Using package **merge**, the two metamodels are linked together.

Figure 2.1 shows an example of a Petri net xDSML. At the top left, its abstract syntax is shown with three classes **Net**, **Place** and **Transition**. At the top right is the execution metamodel by extending the class **Place** with a new property using package *merge*. The `tokens` property is declared in the existing **Place** class which defines the current number of tokens of a Place during an execution.

Two rules *run* and *fire* are defined in the operational semantics to change the execution state of a model conforming to the execution metamodel of a Petri net. The rule *run* repeatedly checks for an enabled **Transition**. In the *fire* rule, one token from each input **Place** of an enabled transition is removed, and one token is added to each of its output **Places**.

Listing 2.2.1 shows the execution transformation for the Petri net xDSML using Kermeta aspects. It relies on the aspect presented in Listing 2.1.1 that defines the execution metamodel. The first aspect (lines 1-16) defines two operations for the Transition class: *isEnabled* is a method that checks whether a transition is enabled, and *fire* is transformation rule introduced in Listing 2.1.1 that fires a transition. The second aspect (lines 17-29) defines one transformation rule named *Run* for the Net class that calls *fire* for all enabled transitions.

```
@Aspect(className=Transition)
class TransitionAspect{
def boolean isEnabled(){
   return _self.input.forall[place|place.tokens > 0]
}
@Step
def void fire() {
if (_self.isEnabled) {
// Removes a token from each input place
  for (Place input : _self.input)
     input.tokens = input.tokens - 1
// Adds a token to each output place
  for (Place output : _self.output)
    output.tokens = output.tokens + 1}
  }
}
@Aspect(className=Net)
class NetAspect {
 @Step
 def void run() {
  while (true) {
    val enabledTransition = _self.transitions.findFirst[t|t.isEnabled]
    if (enabledTransition != null)
        enabledTransition.fire
    else
        return
     }
   }
 }
```

Listing 2.2.1: Execution transformation for the Petri net xDSML, written in Kermeta

2.3 Model Execution Tracing

There are many definitions of the term *tracing* in the literature, such as the use of logging mechanisms to record information about a program's execution [28] or a protocol to capture the behavior of a running program [29]. For this work, we define model execution traces as follows.

Definition 4. *A model execution trace captures related information about the execution of an executable model. This information may include execution states reached by the model, events that occurred during the execution, execution state changes, processed inputs, and produced outputs.*

Tracing an executable model is required for performing various kinds of dynamic V&V activities on the model level. Performing such dynamic V&V activities in early stages of MDE development processes is desirable to improve quality, and prevent rework at later stages.

Typical examples of dynamic V&V include debugging, testing, model checking, manual trace analysis (i.e., trace visualization and exploration) and automated trace analysis (e.g., dynamic analysis). Some of these techniques rely heavily on execution traces as a representation of the analyzed model's behavior. For instance, dynamic analysis or run-time monitoring is commonly defined as "the analysis of the properties of a running program" [30]. It comprises the analysis of the execution trace obtained during execution of a system, and provides a representation of system's actual behavior.

There exist many definitions of the concept of an *execution trace* in the literature. The content of traces mainly depends on the degree of abstraction required by the desired dynamic V&V technique as well as the runtime concepts provided by the languages themselves. Alawneh and Hamou-Lhadj [31] have categorized traces of code-centric systems into statement-level traces, routine call traces, inter-process traces, and system call level traces. In the case of executable models, execution traces may contain different type of information depending on the underlying executable modeling language. In addition, instead of tracing threads and function call stacks, which are common programming language constructs, in model execution, concepts like transitions, states, and actions are often traced.

The information to be traced may be extracted from a model execution in different ways. For instance, for executable modeling languages with operational semantics, the interpreter of the executable modeling language may provide facilities for recording execution traces. Also for executable modeling language with translational semantics, additional elements may be inserted in the target model/code that are responsible for producing traces.

An execution trace must conform to a trace format, which defines the concepts required for representing execution traces. A trace format may be defined using different techniques, such as XML schema, metamodels, and grammars. Furthermore, it may be specific to the considered executable modeling language (e.g., specific to UML state machines) or generic and applicable to any executable modeling language. Recorded execution traces may be stored on a disk using different encodings, e.g., they may be recorded in databases, as simple text files or as XMI documents.

Thus, there are many different dimensions (trace purpose, trace content, trace data extraction technique, trace format, etc.) that can be used to classify and compare existing model execution tracing solutions. Our classification schema, which is developed as part of a conducted systematic mapping study, is presented in Chapter 4.

2.4 A Look at Execution Trace Structures

To gain a better understanding of exactly what structures are mostly used to represent traces, we have summarized several kind of data structures commonly used to construct and manipulate execution traces. An initial list of the trace structures and the relevant approaches to design such structures has been presented in [32]. We have completed this list by adding some other structures and related approaches. Then, we have specified which one produces a scalable trace, and what the respective technique is used. Table 2.1 shows a selection list of the trace structures and the approaches of designing such structures. The table contains different kinds of trace data structures. Some approaches propose a trace structure for a specific xDSML. Some other contain generic trace formats that can be used for any xDSML. Self-defining is another trace format capable of defining custom types for elements of the trace. Finally, in some cases, instead of a specific structure, a execution trace metamodel is defined. We review each data structure in the following paragraphs.

2.4.1 Structures with Specific Concerns

A large number of existing trace data structures are provided for General Purpose Languages (GPLs), because traces basically are used for debugging and analyzing programs conforming to GPLs, such as Java, C or C++. Such data structures focus on the concepts of those languages. An example is the Open Trace Format 2 (OTF2) [33] that contains the concepts such as “thread”, “lock”, and “fork”, which provides a trace structure for parallel software. OTF2 is a scalable trace format for representing of traces at run-time by removing unnecessary data, and storing the repetitive data (i.e., time stamp) only once. Another example is Whole Execution Traces (WET) [34],

which is an advance format to represent a compact and complete execution trace containing control flow, variable values, variable memory addresses, and control and data dependencies. By applying an effective two-tier compression strategy including timestamp compression and stream compression, WET reduces the memory size for storing traces.

Another kind of data structures are used for specific platform. KPTrace [35] and CUBE4 [36] are two structures applicable for operating systems (with concepts such as "system call", "memory allocation" or "interrupt") and distributed systems (with concepts such as "topology", "call path" or "system resources") respectively.

Another proportions of trace data structures are specific to an xDSML. Such structures define execution traces for models conforming to the xDSMLs. By scoping a trace structure to a specific xDSML, it only focuses on the concepts of the xDSML. Thereby, the trace structure provides more expressiveness to capture information for conforming models. We will describe the approaches proposing such trace structures in Section 2.4.4. For instance, Timesquare [37] is a trace metamodel, which defines execution traces for CCSL models. The metamodel represents both logical clocks values and chronometric timestamps from real-world sources.

2.4.2 Generic Data Structures

As mentioned before, generic structures are independent from an xDSML, representing general concepts, which are applicable to any xDSML. While such structures are very limited and uncommon, they are more appropriate structures which allow interoperability between existing trace analysis tools, and simplify analyzing traces. Examples of such trace structures include KMF versioning [5] and semantic model differencing proposed by Langer et al. [6].

2.4.3 Self-defining Trace Formats

One specific kind of trace data structures are so-called self-defining trace formats, or meta-formats. In these structures, a trace includes metadata, which defines the format of the trace itself. This is similar to the definition of new types (Java class or a C struct) in programming languages.

The execution trace of such trace format can be adapted to any usage or context. Common Trace Format (CTF) [38] is a known self-defining trace format that can be used for embedded systems or operating systems tracing. CTF provides a compact execution trace, which is suitable for tracing systems with limited resources. Self-defining formats allow us to define a wide range of potential formats, each requires a specific tool for the analysis of the trace content.

2.4.4 Domain-Specific Trace Metamodel Definition Approaches

Recently, there exist some approaches that propose frameworks to define domain-specific execution trace metamodels. An example of a domain-specific trace metamodel is the trace metamodel for fUML proposed by Mayerhofer et al. [39]. Another example is the TopCased project [40] that provides facilities to define a trace metamodel applicable to discrete events system modeling. Such trace metamodel defines a Trace object as a sequence of events. Few approaches proposed the generation of domain-specific execution trace metamodels automatically. An example is the PromoBox framework [41], which provides the ability to define an execution trace metamodel. The authors extend a clone-based generic execution trace metamodel into a domain-specific metamodel. Bousse et al. [10] proposes a similar approach, which automatically derives a domain-specific trace metamodel for a given input xDSML, with the aim of reducing the semantic gap between the trace and the domain, and improving usability as well.

Table 2.1: A selection of execution trace data structures

Name	Type	Content	Compaction
	Compaction Technique		
Open Trace Format 2 [33]	ASCII format	Parallel software	*
	storing the time stamp only once for sequences of events removing all higher value zero bytes from integer attributes		
Traviando [42]	ASCII format	simulation	*
	identifying and removing cycles from a simulation trace		
UML Testing Profile [43]	Metamodel	Software (UML)	
fUML [39]	Metamodel	fUML	
Timesquare [37]	Metamodel	Time, Timesquare	
MPI Trace Format [31, 44]	Metamodel	HPC	*
	transforming a call tree into an ordered DAG where similar subtrees are represented only once.		
Compact Trace Format [4]	Metamodel	Software	*
	transforming a call tree into an ordered DAG where similar subtrees are represented only once		
KPTrace [35]	ASCII format	Operating systems	*
	time compression within the T-charts view		
CUBE4 [36]	Binary format	Distributed software	*
	creating call-path object and reducing repetitions		
Scenario-Based Traces [45]	ASCII format	Sequence charts	
Compact Sequence Diagram [46]	Graph	Software	*
	abstracting repetition patterns and recursive calls included in the trace		
cCCGs [47]	Complete Call Graph	Software	*
	replacing repeated equal sub-trees with a reference to a single instance		
WET [34]	Graph	Software	*
	removing redundancy in the profile information compressing streams of values corresponding to all		
KMF Versioning [5]	other	Generic	*
	sharing immutable objects between the original model and its clones avoiding creation of temporary objects by reusing objects in memory		
SOC-Trace project [48]	Metamodel	Self-defining	*
	applying Best Cut Partition algorithm by using time slicing		
Common Trace Format [38]	Metamodel	Self-defining	*
	using compression scheme for the event packet content		
Pablo SDDF [49]	ASCII/ Binary	Self-defining	
Paje [50]	ASCII/ Binary	Self-defining	
TopCased [51, 40]	Approach	Domain-specific	
Hegedus et al. [13, 52]	Approach	Domain-specific	
Promobox [41]	Generative Approach	Domain-specific	
Filmstrip [53, 54]	Generative Approach	Domain-specific	
Promobox [41]	Generative Approach	Domain-specific	
Filmstrip [53, 54]	Generative Approach	Domain-specific	

2.5 Data Serialization Formats

Data serialization is the process of converting structured data to a format for data sharing or storage. In this section, we present an overview of the common serialization formats that are used as a data carrier in the literature.

XML Metadata Interchange (XMI) [55] is an Object Management Group (OMG) standard that allows to interchange streams or files of data in an XML format. Although XML is the most widely data interchange format, it is not efficient in terms of data size and processing speed. However, XML files can be compressed using Gzip [2].

Flat text format [56] stores data (e.g., traces) in a simple flat file. In particular, the textual logs produced by a program or a formal grammar fall into this category. Flat text format provides a human-readable representation of data that is easy to understand. Therefore, no extra tools are needed to read, debug and administer the serialized data. However, such format has certain limitations and can make data files very big. furthermore, this is not a good solution for serialization of the objects that are part of an inheritance hierarchy or contain pointers to other objects.

Efficient XML Interchange (EXI) [57] is an efficient compact XML representation, which reduces the size of XML and improves processing speed. It is a specification for encoding XML messages into a binary representation. EXI can compress between 1.4 and 100 times the document's original size and over ten times the document compressed with Gzip.

JavaScript Object Notation (JSON) [58] is a lightweight data-interchange format that stores information in an organized, easy-to-access manner. The JSON is a popular alternative to XML because it is more human-readable than XML.

Google's Protocol Buffers (ProtoBuf) [59] is a flexible, efficient, extensible mechanism for serializing structured data. While XML and JSON are text-based data formats, ProtoBuf uses a binary encoding that makes serialized data more compact. Similar to EXI, the ProtoBuf messages are not human-readable after encoding.

[2] http://www.gzip.org/

Chapter 3

Abstraction and Compaction Techniques

This chapter introduce the techniques that have been mostly used for the abstraction and compaction of traces. For each technique, we present several approaches proposed by researchers. In Sections 3.1, 3.2 and 3.3, we introduce the techniques that are used to deal with the large size of data in three different contexts containing code-centric development, model driven development, and database domain, respectively.

3.1 Trace Abstraction in Code-Centric Approaches

3.1.1 Trace Visualization

Trace visualization is a technique concerned with the extraction of high level views of the run-time information to support system comprehension. Most approaches (e.g., [60], [4], [61]) use a UML sequence diagram to visualize interactions among grouped objects, and depict the behavior of the program.

Sharp et al. [62] proposed several methods to explore a large-scale sequence diagram. They applied some methods (e.g., filtering methods based on "the time" and zooming function) for reducing the amount of run-time information.

Prada-Rojas et al. [63] provided a compact view from the information within execution traces. The view is a small representation of the traces that provides a global view of the information and a summary of the execution traces. The authors proposed a tool named OutlineView to present less data but more useful information, by summarizing key features of large embedded traces. Using this tool, user is able to select the trace events regarded to the global view, specify the functions to

apply to the selected events, and display and analyze the resulting view. In fact, this approach is a visualization , and no compaction is applied to execution traces.

3.1.2 Trace Exploration

Trace exploration is concerned with techniques that allow browsing the content of traces, and searching the trace content for specific components easily[3]. Such techniques can reduce the amount of information displayed.

For example, SEAT (Software Exploration and Analysis Tool) [64] is a trace exploration tool that provides various capabilities for the exploration of traces. Using SEAT, a software maintainer can explore the trace by searching for specific components, filter the trace content using several techniques such as pattern matching, sampling, and so on.

3.1.3 Abstracting the History of Object Interactions

This technique reduces the size of traces by abstracting the dynamic interactions among objects of the system [65]. It can help software engineers to understand the system.

As an example, Hamou-Lhadj et al. [4] proposed a method for obtaining the summary of an execution trace by removing utility objects which do not implement key system concepts. They proposed a metmodel called the CTF (Compact Trace Format) which represents traces of routine calls as directed acyclic graphs. This way, common subtrees are represented only once. They showed that this native compaction can result in almost 90% compaction ratio.

Taniguchi et al. [46] proposed a method to extract compact sequence diagrams from dynamic information of object-oriented programs. This method gets an execution trace of method calls of the target program. They proposed four compaction rules to reduce the size of execution traces. This compaction is performed by abstracting some repetition patterns and recursive calls appearing in the trace. For compaction of repetitions, a repetition of similar sub-trees in a call tree are detected, and replaced with one representative, which shows whole repeated structure and the number of repetitions. The compacted execution trace is translated into a compact sequence diagram.

Noda et al. [66, 67] proposed a technique that generates abstracted sequence diagrams from the information of applied GoF design patterns in a source code. To abstract an execution trace, the authors defined some grouping rules for each design patterns. By applying static analysis, non-useful objects and complex interactions are removed from the application, then the objects belonged to a design pattern are grouped. The authors used a UML sequence diagram to visualize interactions among grouped objects, and depict the behavior of the program. Similarly, another approach [66] abstracts execution traces by identifying and grouping correlated objects. Therefore, the size of execution traces are reduced due to the reduction in the number of objects in the execution trace. In this approach, the object interactions are visualized with a sequence diagram.

3.1.4 Graph Reduction

Graph reduction techniques aim to reduce the generated graph of the trace to a smaller graph by detecting, removing and replacing the similar graph nodes. Quante and Koschke [68] introduced a technique to build an object process graph through dynamic analysis using run-time information. In this approach, the behavior of different components of a program is extracted as dynamic object process graphs. The size of graph can be reduced by removing branch nodes, unnecessary label nodes, local loops ,and irrelevant subgraphs. These are repeatedly applied until the graph cannot be simplified any more. The authors defined an object graph construction process to build an object process graph through dynamic analysis using run-time information. The process includes filtering mechanism to extract the related information for object trace. Using filtering method, some parts of code are instrumented that deal with a certain type of expression. Then, a graph is generated from the resulting object trace. The size of graph can be reduced by removing branch nodes, unnecessary label nodes, local loops and irrelevant subgraphs. This is repeated until the graph cannot be simplified any more. The process makes easier the analyzing and understanding the program.

Hamou-Lhadj and Lethbridge [4] proposed an approach that represents trace as a tree structure, and transforms into a more compact ordered directed acyclic graph (DAG) by representing similar

subtrees only once. For this work, the authors extended Valiente's algorithm that traverses the tree, and identifies trace patterns. By defining a set of matching criteria, the algorithm considers detecting similar subtrees that are not necessarily identical. The technique is lossless, meaning that the original trace can fully be reconstructed from the compact one.

Likewise, Alawneh et al. [31] applied a similar approach to represent run-time information generated from HPC applications. They represented traces as an ordered directed acyclic graph (DAG) by capturing common subtrees only once. The structure promotes to be adopted as a standard exchange format that is scalable to very large traces.

3.1.5 Partitioning and Clustering

Clustering is a data mining technique that is considered as a trace compaction technique to reduce the size of traces. It provides the capability for identifying the specific parts of the trace such as most related parts of the trace, frequent patterns or specific components. Dugerdil and Repond [69] used a software clustering technique based on the dynamic analysis of method calls while executing a scenario of a system. They implemented a clustering technique for identifying the set of functional components by splitting the execution trace in contiguous segments, and observe collaborating classes presented in each segment.

Zaidman and Demeyer [70] proposed a heuristic approach that reduces the large size of a trace by finding frequent patterns within the trace. They analyzed consecutive samples of the trace to identify recurring patterns of events having the same global frequencies. In other word, they finds the events with similar frequency, and splits the trace into frequent event clusters. First, irrelevant events (e.g., low-level method calls) from the trace is removed. Then, the program is executed based on a specific scenario, and a file containing a sequence of all method calls is provided. After counting number of events included in the trace, the euclidean distance is applied for the sequence of events. Finally, the trace is analyzed for identifying regions including recurrent patterns that are more interesting and more frequent.

Bose and Aals [71] proposed a technique for clustering event logs. This technique considered

sub-sequences of activities, which exist across multiple traces. This sub sequences are with different lengths. The idea is finding similar regions (sequence of activities) within a trace, and sharing high similarity regions between two or more processes. The similar regions across a set of traces in an event log can be shared between two or more process instances. Such technique provides ability to cluster traces so that the regions common between different parts of the trace are put in the same cluster. For large data sets containing many repeats, a filtering mechanism for the repeats is used.

Song at el. [72] presented an approach using trace clustering for event logs. The authors proposed a format for information in an event log, and implemented divide-and-conquer approach in a systematic manner. In this approach, traces are characterized by profiles, which are a set of related items representing the trace from a specific perspective. The profile allows applying any clustering algorithm that can be employed for the actual partitioning of the log. The event log is divided into several subgroups, and constructs process models, each containing a list of elements.

3.1.6 Program slicing

Program slicing is a technique that splits the program in several slices, each one is the part of the program that affects the value of a chosen variable during a program execution [73].

Smith and Korel [74] proposed a technique of slicing event traces to reduce the number of events for analysis. This technique uses a slicing algorithm to identify several types of dependencies between events. All events that are irrelevant or do not affect the starting event are removed from the event trace that can further reduce the size of the sliced event trace. Dhamdhere et al. [75] followed similar approach, and provided a compact execution history for dynamic slicing of programs by focusing on critical statements in a program. An instrumentation algorithm is used to identify critical nodes and similar loops for the summarization. In this technique, only critical statements appear more than once in an execution trace, and all other statements appear at most once.

Zaidman et al. [70] proposed a technique that applies web mining techniques to execution

traces. This technique is based on the idea that a large trace contains many unimportant sections, such as long loops in the execution. The trace can be divided to a number of slices using aspect oriented programming (AOP). The approach includes several steps. First, a dynamic call graph is built from the information contained in the trace. Then, web mining techniques (e.g., HITS algorithm) are applied on the trace to identify irrelevant parts of the trace. Finally, a compacted call graph is derived from the dynamic call graph by considering the interesting and relevant sections in program execution. The reduced size of the trace improve dynamic analysis and program comprehension process.

3.1.7 Pattern Detection

Pattern detection is the ability to group similar sequences of events in the form of execution patterns. Pattern matching is an efficient technique to reduce the size of traces by detecting execution patterns, and representing the same patterns only once [60]. For this work, such technique often uses a set of matching criteria to generalize the sequences of events so that they can be considered as instances of the same pattern.

Pauw et al. [60, 76] proposed a tool called Ovation that visualizes traces using a tree view based. This tool allows users to browse the trace at various levels of detail. It also provides the ability to identify execution patterns, and to eliminate contiguous repetitions of sequences of calls. To overcome the large size of the trace, similar sequences of events are considered as instances of the same pattern. The authors used a generalization mechanism to identify repetitive execution patterns. The patterns are included in loops and recursive functions. In this approach, a set of matching criteria including namely, identity, repetition, and depth-limiting is defined for the pattern detection. These matching criteria require to be set. For example, the depth-limiting criterion involves setting the depth at which two sequences of events need to be compared. In some case, the different combinations of matching criteria will result in different filtering of the content of traces.

Sartipi and Safyallah [77] proposed a pattern discovery technique to extract frequent patterns

in the execution traces of a software system. The proposed approach uses dynamic analysis, data mining technique sequential pattern discovery, and concept lattice analysis. The authors applied three different techniques to deal with large execution traces. First, the the loop-based repetitions are removed from the trace by using a top-down program analysis. Then, a data mining sequential pattern discovery [78] algorithm is applied to extract frequent parts of the trace. Finally, the last technique is based on string manipulation algorithms [79] that is used for identifying repetitive patterns in a string of elements. All these techniques are performed in four different stages: trace extraction, pattern mining, pattern analysis, and structural evaluation.

3.1.8 Hiding Components

This technique is used for removing some information (e.g., all invocations of a specific method) from the trace that can reduce the trace size. For instance, using Program Explorer [80], an analyst is able to remove methods, specific objects or even classes. This can be done by pruning and slicing of data. Similarly, Hamou-Lhadj et al. [4] reduced the trace size by filtering the content of traces with removing implementation details including utilities. The authors used fan-in analysis method to detect utility components. They identified categories of implementation details such as the components that implement data structures, mathematical functions, and components that implement input/output operations, and so on. Note that in this technique, the components that have no effect on the comprehension of the trace should be considered to be removed.

3.2 Trace Abstraction in Model-Driven Approaches

3.2.1 Sharing Immutable Objects

Fouquet et al. [81, 82] developed the Kevoree Modeling Framework (KMF), which is an alternative to EMF [83], specifically designed to support models at run-time in terms of memory usage and run-time performance. KMF provides the same features than EMF for code generation

facilities and models (un)marshalling. To deal with the large size of the model during run-time, the authors used a technique that duplicates only the mutable parts of a model, while applying a flyweight pattern [84] to share the immutable objects between the original model and its clones. KMF also used an additional technique for reducing the memory footprint by avoiding the creation of temporary objects. To this work, the objects are reused for loading and saving in memory instead of creating temporary objects. Such mechanisms improve the memory usage.

Likewise, Bousse et al. [27] proposed a technique relying on data sharing among run-time representations of model clones. As a metamodel contains both mutable and immutable run-time data, their approach is based on the idea that the immutable run-time data can be shared between run-time representations of a model and its clones. More precisely, such technique aims to avoid duplicating immutable run-time objects which are the objects that cannot change during execution. These objects can be shared between the original model and its clones, then results reducing memory used when generating the clone.

3.2.2 Avoiding Redundancy in Traces

Bousse et al. [27] presented a generative approach to automatically derive multidimensional domain-specific trace metamodels that provide facilities for efficiently processing traces. Such metamodels define Trace object as a sequence of execution steps and execution states. In this approach, execution states contain the values of the mutable properties of a model. Thereby, the trace metamodels reduce redundancy within traces when manipulating traces by creating a single object per value change of a mutable field instead of storing the same value twice; hence improves scalability in space. However, even the approach reduces the memory footprint of traces, the trace still contains redundant patterns within execution steps, and repetition in states as well. Yet, this technique still require substantial memory usage due to the redundant patterns and repetitions.

3.2.3 Recording Modifications of the Dynamic Model

Hegedus et al. [85, 52] proposed a generic execution trace metamodel that is manually extended into a domain-specific trace metamodel using inheritance relationships. The authors used three different ways for representing dynamic information during a model execution. The first method is *Snapshot*, a simple representation of the trace, which stores all dynamic information for specifying the new state of the model element. Although this way is easy to implement, the trace is complex and large. The second method is *trigger*, used by event-driven languages, when an event is triggered. Instead of storing all relevant event dynamic information, only the event is recorded. The third one is *change*, in which instead of storing all dynamic information of the new state of the model, only the modification (delta) between two subsequent states of a dynamic model element is represented. For this work, the approach stores both values of dynamic model elements before and after the modification. Compared to the other methods, such method is more effective to reduce the size of model execution traces.

3.3 Data Compression Techniques in Database Domain

Data compression is widely used in data management to save storage space and network bandwidth. It is used to reduce the size of the data, improve performance, and save storage space and network bandwidth as well [86]. Indeed, database performance strongly depends on the amount of available memory. Therefore, it is required to reduce the size of data effectively by keeping and manipulating data in memory in a compressed form. In the following, we introduce techniques to allow data compression on data management.

3.3.1 Column-Oriented Database Systems

In-Memory Column Store is a technique for storing large amount of data in database. However, it has major effects on memory consumption and high speed in query. There are several architectures for column store databases. The most important ones are presented in the following.

C-Store [87] is a column-oriented database management system that stores data by column and not by row. The column is compressed using a column-specific compression method, and sorted in the corresponding table. The compression method for each column depends on the data type, the number of distinct values, and storing of data. Each column in C-Store may be stored several times in several different sort orders. There are numerous possible compression schemes that can be applied, i.e., run-length encoding, bit-vector encoding, dictionary compression and patching.

Run-Length Encoding (RLE) [88] is a simple form of lossless data compression that compresses runs of data in a column to a compact representation. Thus, it is appropriate for the columns with the reasonable-sized runs of the same value. These runs are replaced with value, start position, and run-Length, where each having a fixed number of bits.

Dictionary [89] is a class of lossless data compression algorithms, which creates a dictionary table for an entire table column sorted on frequency, and represents values as the integer position in this table.

3.3.2 Rainstor

RainStor [90] is a column store technique for storing data, in which every unique value in the dataset is stored once (and only once). In this technique, every data row is represented as a binary tree that the original record can be reconstructed by using a breadth-first traversal of the tree. Unlike the column-store approaches that create dictionaries and search for patterns only within individual columns, RainStor's compression algorithm finds patterns across different columns.

Consider a simple example to understand how this technique works. Table 3.1 contains 15 records with four attributes, and represents a subset of *Order* data from a particular retail enterprise that sells bicycles and related parts. As presented in the table, there is dependency between some of the columns. In particular, the value of *Shipdate* is usually 1 or 2 days after the *Orderdate*, and the value of *Price* for different products are usually consistent across *orders*, but there may be slight variations in the *price* value. Figure 3.1 represents a forest of binary trees, the compressed view of data of the Table 3.1 using RainStor technique. For the records of *Orders*, we have 15

Table 3.1: Excerpt of order data of a retail system (taken from [90])

Id	Orderdate	Shipdate	Productname	Price
1	03/22/2015	03/23/2015	“bicycle”	300
2	03/22/2015	03/24/2015	“lock”	18
3	03/22/2015	03/24/2015	“tire”	70
4	03/22/2015	03/23/2015	“lock”	18
5	03/22/2015	03/24/2015	“bicycle”	250
6	03/22/2015	03/23/2015	“bicycle”	280
7	03/22/2015	03/23/2015	“tire”	70
8	03/22/2015	03/23/2015	“lock”	18
9	03/22/2015	03/24/2015	“bicycle”	280
10	03/23/2015	03/24/2015	“lock”	18
11	03/23/2015	03/25/2015	“bicycle”	300
12	03/23/2015	03/24/2015	“bicycle”	280
13	03/23/2015	03/24/2015	“tire”	70
14	03/23/2015	03/25/2015	“bicycle”	250
15	03/23/2015	03/25/2015	“bicycle”	280

relevant binary trees, each are shown using the green circles at the top of the figure. For instance, the binary tree corresponding to the first record is depicted on the left side of the figure. The root of the tree has two children: the intermediate nodes “A” and “E”. Node “A” points to 03/22/2015 (related to the *Orderdate* of record 1), and to 03/23/2015 (related to the *Shipdate* of record 1). Node “E” points to 300 as the *price* of the record 1 and “bicycle” as the *Productname* of the record 1. Three other root nodes point to the node “A” that are corresponding to the records 4, 6, and 7 due to the same values for the *Orderdate* (03/22/2015) and the *Shipdate* (03/23/2015). Likewise, for the node “E”, there is an additional arrow from the root tree corresponding to record 11 as it contains the same values with record 1 for the *Productname* (bicycle) and the *price* ($300).

RainStor structure yields a data reduction rate of 40:1, i.e., it requires 40 times less storage. The shared internal nodes are the main difference between RainStor’s compression algorithm and a column-store technique. RainStor provides more capability to search for the similar patterns existed in the dataset, and compresses data by pointing the roots of the tree to the shared nodes related to the patterns.

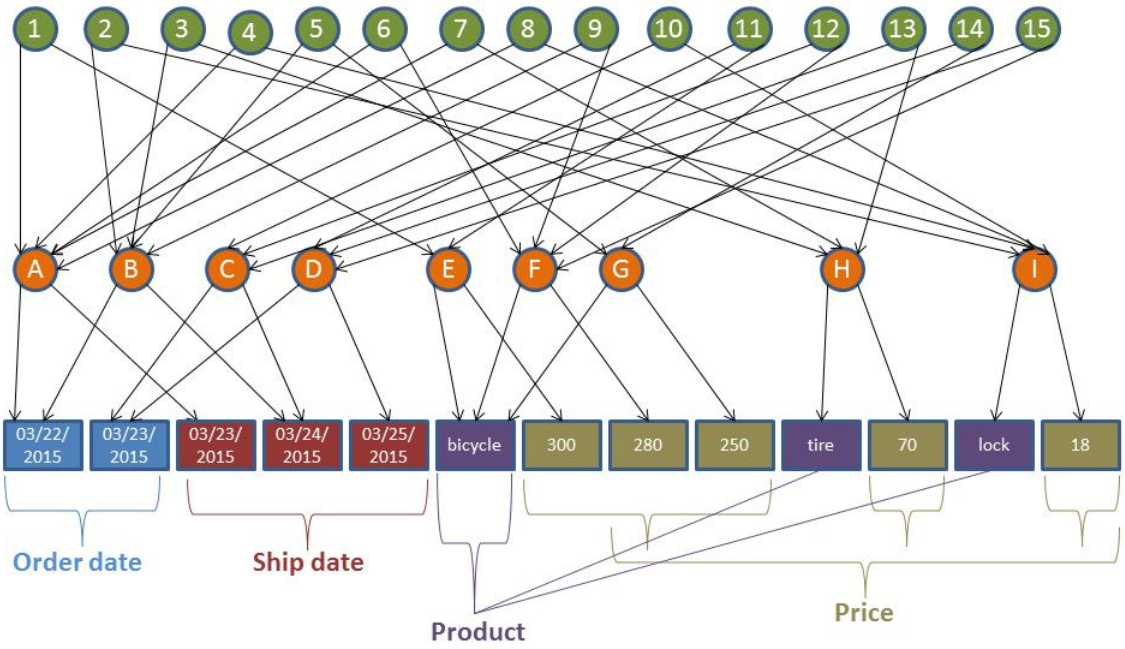

Figure 3.1: Forest of binary trees compression for the order records depicted in Table 3.1 (taken from [90])

Part II

Contributions

Chapter 4

A Taxonomy for Model Execution Tracing Approaches

In this chapter, we present our first contribution, which is a taxonomy for model execution tracing approaches, obtained from a systematic mapping study. In Section 4.1, we introduce the context of our contribution and our proposal. In Section 4.2, we describe the research method used for conducting the mapping study and the classification scheme applied for the research. Continuing, Section 4.3 reports the main findings. In Section 4.4, we present identified open challenges and present future research directions. In Section 4.5, we evaluate our approach and findings by discussing limitations and threats to validity. Section 4.6 outlines the main related work. Section 4.7 summarizes the results and concludes.

4.1 Introduction

In the last decade, there has been a noticeable increase in the number of papers on tracing techniques for executable models, but without a clear direction where the field is heading. Furthermore, it is not clear what the advantages and limitations of the proposed techniques are. We believe this is due to the lack of a survey on the state of the art and of a classification of existing work.

To fill the aforementioned gap, we conducted a systematic mapping study on the existing proposals for model execution tracing solutions following the guidelines presented by Kitchenham and Charters [91] and Petersen et al. [92]. We examined 64 research studies from an initial set of 645 and classified them based on the following facets: (1) the types of models that are traced,

(2) the supported execution semantics definition techniques, (3) the traced data, (4) the purpose of model execution tracing, (5) the used data extraction technique, (6) the used trace representation format, (7) the used trace representation method, (8) the language specificity of the trace structure, (9) the data carrier format used for storing traces, and (10) the maturity level of model execution tracing tools. Using this classification, we evaluated the state of the art of solutions for model execution tracing, and pointed to promising future research directions in this area.

The main contributions of this mapping study are (i) a framework for classifying and comparing solutions for model execution tracing, (ii) a systematic review of the current state of the art in model execution tracing, and (iii) an exploration of open research challenges in model execution tracing. The study targets researchers who want to contribute further to the area of model execution tracing, as well as practitioners who want to gain insight on existing model tracing solutions.

It is worth noting at this point that the conducted study focuses on approaches for tracing executable models. Thus, other tracing techniques employed in MDE, such as maintaining traceability links between source and target models of model transformations, and tracing the execution of model transformations, are out of the scope of this survey. Also, we exclude tracing techniques for programs written in general-purpose programming languages.

4.2 Research Method

To conduct the systematic mapping study, we followed the guidelines presented by Kitchenham and Charters [91] and Petersen et al. [92] in order to identify, evaluate, and present existing research studies around our topic. The goal of this survey is to answer the following research questions.

Q1 (Type of Model): Which executable modeling languages are targeted by model execution tracing approaches?

Q2 (Semantics Definition Technique): Which techniques are used to define the execution semantics of executable modeling languages targeted by model execution tracing approaches?

Q3 (Trace Data): What kind of data is captured in model execution traces?

Q4 (Purpose): For what purposes is model execution tracing used?

Q5 (Data Extraction Technique): Which techniques are used for extracting run-time information from model executions in order to construct execution traces?

Q6 (Trace Representation Format): Which data representation format is used for defining the trace data structure?

Q7 (Trace Representation Method): How is the trace data structure defined?

Q8 (Language Specificity of Trace Structure): Is the data structure specific to the considered executable modeling language, specific a particular kind of executable modeling language, or considered independent of any executable modeling language (i.e., generic)?

Q9 (Data carrier format): Which data carrier format is used for storing traces?

Q10 (Maturity Level): How mature are available tools for model execution tracing?

The study protocol includes three phases, namely planning, conducting, and reporting. In the following, we discuss the planning and conducting of the study. The study results are reported in Section 4.3.

4.2.1 Review Planning

The first phase of our survey process is planning, which consists of defining the search strategy and review process. In the following, we explain the search strategy. The review process is outlined as part of the discussion of the review conduction in Section 4.2.2.

We used the following online libraries to find research studies related to model execution tracing. With this list of online libraries, we aimed at achieving a comprehensive coverage of publication venues from all major publishers in the field of software engineering.

- ACM Digital Library (http://dl.acm.org)
- IEEE Xplore (http://ieeexplore.ieee.org)
- ScienceDirect (http://www.sciencedirect.com)
- Springer Link (http://www.springer.com)
- Scopus (http://www.scopus.com)

The terms we used to select relevant research studies are as follows. Each term includes several keywords meaning that at least one of the keywords has to be present in a paper.

A = *model tracing, model-based trace, execution trace, tracing, trace*

B = *MDE, model-driven, model-level, model-based*

C = *meta-model, metamodel, modeling language*

D = *model execution, model verification, dynamic analysis, executable model, xDSML*

The overall search string can be combined in the following way:

$$\textbf{Search String} = (A \wedge (B \vee C \vee D)) \tag{4.1}$$

The rationale behind using this search string is to identify the largest number of research studies related to model execution tracing. We performed an advanced search in the aforementioned online research databases and search engines using this search string.

Furthermore, we defined two types of inclusion and exclusion criteria to decide whether a publication found in the search should be included in the study or excluded. The first type is a set of format-related criteria, such as publication language and publication type. The second type is a set of content-related criteria. The selection criteria used in this study are given bellow. The

publications have been selected on the basis of these criteria through a manual analysis of their titles, abstracts, and keywords. when in doubt, also the conclusions of a publication have been considered.

Inclusion Criteria:

1. Publications in peer-reviewed journals, conferences, and workshops
2. Publications that address problems in the field of model execution tracing
3. Publications dealing with recording execution traces for models or designing execution trace formats for executable modeling languages

Exclusion Criteria:

1. Books, web sites, technical reports, dissertations, pamphlets, and white-papers (based on the guidelines suggested by Adams et al. [93] and Petersen et al. [92]).
2. Summary, survey, or review publications
3. Non peer-reviewed publications
4. Publications not written in English
5. Publications after February 2018, the time the final search for primary studies was conducted
6. Publications on traceability in model transformation
7. Publications that do not consider executable models
8. Publications not focusing on MDE (e.g., execution tracing in code-centric approaches)

4.2.2 Review Conduction

The second phase of our study process, review conduction, consists of three main activities: article selection, data extraction, and article classification, which are elaborated in the following.

4.2.2.1 Article Selection

The article selection comprised a pilot study, the actual selection of primary studies, and the assessment of the quality of the selected primary studies.

Pilot study Before the actual selection of articles, we performed a pilot study as suggested by Kitchenham and Charters [91] and Petersen et al. [92] to confirm the reliability of our selection criteria.

In this pilot study, a set of ten articles was preselected from different sources and publishers. This list was defined based on the bibliography of one of our earlier papers [94]. This selection included seven articles that should be included in the study as primary studies and three articles that should be excluded.

The selected articles were then given to a domain expert who was not involved in the planning phase of the study process. Therefore, he was not biased by the search process. He was asked to decide based on the defined inclusion and exclusion criteria which of the selected articles should be considered as primary studies and which ones should be excluded from the mapping study. The results were then cross-checked against the initial classification of the preselected articles in primary studies and non primary studies.

Pilot study results: The first execution of the pilot study failed. Out of the ten articles, only six were correctly classified. In particular, five articles were correctly identified as primary studies and one was correctly identified as non primary study. Based on these results, the selection criteria were refined and the pilot study re-executed. After the second execution, the results were acceptable: Seven articles were correctly identified as primary studies and one was correctly identified as non primary study. The two articles that were assessed differently focused on dynamic analysis in code-centric approaches, which is beyond the scope of this mapping study. This led us to add an extra exclusion criterion excluding tracing approaches not focusing on MDE (exclusion criterion 8).

Primary studies selection This step comprised the search for relevant publications using the search string introduced in Section 4.2.2, the elimination of duplicate publications found in multiple online libraries, and the filtering of the publications by applying the aforementioned selection criteria.

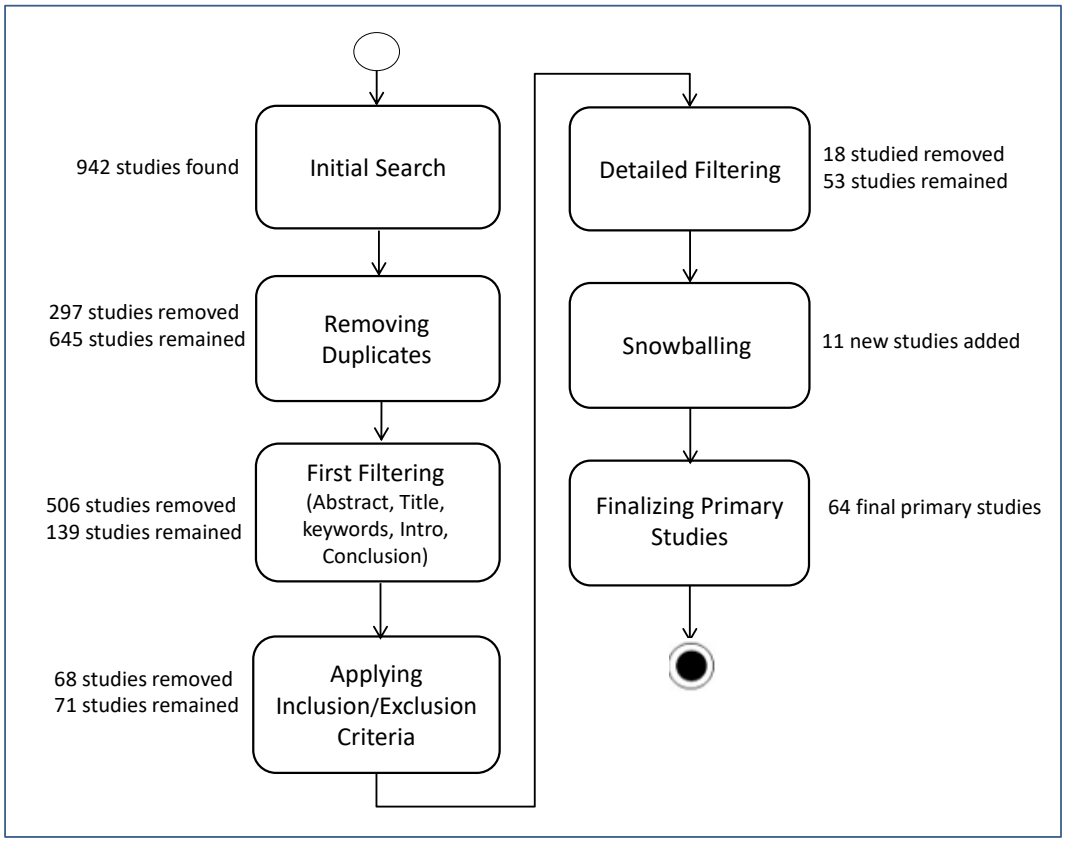

Figure 4.1: Primary studies selection process

Figure 4.1 shows the primary studies selection process along with the obtained results of the tasks. The initial search process returned 942 results including 297 duplicates. For the studies that were identified in more than one online library, we considered their original publisher. In order to assess the relevance of the found studies to our topic, we reviewed their titles, abstracts, keywords, introduction sections, and if needed conclusion sections. In this step, 506 studies were rejected, while 139 moved on to the next step. Next, we applied the inclusion and exclusion

criteria. The result was a set of 71 studies. In the next step, a more detailed filtering was applied by inspecting the entire content of the remaining 71 studies. Most of the studies eliminated in this step were related to traceability in model transformation, and also dynamic analysis in code-centric approaches. The filtering yielded 53 remaining studies. In order to minimize the risk of missing any relevant studies, we performed a snowballing step by checking the references of the remaining 53 studies and identified 11 more studies fulfilling the inclusion criteria. Hence, these studies were added to the primary studies. The final set of primary studies investigated in this mapping study consists of 64 studies.

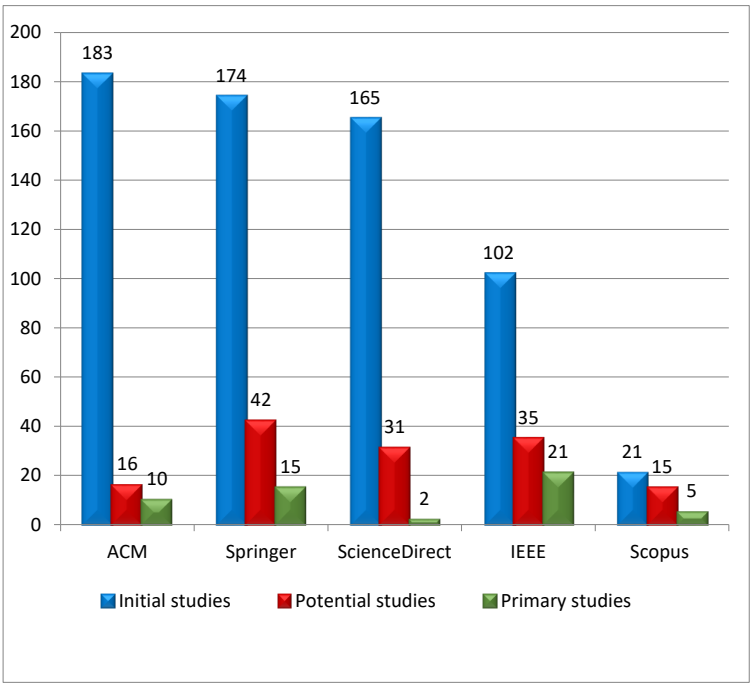

Figure 4.2: Studies retrieved through online libraries

Publication trends: Figure 4.2 shows for each used online library the total number of studies that were retrieved using the defined search string (initial studies), the number of studies remaining after the removal of duplicates and first filtering (potential studies), and the number of studies finally included in the mapping study after applying inclusion and exclusion criteria and detailed

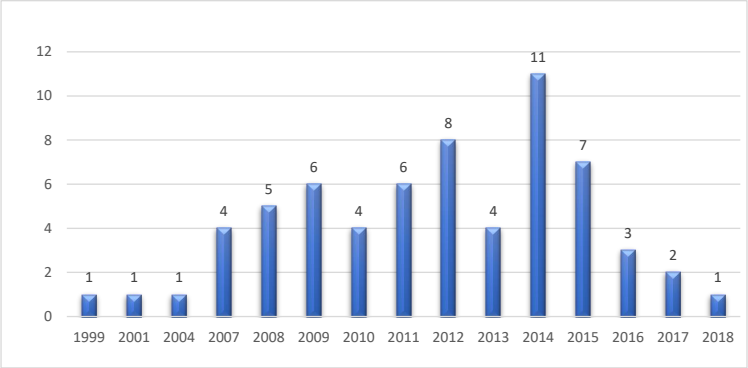

Figure 4.3: Primary studies per year

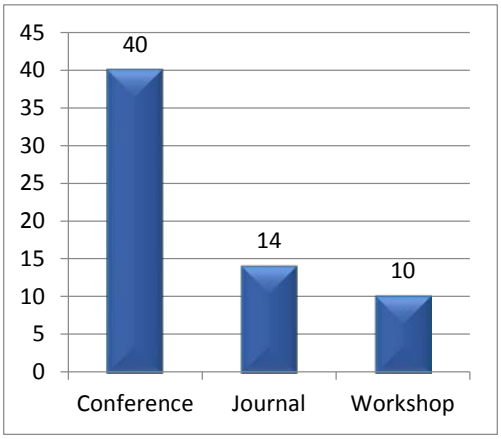

Figure 4.4: Primary studies per publication type

filtering (primary studies). Figure 4.3 shows the publication years of the primary studies. As can be seen in this figure, first publications on the topic of model execution tracing appeared around the year 2000, but only in 2007, the topic gained more interest by the scientific community leading to an increase in publications on this topic per year with the highest number of publications in the year 2014.

Finally, Figure 4.4 presents the distribution of the selected primary studies based on the publication type. We only included studies published in peer-reviewed workshops, conferences, and journals. Please note that we considered symposia and congresses also as conferences. From the 40 primary studies shown in Figure 4.4 as conference papers, 4 have been presented at symposia

and 1 at a congress.

The figure shows that studies in the field of model execution tracing have been mostly published in conference proceedings.

Quality assessment We also evaluated the quality of the selected primary studies as suggested by Kitchenham and Charters [91] and Petersen et al. [92] to make sure that they are of sufficient quality to be included in a systematic mapping study. For this, we developed a checklist containing five quality assessment questions as presented in Table 4.1. The questions are based on the suggestions given in [95, 91] and [92], as well as the questions used in a study by Santiago et al. [96]. The questions have been answered for all primary studies by Fazilat Hojaji and Bahman Zamani. Thereby, the score values were 'Yes' = 1, 'Partly' = 0.5 or 'No' = 0.

Table 4.1: Quality assessment questionnaire

Topic	**Question**
Objective	Did the study clearly define the research objectives?
Related work	Did the study provide a review of previous work?
Research methodology	Was the research methodology clearly established?
Validity and reliability	Did the study include a discussion on the validity and reliability of the procedure used?
Future work	Did the study point out potential further research?

Figure 4.5 shows the percentages of primary studies assigned 'Yes', 'Partly', or 'No' on the five quality assessment questions. It shows that for each of the five questions, most of the studies (%83-%100) score 'Yes' or 'Partly', which confirms that the selected primary studies are of sufficient quality to be included in this mapping study. We also calculated ("% total score") for each quality assessment question, which shows the percentage of scores obtained by all the primary studies assigned for a given quality assessment question over sum of the scores obtained by all primary studies for all QA1 to QA5. The arithmetic mean of the scores is 3.77 and the standard deviation 0.95. Figure. 4.6 shows a pie chart depicting the distribution of scores for the assessment questions. It illustrates that QA1 obtained the highest score (%23) over the total score, while QA5 has the

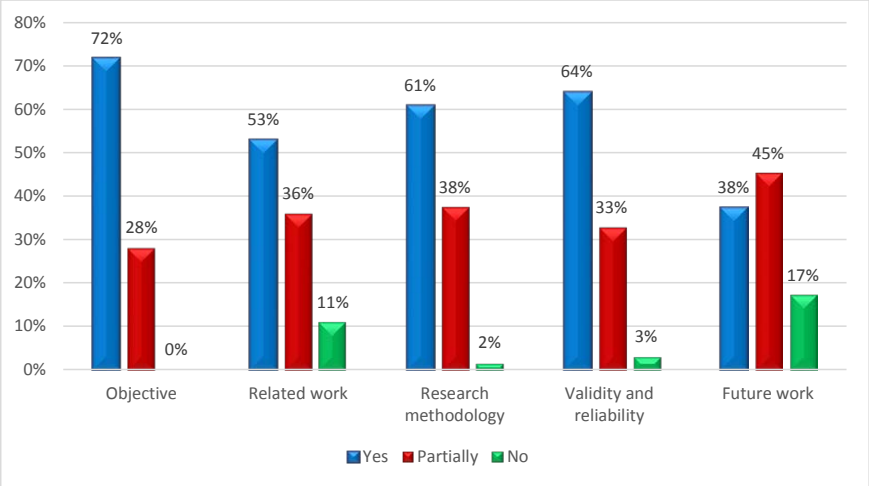

Figure 4.5: Results of quality assessment of selected primary studies

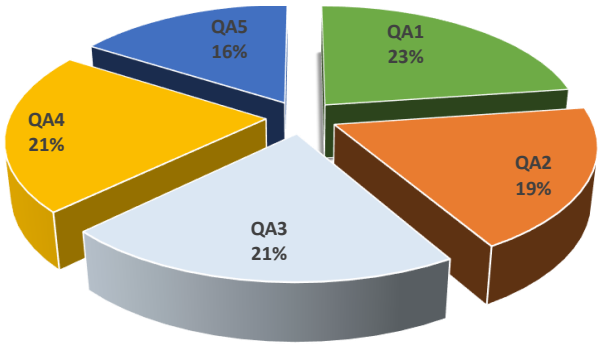

Figure 4.6: Total score for quality assessment questions

least (%16).

4.2.2.2 Data Extraction and Classification Scheme

In this section, we describe how we classified the selected primary studies. We developed a classification scheme by inspecting the content of all 64 selected primary studies with the goal of addressing our research questions. In particular, for each research question, we assigned keywords to the primary studies, which provide answers to the research question. For example, we assigned the keyword “interactive model-level debugging” for research question Q4 if the purpose of the

model execution tracing approach presented in a primary study was to realize this type of dynamic V&V technique. Subsequently to this keywording phase, the keywords assigned for each research question were clustered and for each cluster, a keyword encompassing all clustered keywords was assigned. The result was a set of attributes for each research question that was used to classify the primary studies for this research question. The attribute sets are described in the following.

Types of Models (Q1) This attribute set is used to characterize model execution tracing approaches concerning supported executable modeling languages. For this, we defined the following attributes.

- *Any*: refers to approaches that can be applied to any executable modeling language, i.e., approaches that can be used to trace the execution of models conforming to any executable modeling language.
- *UML models*: refers to approaches that have been specifically designed to trace the execution of models conforming to UML or a subset of UML.
- *Workflow models*: refers to approaches that have been specifically designed to trace the execution of workflow models that define the flow of work in processes. Workflow models can be expressed in executable modeling languages like Petri nets and BPMN.
- *Other*: refers to approaches to which the other attributes do not apply, i.e., approaches that are designed to trace the execution of models conforming to particular executable modeling languages or kinds of executable modeling languages other than UML and workflow modeling languages. Note that approaches in this category are applicable to a restricted set of executable modeling languages, while approaches in the category *Any* can be applied to any executable modeling language.

Semantics Definition Techniques (Q2) This attribute set refers to the way execution semantics are defined for the executable modeling languages supported by a model execution tracing

approach. As introduced in Section 2.2, we distinguish translational and operational semantics. Hence, the attributes are defined as follows.

- *Translational*: refers to approaches applicable to executable modeling languages whose execution semantics are defined in a translational way.
- *Operational*: refers to approaches applicable to executable modeling languages whose execution semantics are defined in an operational way.
- *Other*: refers to approaches applicable to executable modeling languages whose execution semantics are neither defined translational nor operational, but with some other kind of semantics definition technique, such as denotational semantics.
- *Unknown*: refers to approaches where no information about the kind of supported execution semantics is provided in the associated primary studies.

Trace Data (Q3) With this attribute set, we characterize model execution tracing approaches concerning the data recorded in execution traces. In particular, we used the following attributes.

- *Event*: refers to approaches that trace events occurring during the execution of a model.
- *State*: refers to approaches that trace information about the evolution of the execution state of a model.
- *Parameter*: refers to approaches that trace inputs processed by the execution of a model or outputs produced by the execution of a model.

Purpose (Q4) This attribute set is concerned with the purpose of the investigated model execution tracing approaches, in particular, the purpose of execution traces produced by the individual approaches. We determined the purposes from the primary studies by considering the applications of execution traces mentioned or explicitly presented by the authors as part of their contribution.

This way, we identified the following purposes of model execution traces that serve as attributes for this research question.

- *Debugging*: refers to techniques to interactively control and observe the execution of a model in order to find and correct defects. Model execution traces can be utilized in different ways for the purpose of debugging executable models. For instance, execution traces can be used in omniscient debugging to travel back in time in the execution to visit previous execution states, to replay past executions, or to retrieve the run-time information about the execution of a model that should be shown to a user.

- *Testing*: refers to techniques for testing models concerning functional or non-functional properties or for testing applications with the help of models concerning functional and non-functional properties. Execution traces can be used in testing, for instance, as oracles or as basis for evaluating test cases providing the necessary run-time information to determine the success or failure of a test case.

- *Manual analysis*: refers to techniques for manually analyzing the execution behavior of a model or the modeled system. Such techniques are mostly concerned with the visualization and querying of model execution traces.

- *Dynamic analysis*: refers to the analysis of run-time information gathered from the execution of a model, similar to the definition of dynamic analysis of programs given in [97]. Thereby, gathered run-time information can be analyzed for different properties, including general behavioral properties like deadlock-freeness, functional properties, and non-functional properties. Execution traces have a natural application in dynamic analysis as they record run-time information about a model or program execution.

- *Model checking*: refers to techniques in which all the possible execution states of a model are checked with respect to some property. Model checking may rely on execution traces for representing the state space of model or for representing counter examples found for violated properties.

- *Semantic differencing*: refers to techniques that compare execution traces of models to understand the semantic differences between them.

Data Extraction Techniques (Q5) This attribute set focuses on the techniques used for the extraction of the traced run-time information during model execution. We categorized the data extraction techniques identified in the investigated primary studies using the following attributes.

- *Source instrumentation*: Elements are added to the executable model, which are responsible for the construction of execution trace.

- *Target instrumentation*: This data extraction technique only concerns model execution tracing approaches considering executable modeling languages with translational semantics. In this technique, elements are added to the target model or target code generated from a model for its execution. These introduced elements are responsible for the construction of execution traces.

- *Interpreter*: This data extraction technique only concerns model execution tracing approaches considering executable modeling languages with operational semantics. In this technique, execution traces are constructed by the interpreter of an executable modeling language (i.e., by the executable modeling language's operational semantics), or by the execution engine responsible for executing the operational semantics of an executable modeling language.

- *External tool*: The information to be recorded in an execution trace is provided by an external tool. Such an external tool could be, for instance, a model checker.

- *Other*: This attribute is assigned to approaches that use none of the data extraction techniques represented by the other attributes.

Trace Representation Format (Q6) This attribute set refers to the kind of format used for the representation of execution traces. We categorized the trace representation formats of model execution tracing approaches using the following attributes.

- *Metamodel*: the data structure for representing traces is defined using a metamodel.
- *Text format*: the data structure for representing traces is defined through some well-defined text format. In particular, approaches defining a formal grammar for representing traces or producing traces in the form of well-structured log outputs fall into this category.
- *Other*: this attribute is assigned to model execution tracing approaches that use trace representation formats other than the ones captured by the attributes given above.
- *Unknown*: this attribute is assigned to approaches where the associated primary studies do not mention the used trace representation format.

Trace Representation Method (Q7) This attribute set refers to the method used for defining the trace representation format. It includes the following attributes.

- *FR (framework)*: refers to approach that provide a framework for defining custom trace representation formats.
- *AG (automatically generated)*: refers to approach that automatically generate a trace representation format for a given executable modeling language.
- *MD (manually developed)*: refers to approaches that use a trace representation format manually developed for the respective approach.
- *AE (already existing)*: refers to approaches that rely on some existing trace representation format.
- *Unknown*: refers to approaches where the associated primary studies do not mention the used trace representation method.

Language Specificity of Trace Structure (Q8) This attribute set categorizes model execution tracing approaches concerning the language-specificity of the used trace data structure. For this categorization, we defined the following attributes.

- *Language-independent*: refers to approaches that either rely on generic data structures for representing execution traces, i.e., data structures that can be used to represent execution traces of models conforming to any executable modeling language, or approaches that support the creation of executable modeling language-specific trace data structures but for any executable modeling language.

- *Language-specific*: refers to approaches that rely on a data structure specific to the tracing of models conforming to a particular executable modeling language.

- *Specific to a certain kind of language*: refers to approaches that rely on a data structure that is specific to the tracing of models conforming to a particular kind of executable modeling languages, i.e., trace data structures that do not only support the tracing of models conforming to a single executable modeling language but that are not general enough to trace models of any executable modeling language.

Data carrier format (Q9) This attribute set refers to the format used to store execution traces. Based on the investigated primary studies, we identified the following used data carrier formats.

- *Text*: refers to approaches storing execution traces in simple text files.

- *XML*: refers to approaches storing execution traces in XML syntax, which includes, for instance, XMI files.

- *Database*: refers to approaches storing execution traces in databases.

- *Unknown*: refers to approaches where the associated primary studies do not discuss the supported data carrier formats.

Maturity Level (Q10) With this attribute set, we capture whether model execution tracing approaches offer tool support and how mature this tool support is. To measure the maturity level of approaches, we used the four-level scale proposed by Cuadros Lopéz et al. [98] defined as follows:

- *Level 1 (not implemented)*: The approach is not implemented in a tool.
- *Level 2 (partially implemented)*: The approach is implemented in a prototype tool but not all features are supported.
- *Level 3 (fully implemented)*: The approach is completely implemented in a tool. The tool has been used for several applications to validate the approach.
- *Level 4 (used in industrial practice)*: The approach is completely implemented in a tool and the tool has been used in industrial practice.

4.2.2.3 Article Classification

This step comprised the assignment of attributes to the selected primary studies, the summarization of primary studies into distinct model execution tracing approaches, and the analysis of the resulting classification of investigated approaches to summarize the research body.

Attribute Assignment We classified the selected primary studies based on the attribute sets defined above. We achieved this by reading the complete content of the primary studies. In particular, each primary study was thoroughly reviewed by one of the authors, who also assigned the attributes to the respective primary study. At least one of the other authors reviewed the resulting classification. Whenever there was a disagreement, an in-depth discussion was done to reach a consensus among the authors and make sure the classification is correct. In some cases, we had to review the same primary study several times to make sure that we interpret its content correctly. The classification was done in a spreadsheet that was shared among the authors and also used to keep notes about additional details of the primary studies and exchange comments on individual classifications.

Summarization of primary studies There are cases where multiple primary studies present the same model execution tracing approach but on different levels of detail or in different development stages. For instance, some journal articles selected as primary studies are extensions of earlier

work of the authors published in the proceedings of conferences or workshops, which were also selected as primary studies. Thus, we decided to summarize such tightly related primary studies as one approach and analyze the classification of approaches instead of the classification of primary studies. Thereby the classification of an approach is the union of the attributes assigned to all primary studies summarized in this approach. This summarization step yielded 33 approaches from the 64 selected primary studies.

Classification results The attribute assignment and summarization of the 64 selected primary studies resulted in a classification of 33 approaches, which can be regarded as the body of knowledge in model execution tracing. The results of the classification are presented in Figure 4.7, which shows the frequency in which the individual attributes have been assigned to the investigated model execution tracing approaches per research question. These results are discussed in detail in Section 6. The attributes assigned to each individual approach are shown in Table 4.2 (for Q1-Q3), Table 4.3 (for Q4-Q5), and Table 4.4 (for Q6-Q10), which are given in the end of this chapter. In these tables, the investigated approaches are numbered from A01 to A33.

All artifacts prepared for this work including the spreadsheet containing the classification of the selected primary studies and bibliographic information have been collected in a replication package that has been made publicly available[1].

4.3 Results

In this section, we discuss the classification results of the investigated approaches per research question as given in Figure 4.7 and Table 4.2-4.4.

[1] https://drive.google.com/drive/folders/1wX1xu10bd5vmXp_UDIjRFB2_WhmFBx5-?usp=sharing

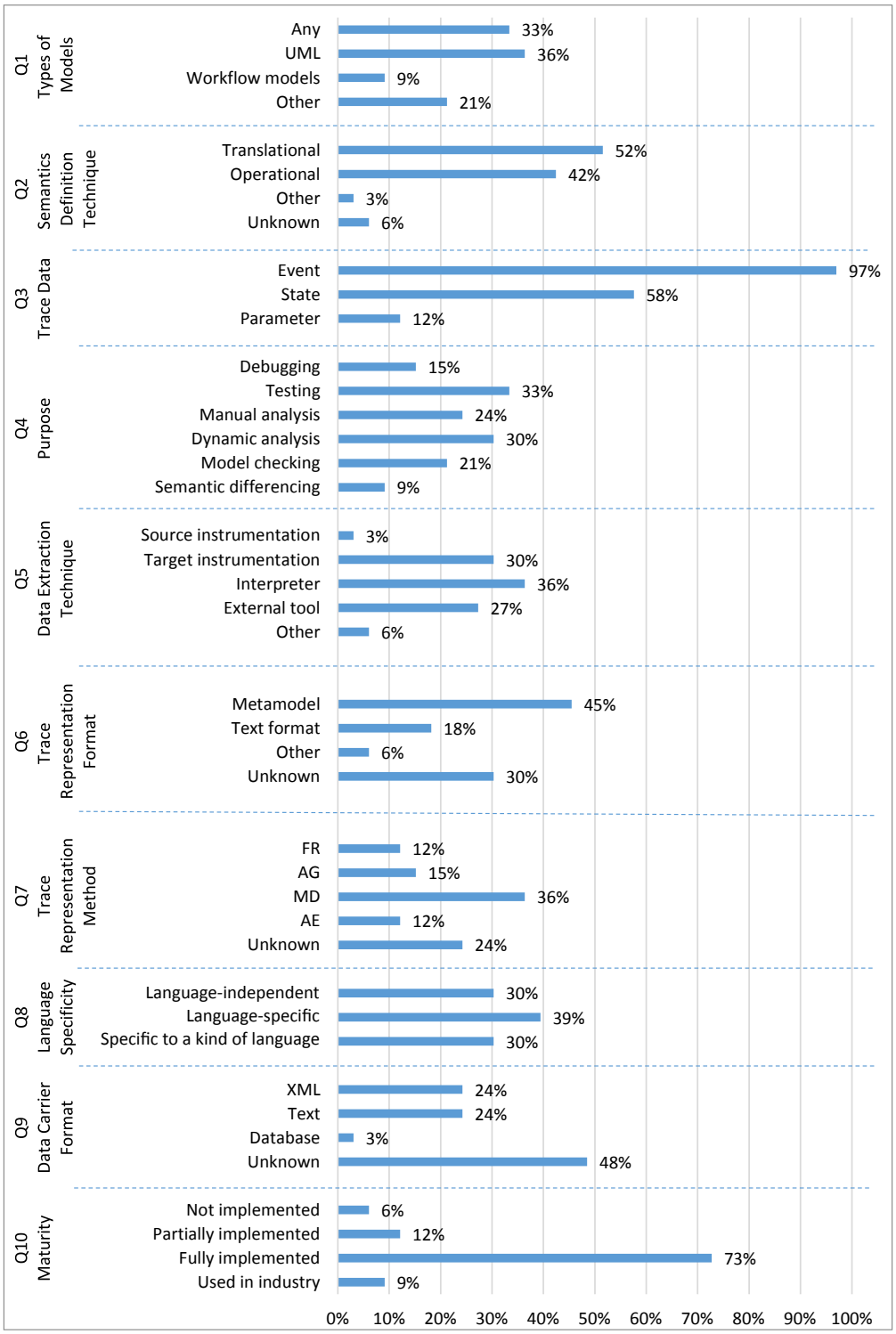

Figure 4.7: Classification of model execution tracing approaches

4.3.1 Types of Models (Q1)

We discovered that concerning the targeted executable modeling languages, the investigated model execution tracing approaches can be classified into three categories where each category comprises around one third of the approaches: 36% of the approaches target UML models, 30% target workflow models or models conforming to other executable modeling languages, and 33% are independent of any executable modeling language.

Most of the approaches targeting UML models consider specifically UML activity diagrams or UML state machines. Other behavioral diagrams, such as UML sequence diagrams, have been a target only by a few approaches in this category.

Only a minority of the approaches, namely 9%, is devoted to workflow models. Three approaches fall into this category. Other executable modeling languages are targeted by 21% approaches: MCSE description models [99] are targeted in A01 [100], stochastic discrete event simulation models in A10 [101, 42], COLA models [102] in A12 [103, 104], CCSL clock constraint specifications in A16 [105, 106, 37], story diagrams in A18 [107], live sequence charts in A28 [108], and Event-B models in A30 [109]. It is worth noting that each of these executable modeling languages is targeted by exactly one of the investigated approaches, i.e., none of them is addressed in two or more approaches.

Especially in recent years it seems that more attention is directed towards approaches that provide generic tracing mechanisms that can be applied on models conforming to any executable modeling language: 64% of these approaches (seven out of eleven) appeared in the last five years (publication dates from 2013 to 2018), while only 36% (four out of eleven) occurred between 2008 and 2011.

4.3.2 Semantics Definition Technique (Q2)

Concerning the semantics definition technique, we discovered that about one half of the investigated model execution tracing approaches assume that the execution semantics of supported executable modeling languages are defined in a translational way (52%), while the other half of the

approaches rely on operational semantics (42%). Only one approach, approach A07 [51, 110, 12], supports both translational and operational semantics. However, the authors introduce only a very abstract pattern of how to design executable modeling languages as well as tracing infrastructures for such executable modeling languages, rather than providing a concrete tracing infrastructure or tooling that could be directly used.

Approach A26 [111, 112] falls into the category "Other", because it considers denotational semantics. For two approaches, we could not identify the supported semantics definition techniques from the related primary studies, which is why they were assigned the category "Unknown".

From this data we conclude that the majority of existing model execution tracing approaches supports executable modeling languages with either operational semantics or translational semantics, while there are no concrete solutions for offering model execution tracing capabilities to executable modeling languages irrespectively of how their execution semantics are defined.

Another interesting finding is that the majority of approaches applicable to any executable modeling language, namely 64%, rely on operational semantics. In contrast, 67% of the approaches targeting UML rely on translational semantics. The latter confirms the finding by Ciccozzi et al. [17] that translational semantics are predominantly used for the execution of UML models—in 85% of the investigated solutions—rather than operational semantics.

4.3.3 Trace Data (Q3)

All of the investigated approaches except one (97%) trace events that occur during the execution of a model. Thereby, 42% of all approaches do only trace execution events and not any further information, i.e., these approaches produce traces that are basically sequences of events that occur during a model execution. Equally many approaches (again 42%) do trace besides execution events also information about the evolution of the execution state of models. Only a very small fraction of the investigated studies, namely 12% capture rich traces that record execution events, execution states, as well as inputs and outputs.

4.3.4 Purpose (Q4)

Most of the investigated model execution tracing approaches have been applied for realizing either testing techniques or dynamic analysis techniques with 33% and 30% of approaches, respectively. They are followed by 24% of approaches applied for manual analysis and 21% of approaches applied for model checking. Only few approaches have been applied for debugging and semantic model differencing, namely 15% and 9%.

The low number of approaches applied for debugging and semantic model differencing can be explained with two reasons: First, only specific kinds of debugging techniques actually require traces with omniscient debugging and debugging previous execution (i.e., replaying executions) being the most prominent examples. Second, semantic model differencing is a rather young research area with only few proposals made so far.

Interestingly, most of the investigated approaches, namely 70%, have been applied on one kind of model analysis technique only, whereas only 27% of approaches have been applied to realize two different kinds model analysis techniques. The most common combination of model analysis techniques is the combination of manual analysis and dynamic analysis, which is reported by 15% of the investigated approaches (or 56% of the approaches applied on a combination of two analysis techniques). All other combinations are only reported for one approach each. It is also worth mentioning that manual analysis is the model analysis technique that has been most often combined with other model analysis techniques (in 21% of all approaches or 78% of approaches applied on a combination of two analysis techniques).

There is only one approach, A20 [39, 14, 113, 114, 115], which uses traces recorded with the proposed model execution tracing approach for realizing three different kinds of model analysis techniques, namely testing, dynamic analysis, and debugging.

Another interesting finding is that more than half of the investigated model execution tracing approaches that are applicable to any executable modeling language have been applied for model checking, namely 55%. The other model analysis techniques are only considered by 1-2 approaches each. In contrast, half of the approaches targeting UML have been applied for model

testing, followed by 33% of approaches applied for manual analysis, and 25% of approaches applied for dynamic analysis. Only two of the approaches targeting UML have been applied for debugging, and one each for model checking and semantic model differencing.

4.3.5 Data Extraction Techniques (Q5)

The most common data extraction techniques used by the investigate model execution tracing approaches are tracing by an interpreter (36%), tracing through target instrumentation (30%), and tracing by an external tool (27%).

As already discussed in Section 4.2.2.2, tracing by an interpreter only concerns executable modeling languages with operational semantics. From the model execution tracing approaches supporting operational semantics, 86% rely on this data extraction technique. The other 14% rely on external tools for extracting the runtime data to be traced.

Similarly, tracing through target instrumentation is only applicable for approaches supporting translational semantics. From the approaches supporting translational semantics, 59% rely on target instrumentation for data extraction. The majority of the remaining model execution tracing approaches supporting translational semantics extract runtime data through external tools, namely 29%. Only one approach supporting translational semantics, approach A02 [116], relies on source instrumentation for data extraction, and one approach, approach A12 [103, 104], relies on a middleware that is part of the targeted execution platform for data extraction and is hence classified as "Other" for the data extraction technique.

It is also worth mentioning that approach A07 [51, 110, 12], which supports both translational and operational semantics (cf. Section 4.3.2), uses a model checker (i.e., an external tool) for constructing traces when execution semantics are defined in a translational way, and construct traces through the interpreter when the execution semantics are defined in an operational way.

Looking at the "External Tool" category, 44% of the approaches assigned to this category support only translational semantics, 22% support only operational semantics, and 11% (1 approach) support both translational and operational semantics. For 22% of the approaches extracting data

through external tools, the supported semantics definition technique is unknown.

For the "Other" category, we already mentioned approach A12 [103, 104], which supports translational semantics and relies on a middleware for extracting the runtime information to trace. The second approach assigned to this category, approach A26 [111, 112], supports denotational semantics and traces are directly constructed through the denotational semantics implemented in Haskell.

From this data, we conclude that for model execution tracing approaches supporting operational semantics, runtime data is extracted primarily by the interpreter, while for approaches supporting translational semantics, runtime data is extracted primarily through target instrumentation or external tools.

4.3.6 Trace Representation Format (Q6)

Our results on trace representation formats clearly indicate that metamodels are most frequently used to define the data structure for representing model execution traces. In particular, 45% of the investigated approaches rely on metamodels. This result was expected, since we study execution tracing approaches for executable models and executable models commonly instantiate metamodels as well.

More surprisingly, only 18% of model execution tracing approaches define textual trace representation formats and from these approaches, only one approach, approach A26 [111, 112], actually defines a grammar for representing traces while the others use a less formal trace format, such as structured logs.

Approach A10 [101, 42] directly uses an XML format for representing traces and approach A31 [117, 118] uses a graph-based representation. These two approaches (6%) have been classified as "Other".

For almost one third of the investigated approaches (30%), we could not identify the used trace representation format. Hence, these approaches have been assigned to the category "Unknown" for Q6.

4.3.7 Trace Representation Method (Q7)

Our classification results show that nearly half of the investigated model execution tracing approaches (48%) use a selected trace representation format. In particular, more than one third of the approaches (36%) use trace representation formats that were manually developed particularly for the respective approach, while only 12% rely on existing trace formats. An example of the latter case is approach A30 [109] which reuses event trace diagrams (ETDs) proposed by Rumbaugh et al. [119]) as trace format.

In contrast, 27% of approaches support custom trace representation formats that are tailored towards the executable modeling language used to define the traced executable models. In particular, 15% of approaches automatically generate trace representation formats for executable modeling languages, while 12% of approaches provide a framework for manually defining custom trace representation formats. However, it has to be noted that this kind of approaches are a minority. An example of an automated approach is the generative approach A23 [10, 120, 121, 7, 122], which automatically derives multidimensional domain-specific trace metamodels for xDSMLs. Approach A22 [123, 124, 125, 126] is an example of a framework for manually defining custom trace formats. In particular, it allows the definition of textual trace formats for UML models using so-called *trace directives*.

For the last category of approaches comprising 24%, the trace representation method is unknown, i.e., not mentioned by the associated primary studies.

4.3.8 Language Specificity of Trace Structure (Q8)

Concerning the language specificity of the used trace data structure, the investigated approaches can be categorized into three groups of almost equal size: 39% of the approaches use a trace data structure that includes concepts specific to a certain executable modeling language, 30% of the approaches use a trace data structure that can be reused for executable modeling languages of a specific kind, and 30% of the approaches use a trace data structure that is independent of any executable modeling language. This applies to the approaches A09 [127, 128, 45], A12 [103, 104],

A20 [39, 14, 113, 114, 115], and A21 [61, 129]. Approach A02 [116] is the only one that does not trace execution events but only execution st

The majority of the approaches using a trace data structure specific to a particular executable modeling language target the UML language, namely 61%. In contrast, no particular trend could be observed for the approaches using trace data structures specific to a certain kind of executable modeling language. The approaches using a language-independent trace format target no particular executable modeling language or type of executable modeling language but support any executable modeling language.

4.3.9 Data Carrier Format (Q9)

Only little information could be extracted form the primary studies concerning used data carrier formats for storing execution traces: For almost half of the approaches (48%), no data carrier formats have been mentioned.

The other half of the investigated approaches either uses an XML format or plain text format for storing execution traces, namely 24% each. Only one approach, approach A19 [130], persist execution traces in a database.

n this approach, a UML profile is generated for tracing system executions using a UML state machines. A persistence component transmits the runtime data obtained from the execution to a trace database.

4.3.10 Maturity Level (Q10)

Our results for the classification of model execution tracing approaches concerning maturity level show that the majority of investigated approaches (73%) is fully implemented. From this we conclude that the field of model execution tracing has reached a moderate level of maturity. However, among the investigated approaches, only three (9%) have been subjected to an empirical validation in industrial settings. In fact, the most common method used to validate the investigated approaches is through case studies. While the majority of the case studies demonstrate the com-

plete implementation of the approaches, they fail to show the approaches' usefulness in industrial settings. Hence, little is known about the value of existing model execution tracing approaches for industry and evaluations of this aspect are hence needed to further mature this research field.

4.4 Future Research Directions

In the following, we discuss directions for future work on model execution tracing, building upon our research results presented in Section 4.3. In our opinion, it is necessary to address the following topics not only from a research perspective, but in collaboration with tool vendors and end users to ensure widespread tool support and to achieve industry adoption.

Scalable trace data structure: From the results obtained for research question Q3 on the kind of traced data, we can see that in fact a lot of data is recorded by existing model execution tracing approaches: Almost all approaches record information about occurred execution events and more than 40% keep detailed execution state information. This means, however, that traces are expected to grow large, which may cause scalability issues in both memory needed for storing traces and time needed for processing them. Nevertheless, we could only identify three of the investigated approaches that aim at addressing these scalability issues. In particular, Bousse et al. aim to address this issue in their approach A23 [10, 120, 121, 7, 122] by sharing data among captured states so that only changes in data are recorded. Similarly, Hegedus et al. propose in their approach A14 [52] to reduce traced state information by only capturing state modifications and events related to state modifications.

In contrast, Kemper and Tepper propose in their approach A10 [101, 42] to remove repetitive fragments from traces using heuristic methods, such as cycle reduction. While the aforementioned approaches consider some sort of trace compaction, they utilize different optimization potentials and leave open whether the achievable compaction is sufficient for industry applications. Thus, we see the need for more detailed studies on scalable model execution tracing solutions.

Common trace exchange format: The results obtained for research question Q7 on trace

representation methods clearly shows that existing model execution tracing approaches rely on their own custom trace formats being either defined manually or generated automatically. Only four of the investigated approaches reuse some already existing trace format.

Giving this large variety of used trace formats, it seems apparent that a common format for exchanging model execution traces is needed. Such an exchange format, however, has to support the representation of executable modeling language-specific concerns in different levels of detail. This is indicated by the results obtained for the research questions Q8 and Q3 that show that existing model execution tracing approaches record information specific to particular executable modeling languages or kinds of executable modeling languages, and that besides execution events, information about execution states and processed inputs are also relevant to be traced in specific contexts. A common trace format should be expressive enough to capture the required runtime information for any executable modeling language. Also, it should represent traces in a compact form to enable scalability of the analysis tools. Thereby, scalability should be considered as a key requirement when defining a common trace format. Examples of trace formats for traces generated from code-centric approaches are Compact Trace Format(CTF) proposed by Hamou-Lhadj and Lethbridge [150, 151] and Message Passing Interface Trace Format (MTF) proposed by Alawneh and Hamou-Lhadj [152]. These trace formats model traces of routine calls and inter-process traces, respectively, in a compact way, in order to facilitate efficient interchange of traces among trace analysis tools. A similar effort should be invested in defining standard trace formats for traces of model executions that would facilitate interoperability among V&V tools and hence make V&V tools available to a broader user base.

Extended support for semantics definition techniques: Our results for research question Q2 show that all approaches except one support either translational semantics or operational semantics but not both. Executable languages use many different techniques for defining operational semantics/ interpreters (e.g., programming languages, action languages, and model transformation languages) and translational semantics/ compilers (e.g., model-to-model transformation languages, target modeling languages, code generators, target programming languages) [14]. Model execution

tracing approaches focus on one of these techniques (either translational or operational). Therefore, they are applicable in a very narrow scope. By supporting different semantics definition techniques, we can reuse the same model tracing approach in more scenarios. It is based on the separation of concerns principle in order to separate the concern of how to implement an executable language (i.e., semantics definition technique) from how to trace model executions.

This would also enable the application of V&V tools for executable modeling languages defined with either semantics definition technique.

Empirical validation: While the majority of investigated model execution tracing approaches has been implemented in prototype tools, there exists very little empirical evidence about the usefulness of these approaches. Even with complete demonstrations, a considerable amount (more than 90%) of the approaches lack any empirical validation in industrial settings, as shown by our results obtained for research question Q10. To mature the field of model execution tracing, empirical validations of existing solutions need to be performed. We intend in the future to work on investigating the state of adoption of model execution tracing in industry. We also intend to work with developers of model-driven systems to understand the state of practice of model execution tracing and what the challenges developers face when using the related techniques.

4.5 Limitations and Threats to Validity

Despite the care taken in the definition of the research method, our mapping study is subject to known threats and limitations. The most serious threats to the validity of our research results are researchers' bias in searching, selecting, and classifying studies. To mitigate this risk, we applied and strictly followed the guidelines suggested by Kitchenham and Charters [91] and Petersen et al. [92].

To mitigate the risk of missing relevant studies, we have performed automated searches in the most popular digital online libraries in the field of software engineering. The search strings used for this have been derived from the defined research questions. Furthermore, we have performed a

forward snowballing step to identify additional studies that could be relevant for our research.

To ensure the reliability of our selection criteria for primary studies as well as ensure the quality of selected primary studies, we have also performed a pilot study and a quality assessment. Through the pilot study, we could improve the original selection criteria and the quality assessment showed that the selected primary studies are of good quality.

In order to reduce the threat of misclassifying the selected primary studies, we reviewed their full texts thoroughly instead of reviewing only abstracts, introductions, and conclusions. Furthermore, the classification of each primary study was reviewed by at least one of the authors that was not involved in the original classification. Any discrepancies were resolved by reading the affected primary study again and discussing its classification in detail.

4.6 Related Work

There exist several systematic review studies that have been conducted in the context of MDE. However, none of these studies target approaches for model execution tracing approaches. To the best of our knowledge, this is the first study that aims to survey the state of the art in this area. In this section, we summarize recent surveys in the domain of MDE that are related to our study.

Ciccozzi et al. [17] conduced a systematic review of research studies and tools concerned with the execution of UML models, which is also considered in 36% of model execution tracing approaches investigated in our work. The authors analyzed the identified research studies on UML model execution concerning publication trends, technical characteristics, and evidence provided on industry adoption. Tools were analyzed concerning technical characteristics only, which include among others UML modeling (required diagrams, use of action languages, etc.), execution strategy (translation, interpretation, execution tools and technologies, etc.), intended benefits, and readiness level. The findings show a growing scientific interest in UML model execution starting from 2008, which is consistent with our findings for model execution tracing, which show an increase in publications on the topic from 2007. Furthermore, the study revealed that translational

semantics has been predominantly used for the execution of UML models rather than operational semantics. Also this finding is consistent with the results of our study. As intended benefits of UML model execution, the study identified reducing the effort for producing executable artifacts and improving the functional correctness of models as the main benefits targeted, the latter being highly related to model execution tracing as it provides the basis for many dynamic V&V techniques used for ensuring functional correctness. However, the study does not investigate in detail the types of V&V techniques provided or applied by the identified research studies and tools. It only investigates whether model-level interactive debugging, model simulation (i.e., execution of models for analysis rather than execution on the target platform), and formal specification languages (e.g., for the purpose of model checking) are supported with the result that model-level interactive debugging and formal specification languages are only supported by few approaches while model simulation is supported by half of the approaches letting the authors suggest that model execution is considered beneficial for early design assessment. In contrast, we investigate in more detail which kinds of dynamic V&V techniques are realized based on model execution tracing. Interestingly, the authors identify the need to further enhance the observability of execution models, which includes the ability to record, play back, and analyze execution traces of system operation on the target platform or in a simulation. The last finding that we want to highlight is that most of the analyzed UML model execution solutions have a low technology readiness level and that only a few of the investigated research studies provide evidence through experimentation in industrial settings and based on empirical evaluations. This is also true for the model execution tracing approaches studied in our work.

Szvetits and Zdun [153] applied a systematic literature review for the purpose of classifying and analyzing existing approaches for using models at runtime for self-adapting systems. The existing approaches have been classified concerning objectives, techniques, architectures, and kinds of models used. The authors revealed the usage of different kinds of models at runtime to achieve various objectives, such as adaptation, policy checking, and error handling. Related to our study, the authors identified monitoring as one objective of models at runtime defining monitoring as

the activity of monitoring "the system by using models which help to trace application behavior". However, other than model execution tracing targeted in this study, models at runtime trace the execution of an application rather than the execution of models. As discussed by the authors, this distinction becomes blurry when executable models are in fact the executable application, i.e. when there is no other implementation-level artifact manually or automatically generated from executable models. The authors point out that in such scenarios, it is not clear how model execution fits into the models at runtime paradigm. However, model execution in general definitely has a role in the models at runtime paradigm, as it facilitates the analysis of model-based representations of application behavior to, for example, simulate the consequences of runtime adaptations or to predict system properties influencing runtime adaptation. Nevertheless, we do not see a direct relationship between model execution tracing and models at runtime.

Dias Neto et al. [154] conducted a systematic review of Model-Based Testing (MBT) approaches. The reviewed approaches have been categorized concerning testing level, tool support, application scope, kind of models used for test case generation, used test coverage criteria, used test case generation criteria, and level of automation. Among other findings, the study revealed that there is a need for providing MBT solutions for testing non-functional requirements, such as usability, security, and reliability, and that most MBT approaches have not been evaluated empirically or transferred to industry. Recently, Gurbuz and Tekinerdogan [155] conducted a systematic mapping study to identify and analyze the state of the art in MBT for software safety. The study revealed that 42% of the investigated primary studies were validated using industrial evidence but that they nevertheless provide no strong evidence of positive effects of MBT for software safety. Related to these findings, we have identified two model execution tracing solutions applied for MBT: In [108], MBT is used for testing functional requirements, while in [133], security testing is considered. Furthermore, we have found most existing model execution tracing approaches to also lack an empirical evaluation in industrial settings.

Nguyen et al. [156] conducted a systematic literature review on Model-Driven Security (MDS), which is the application of MDE techniques and technologies to the development of secure sys-

tems. The authors categorized the identified MDS approaches concerning considered security concerns, applied modeling approach, used model-to-model and model-to-text transformations, application domains, and evaluation methods. The results revealed the need for addressing several security concerns that have been mostly neglected by existing approaches, as well as multiple security concerns simultaneously and in a systematic manner. Furthermore, they discovered a lack of tool support and empirical evaluations. Related to our work, the authors identified that DSMLs play a key role in MDS but that most of the currently existing security DSMLs lack semantic foundation required for automated analysis. They also point out that behavioral models are rarely used but most approaches employ structural models only, which hampers the ability to deal with multiple security concerns simultaneously.

Nascimento et al. [157] conducted a systematic mapping study on Domain Specific Languages (DSLs) to identify existing DSLs, their application domains, tools for their development and usage, as well as techniques, methods, and processes for creating, applying, evolving and extending DSLs. The study surveys DSLs on a high level of abstractions and provides only a high-level overview of existing DSLs and DSL engineering approaches. While DSMLs are identified as one specific kind of DSLs, no investigations targeting specifically executable modeling languages, model execution, or model execution tracing have been performed.

Giraldo et al. [158] conducted a systematic review to identify definitions of quality in MDE. The authors discovered that only 16 out of 134 reviewed studies provide explicit definitions of quality and that these definitions mostly concern the quality of models or the quality of modeling languages. The remaining studies do not provide such an explicit definition. Among them, 40% of the studies propose solutions for quality assurance, such as behavioral verification of models, performance models, and model metrics. This study is related to our work in the sense that model execution tracing approaches in general aim at enabling the performance of dynamic V&V for ensuring the quality of models or modeled systems in terms of functional or non-functional quality properties. To investigate this aspect, we review in this study the objectives of existing model execution tracing solutions.

Santiago et al. [96] conducted a systematic literature review to analyze the current state of the art in the management of traceability in MDE approaches. However, other than in our work, the considered notation of traceability refers to the establishment of relationships between products of the development process and is hence unrelated with the tracing of model executions.

4.7 Conclusion

In this chapter, we presented a systematic mapping study on existing approaches for the tracing executable models. With this study we aim at identifying and classifying the existing approaches, thereby assessing the state of the art in this area, as well as pointing to promising directions for further research in this area.

From 645 research studies found through automatic searches in popular academic online libraries, we finally selected and analyzed 64 primary studies that present 33 unique model execution tracing approaches. These 33 identified approaches were classified concerning supported types of models, supported execution semantics definition technique, traced data, purpose, data extraction technique, trace representation format, trace representation method, language specificity, data carrier format, and maturity.

Our findings show that (i) the majority of approaches target specific executable modeling languages with UML being the most popular one; (ii) model execution tracing approaches either support exclusively executable modeling languages with operational semantics or executable modeling languages with translational semantics; (iii) besides execution events traced by almost all approaches, a significant amount of approaches also record detailed information about execution states; (iv) the majority of model execution tracing approaches has been applied on one kind of model analysis technique only with testing and dynamic analysis being the most frequently used ones; (v) approaches supporting operational semantics rely mainly on executable modeling language interpreters for extracting tracing information, while approaches supporting translational semantics rely mostly on target instrumentation; (vi) metamodels are most frequently used for

defining the trace representation format; (vii) thereby, trace representation formats are mostly specific to the respective approach with only a minority of approaches that reuse existing formats; (viii) trace representation formats are mostly dependent on the supported executable modeling language or specific to a certain kind of executable modeling languages; (ix) traces are serialized either in XML format or as plain text; and (x) only a small minority of approaches have been empirically validated in industrial settings.

The results suggest that more research work is needed particularly on suitable trace representations and broad applicability of approaches with scalability and interoperability being two concerns that have been mostly neglected so far. Furthermore, empirical validations of the usefulness of approaches in real application scenarios are needed to foster the adoption of model execution tracing approaches in practice.

Table 4.2: Classification of model execution tracing approaches for Q1-Q3

Approach	Types of Models (Q1)				Semantics Definition Technique (Q2)				Trace Data (Q3)		
	Any	UML models	Workflow models	Other	Translational	Operational	Other	Unknown	Event	State	Parameter
A01 [100]				*	*				*		
A02 [116]			*		*					*	
A03 [131, 132]		*			*				*	*	
A04 [133]		*			*				*		
A05 [134, 135, 136, 137]		*				*			*	*	
A06 [138]		*				*			*	*	
A07 [51, 110, 12]	*				*	*			*		
A08 [139]	*					*			*		
A09 [127, 128, 45]	*				*				*	*	*
A10 [101, 42]				*				*	*	*	
A11 [140]		*			*				*		
A12 [103, 104]				*	*				*	*	*
A13 [13]	*				*				*		
A14 [52]	*					*			*		
A15 [141]		*			*				*		
A16 [105, 106, 37]				*		*			*		
A17 [142, 143]		*				*			*		
A18 [107]				*		*			*	*	
A19 [130]		*			*				*	*	
A20 [39, 14, 113, 114, 115]		*				*			*	*	*
A21 [61, 129]		*			*				*	*	*
A22 [123, 124, 125, 126]		*			*				*	*	
A23 [10, 120, 121, 7, 122]	*					*			*	*	
A24 [53, 54]	*							*	*	*	
A25 [144]	*					*			*	*	
A26 [111, 112]			*				*		*		
A27 [41, 145]	*					*			*	*	
A28 [108]				*		*			*	*	
A29 [6, 146]	*					*			*	*	
A30 [109]				*	*				*		
A31 [117, 118]			*		*				*		
A32 [147, 148]		*			*				*		
A33 [149]	*				*				*	*	

Table 4.3: Classification of model execution tracing approaches for Q4-Q5

Approach	Purpose (Q4)						Data Extraction Technique (Q5)				
	Debugging	Testing	Manual analysis	Dynamic analysis	Model checking	Semantic differencing	Source instrumentation	Target instrumentation	Interpreter	External tool	Other
A01 [100]				*				*			
A02 [116]		*					*				
A03 [131, 132]			*		*			*			
A04 [133]		*						*			
A05 [134, 135, 136, 137]	*								*		
A06 [138]			*	*					*		
A07 [51, 110, 12]			*	*					*	*	
A08 [139]					*				*		
A09 [127, 128, 45]			*	*				*			
A10 [101, 42]			*	*						*	
A11 [140]		*	*					*			
A12 [103, 104]	*										*
A13 [13]					*					*	
A14 [52]					*				*		
A15 [141]			*					*			
A16 [105, 106, 37]				*					*		
A17 [142, 143]		*							*		
A18 [107]	*								*		
A19 [130]				*				*			
A20 [39, 14, 113, 114, 115]	*	*		*					*		
A21 [61, 129]						*				*	
A22 [123, 124, 125, 126]		*						*			
A23 [10, 120, 121, 7, 122]	*					*			*		
A24 [53, 54]		*			*					*	
A25 [144]		*								*	
A26 [111, 112]				*							*
A27 [41, 145]					*					*	
A28 [108]		*							*		
A29 [6, 146]						*			*		
A30 [109]		*								*	
A31 [117, 118]			*	*				*			
A32 [147, 148]		*						*			
A33 [149]					*					*	

Table 4.4: Classification of model execution tracing approaches for Q6-Q10

Approach	Trace Representation Format (Q6)				Trace Representation Method (Q7)					Language Specificity (Q8)			Data Carrier Format (Q9)				Maturity (Q10)
	Metamodel	Text format	Other	Unknown	FR	AG	MD	AE	Unknown	Language-independent	Language-specific	Specific kind of language	Text	XML	Database	Unknown	
A01 [100]				*					*			*				*	3
A02 [116]				*					*			*				*	2
A03 [131, 132]		*							*			*	*				3
A04 [133]				*			*					*				*	1
A05 [134, 135, 136, 137]				*					*		*					*	4
A06 [138]		*					*				*		*				3
A07 [51, 110, 12]	*				*					*				*			3
A08 [139]	*						*			*				*			3
A09 [127, 128, 45]		*					*			*			*				3
A10 [101, 42]			*				*					*		*			3
A11 [140]				*					*			*				*	2
A12 [103, 104]	*						*				*					*	3
A13 [13]	*				*					*						*	3
A14 [52]	*				*					*			*				3
A15 [141]				*			*				*		*				3
A16 [105, 106, 37]		*						*			*					*	3
A17 [142, 143]	*						*				*			*			3
A18 [107]				*					*		*					*	1
A19 [130]	*					*					*				*		2
A20 [39, 14, 113, 114, 115]	*						*				*			*			3
A21 [61, 129]				*					*		*					*	3
A22 [123, 124, 125, 126]		*			*						*		*				3
A23 [10, 120, 121, 7, 122]	*					*				*				*			3
A24 [53, 54]	*					*				*						*	3
A25 [144]	*					*				*						*	2
A26 [111, 112]		*					*					*	*				3
A27 [41, 145]	*					*				*						*	3
A28 [108]				*				*			*					*	3
A29 [6, 146]	*							*		*				*			3
A30 [109]	*							*			*					*	3
A31 [117, 118]			*				*					*	*				4
A32 [147, 148]				*					*			*				*	3
A33 [149]	*						*					*		*			4

Chapter 5

Generic Compact Trace Metamodel

In this chapter, we present our second contribution, which is the generic Compact Trace Metamodel (CTM) supported by any given xDSML. In Section 5.1, we introduce the context of our contribution and our proposal, and motivate the problem by presenting the requirements for defining CTM. Then, Section 5.2 gives our research approach. Section 5.3 presents our generic trace metamodel, and explain the intuition of our idea regarding the compaction techniques. Continuing, we present CTM and explain how it is provided by applying a set of compaction techniques. Finally, Section 5.4 discuss related work. Section 5.5 concludes the chapter.

5.1 Motivation

In this section, we first give requirements for our approach to define a new trace metamodel, and then we explain the limitations of existing approaches.

5.1.1 Requirements for an execution trace metamodel

In previous chapter, we conducted a systematic survey on model execution tracing approaches and highlighted challenges that need to be addressed when constructing and manipulating execution traces.

The first challenge is that existing model tracing approaches use different formats for representing traces, which hinders interoperability. Having a common exchange format for model execution traces would allow better synergy among V&V tools that rely on execution traces, and hence makes V&V tools available to a broader user base.

Such an exchange format, however, has to support the representation of xDSML-specific concepts at different levels of detail. In other words, a common trace format should be expressive enough to capture the required runtime information for any xDSMl. Traces are known to be large, which cause scalability problems. A large amount of data generated from the execution of a model complicates the process of applying dynamic V&V techniques.

A common trace format must represent traces in a compact form to enable scalability of the analysis tools. Therefore, scalability is of primary importance and calls techniques for a compact representation of traces when defining a common trace format. The preservation of the original information in a trace is also important when applying compaction techniques.

In summary, we considered the following requirements in the design of CTM.

Genericity: CTM should support a wide range of xDSMLs, independent of the meta-programming approaches used for their implementation.

Scalability in space: CTM should handle large execution traces.

Information preservation: CTM should provide a lossless representation of traces.

Performance overhead: The performance overhead caused by using CTM to construct an execution trace should be an acceptable overhead during the execution of a model.

5.1.2 Limitation of existing trace structures

Techniques exist for defining data structures to represent execution traces of models conforming to a given xDSML. For instance, a trace structure may be described using an XML schema as proposed by Kemper and Tepper [101], a text format as proposed by Maoz et al. [45, 127], or metamodels such as the one's proposed by Hegedus et al. [85]. However, the results of our survey on trace representation formats [159] indicate that metamodels are most frequently used to define the data structure for representing model execution traces. In this work, we focus on traces for executable models. As executable models commonly instantiate metamodels, we discuss the limitations of current trace metamodels.

Existing generic trace metamodels. A very few studies (e.g., [5], [6]) propose generic trace metamodels, independent from an xDSML. Although they allow interoperability between existing trace analysis tools, they do not scale up to large traces efficiently. Also, these trace metamodels only capture events that occur during an execution, and lack a complete representation of a trace such as execution states as well as inputs and outputs values.

Existing domain-specific trace metamodels. There exist studies that define trace metamodels including concepts specific to a given xDSML. They rely on their own custom trace formats being either defined manually or generated automatically. This lack of genericity hinders interoperability and the sharing of data among tools that support multiple xDSMLs. Moreover, according to our survey [159], a large number of these techniques record information about occurred execution events and keep detailed execution states information, resulting in scalability problems. For example, in the ProMoBox approach proposed by Meyers et al. [41], a domain-specific trace metamodel is automatically generated for a given xDSML, but such metamodel defines a trace as a sequence of snapshots of the complete executed model to capture execution states.

We identified three approaches that aim at addressing the scalability issue. In particular, Hegedus et al. propose in their approach [85] to reduce traced state information by only capturing state modifications and events related to state modifications. Similarly, Bousse et al. [7] propose a technique to reduce the impact of this problem by sharing data among captured states so that only changes in the data are recorded. Kemper and Tepper [101] use heuristics such as cycle reduction to remove repetitive fragments from traces. While these approaches consider some sort of trace compaction, they still suffer from scalability problems due to the repetitions in the data, questioning whether the achievable compaction is sufficient. In addition, none of these approaches provide a generic exchange format.

We see the need for more scalable generic model execution tracing solutions. The contribution presented in this chapter aims at addressing this need, defining a generic trace metamodel that not only provides a detailed representation of trace but also scales up to large traces by applying

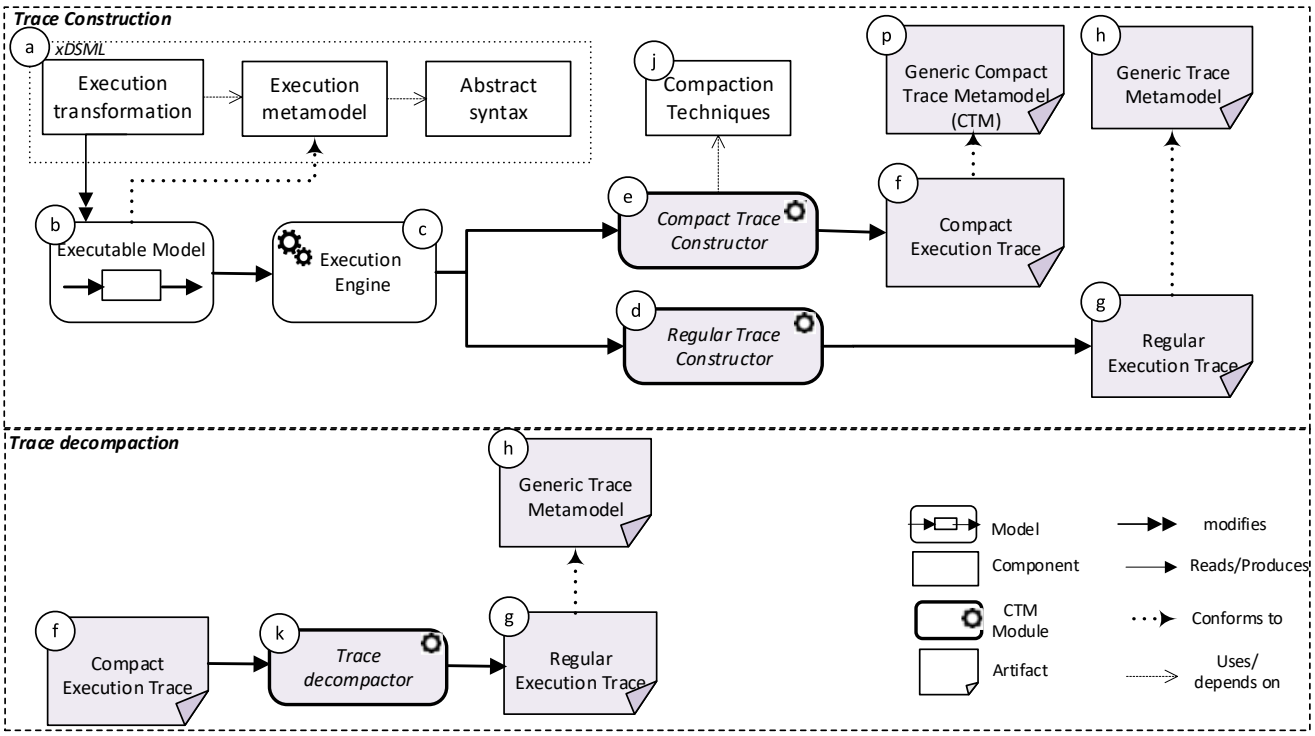

Figure 5.1: Approach overview, with our contributions highlighted in gray

compaction techniques.

5.2 Overview of the Approach

To overcome the limitations observed in existing trace formats, and to better comply with the requirements mentioned in section 5.1.1, we propose a new trace format that can be use to represent traces in a generic and scalable fashion. Our idea relies on the fact that there might be a lot of repetitions in traces. Thereby, we apply a set of compaction techniques to store the repetitive information contained in a trace only once, leading to reduce the size of traces effectively.

Figure 5.1 presents a complete overview of our approach with our contributions highlighted in gray.

For the execution of models, the first step is the definition of an xDSML (a) including the abstract syntax, execution metamodel, and execution transformation. Then, an executable model (b)

conforming to the execution metamodel of the xDSML can be executed in an execution engine (c). The execution transformation is applied to modify the execution state of the model.

There exist two trace constructors in our approach, each generating execution traces of a model. The first one is the *regular trace constructor* (d) that allows constructing traces without compaction. The result is a *regular execution trace* (g) conforming to our proposed *generic trace metamodel* (h).

Using a set of *compaction techniques* (j), the *compact trace constructor* (e) creates traces in a compact representation form. Note that the *compact trace constructor* contains several units, each dealing with the compaction of a part of traces concerning to evolution of the execution state of a model, parameter values as well as loop detection within traces, which will be described in Sec. 5.3.2. Finally, the result of using the *compact trace constructor* is a *compact execution trace* (f) conforming to the *generic compact trace metamodel* (p).

The second part of our approach consists of a *trace de-compactor* (k) that takes a compact execution trace, and generates a regular trace by decompressing the trace. The *trace de-compactor* contains several modules, each reconstructing the corresponding part of a trace and generating a regular trace from the compact one without losing data. The result is a regular execution trace (g) conforming to our generic trace metamodel (h).

It is worth noting that both trace metamodels marked by (h) and (p) support genericity, while CTM takes into account scalability criterion as well. Besides the construction of regular traces, the generic trace metamodel is used for evaluating information preservation of CTM, so that the traces reconstructed after the de-compaction process with the traces generated from the generic trace metamodel is compared to indicate whether these two traces do match, i.e., CTM provides a lossless representation of traces.

In the next section, we present the gray elements in more detail.

5.3 Generic Compact Trace Metamodel

This section explains a two-step process for designing CTM with the aim of supporting the genericity and scalability criteria described in Section 5.1. In the first step, to address the genericity criterion, we identified runtime concepts required for expressing model execution traces that are common to existing executable modeling languages. The result is a generic trace metamodel, which is explained in Section 5.3.1. In the second step, we enhance this generic metamodel with compaction techniques in order to fulfill the scalability prerequisite. The result is CTM, which is described in Section 5.3.2.

5.3.1 Generic trace metamodel

In order to define a generic trace metamodel, we identify all runtime concepts that are required for expressing the trace of executing models that are common in all executable modeling languages. After that, we define their relationships, and create a metamodel.

Figure 5.2 shows our proposed generic trace metamodel. The root class of the metamodel is **Trace**, which contains a sequence of states of the model under execution (**State**) as well as a sequence of events related to the states (**Step**). **Step** is a class used for representing execution steps. Using a tree structure, a **Step** can include other steps represented by the reference `children`. The reference `state` between **State** and **Step** is used to specify the starting and ending state of a step.

A state contains the states of all dynamic objects (**ObjectState**) at a given point in time of the execution. Thereby, the state of a dynamic object is given by the current values of all its dynamic fields. An **ObjectState** represents the state of a specific object, whereas a **State** represents the state of all objects of a model. At any given point in the execution, the state of an object of the executed model is defined by the values of all its dynamic fields (e.g., `tokens` values in a Petri net).

An **ObjectState** object is related to a **TransientObject**, which corresponds to an object of the executed model. We distinguish between **StaticTransientObject** and **DynamicTransientObject**. The **StaticTransientObject** class refers to objects that are defined in the executed model, while

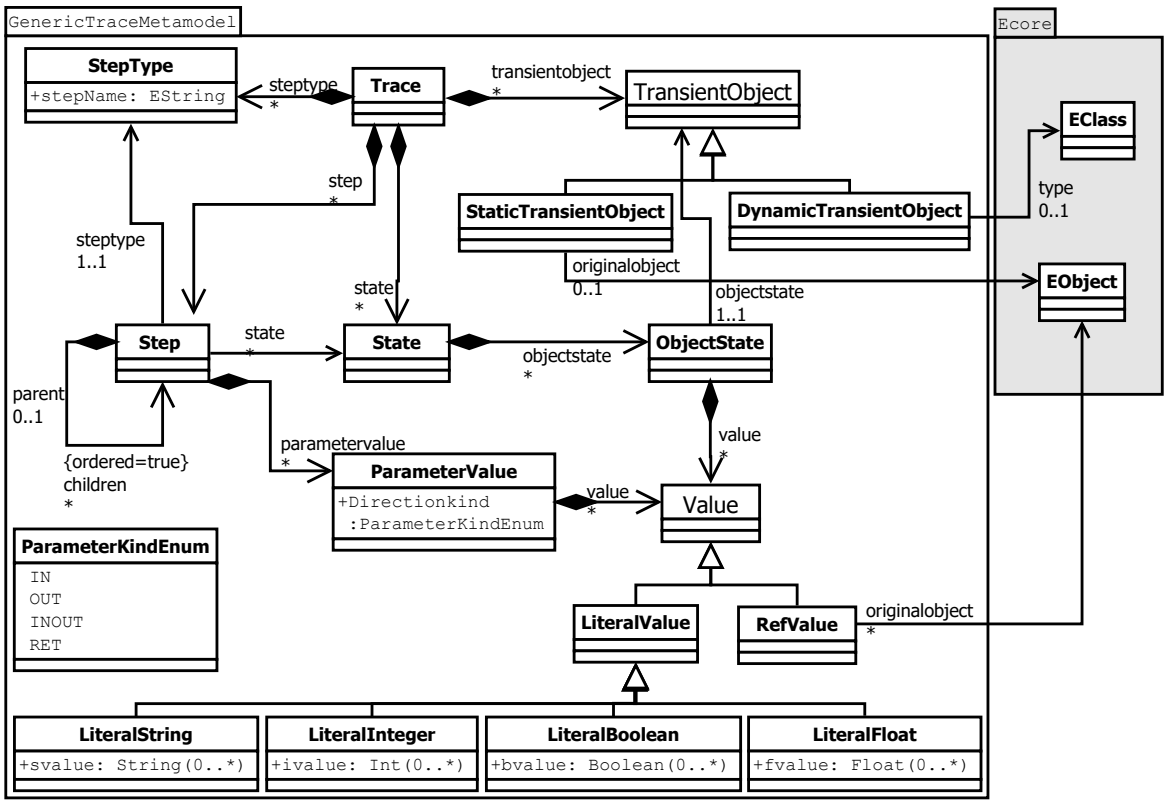

Figure 5.2: CTM generic trace metamodel

the **DynamicTransientObject** class refers to objects that are only created during execution.

When creating an execution trace, one **StaticTransientObject** is created for each object existing in the model. The relationship between the **StaticTransientObject** and the original model object is stored with the reference **originalobject** to Ecore's metaclass **EObject**. Note that all objects contained in a model, which have been defined by an Ecore metamodel, inherit from EObject. Similarly, one **DynamicTransientObject** is created for each dynamic object created during the execution. The type of the object is stored using the reference `type` to Ecore's metaclass **EClass** to represent the objects created only during execution.

For example, in a Petri net model, each **StaticTransientObject** object is linked to the Place object whose states is captured. Besides, no object is created during the execution of a Petri net model. Therefore, the trace does not contain any **DynamicTransientObject** objects.

Similarly, one **DynamicTransientObject** is created for each dynamic object created during the

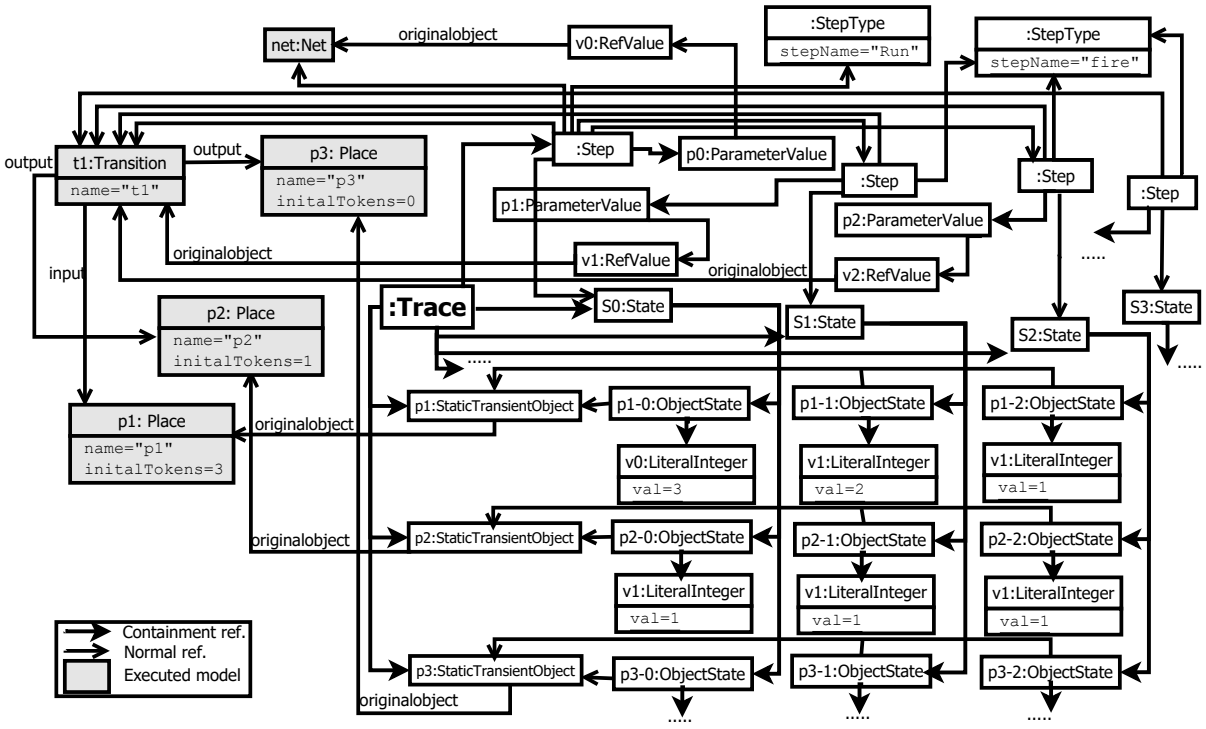

Figure 5.3: Excerpt of execution trace of the Petri net example that conforms to the proposed generic trace metamodel

execution. The type of the object is stored using the reference `type`.

The values of dynamic fields are stored as elements typed by the abstract class **Value**, which can either be a **LiteralValue** or a **RefValue**. The class **LiteralValue** is an abstract class for defining literal values; each containing an attribute referring to a sequence of values of a particular primitive type. For example, the class **LiteralBoolean** is for the specification of either a Boolean value or a sequence (array) of Boolean values. Similarly, the class **RefValue** represents references among objects.

The inputs and outputs of an execution step are recorded using the **ParameterValue** class containing the Enum field `directionkind` representing the parameter type (input, output, input-output, return) and the `value` reference pointing to the **Value** class. The **StepType** class is used to represent the type of each step, which is recorded only once in a trace, instead of storing it for each step instance.

Example: To show how this metamodel can be used to capture trace elements of an executable

model, we consider the previous Petri net example model shown in Figure 2.2. Figure 5.3 illustrates an excerpt of the trace obtained from the execution of the Petri net model. Using an object diagram, we show the content of the executed model at the left of the figure, and the generated trace of the model at the right of the figure. The **Trace** root contains one root **Step** for the application of the execution rule *run*, which itself contains three nested **Step** elements representing the firing of transition *t1*. Thus, the trace contains four **Step** elements. One **Step** is linked to **StepType** *Run* representing the complete Petri net *run*, and three steps are linked to the **StepType** *fire* representing the firing of the transition *t1*. The excerpt of the trace also shows three of the recorded **StaticTransientObject** objects, one per **Place** object *p1*, *p2*, and *p3*. Note that no object is created during the execution of a Petri net model. Therefore, the trace does not contain any **DynamicTransientObject** objects.

The trace contains four **State** objects with three **ObjectState** objects; each representing the current value of the `tokens` field of the respective **Place** object. The `tokens` value is represented by using **LiteralInteger** objects.

To represent the **ParameterValues** of steps, four **ParameterValue** objects are created, one pointing to the **Net** object provided to the *run* rule and three pointing to transition *t1* provided to the *fire* rule. The **Net** and **Transition** are referenced using **RefValue** objects.

Overall, using our metamodel, we needed 46 objects and 70 references to represent the trace generated during the execution of the Petri net example model. As shown in Figure 5.3, there exist many repetitions in the trace, particularly in the **ObjectState**, **State**, **ParameterValue** and **Value**. Additionnally, there are repetitions of **Steps** due to the existence of a loop in the model, causing the Transition t1 to be fired three times. We show in the next subsection how this generic metamodel is enhanced with compaction techniques in order to reduce the size of the trace model.

5.3.2 CTM Compaction

To reduce the size of CTM traces, we propose a multi-part compaction strategy by applying customized compaction techniques to different parts of the generic trace metamodel defined in the

previous section. The key idea is to compact repetitive parts of a trace.

5.3.2.1 Dealing with Repetitions in State

The first part of our compaction strategy focuses on execution states. As explained previously, each **State** contains the states of all objects in the executed model after each execution step. Since it is likely to have unchanged dynamic properties of objects in a given step, there might be a lot of repetitions in **State** objects.

As an example, to represent the states of the trace from the example model (Figure 5.3), we needed 31 objects (4 State, 12 ObjectState, 12 Value, 3 StaticTransientObject) and 64 references (4 state, 24 objectstate, 24 value, 12 transientobject). After firing *t1* for the first time, the state of *p1* and *p3* changes but the state of *p2* remains unchanged. Similarly, the state of *p2* remains unchanged after firing *t1* for the second time. Instead of storing all **ObjectState** objects that represent a new state of the executed model, we can design a technique that captures only the modification (delta) between two states. For the example model shown in Figure 5.3, this means storing only the **ObjectState** for *p1* and *p3* at each execution step.

Figure 5.4 shows the excerpt of the adapted trace metamodel dealing with the compaction of **State** information. The new concepts and relationships in comparison to the generic trace metamodel (shown in Figure 5.2) are highlighted in blue. First, the redundancies of **ObjectState** objects are reduced by adding a containment reference `objectstate` to the class **Trace** which allows to create **ObjectState** objects that are equivalent only once. Moreover, an **ObjectState** might be the same for different objects, meaning that the values of their corresponding fields are the same. To support this, we add the class **TransientObjectState** between the class **State** and the class **ObjectState** and a reference `transientobject` to the class **TransientObject**. More precisely, instead of having a reference from the class **ObjectState** to the class **TransientObject**, the class **TransientObjectState** defines the relationship between these two classs. Such structure allows to use an **ObjectState** object for different **TransientObject** objects. The **TransientObjectState** objects are only created for the **TransientObject** objects that have changed in the current state.

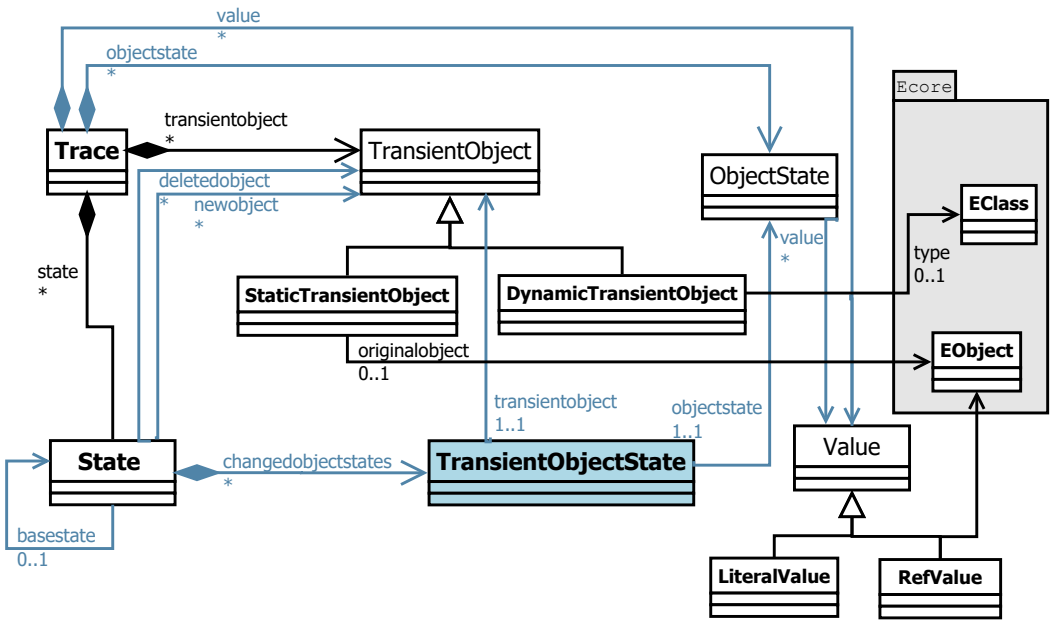

Figure 5.4: Excerpt of CTM modeling concepts related to **State** with the changes highlighted in blue

The changes of objects can be obtained by comparing their respective **ObjectState** objects in the current **State** with the **ObjectState** objects belonging to the previous **State**. However, instead of the previous **State**, we can inspect the most similar **State** within the execution trace to obtain delta **State** objects. To do this, we add a reference `basestate` to the class **State**, specifying the **State** object that is the closest to the current **State** object (which can be achieved by comparing their **ObjectState** objects).

For the compaction of **States**, we used a notification framework to track the changes that are made to the dynamic objects of a model during an execution. This helped us to represent only the modifications between states.Note that we implemented an additional procedure that gives the same functionality as the notification framework. The procedure is independent of the execution environment, and can be used for the State compaction (instead of the notification framework) in the case of not using Gemoc Studio. The value of `basestate` reference is determined by using an algorithm that compares the current **State** with other existing **State**s in the trace, and finds the closest one to the current **State** object. The algorithm scans **State** objects within the trace,

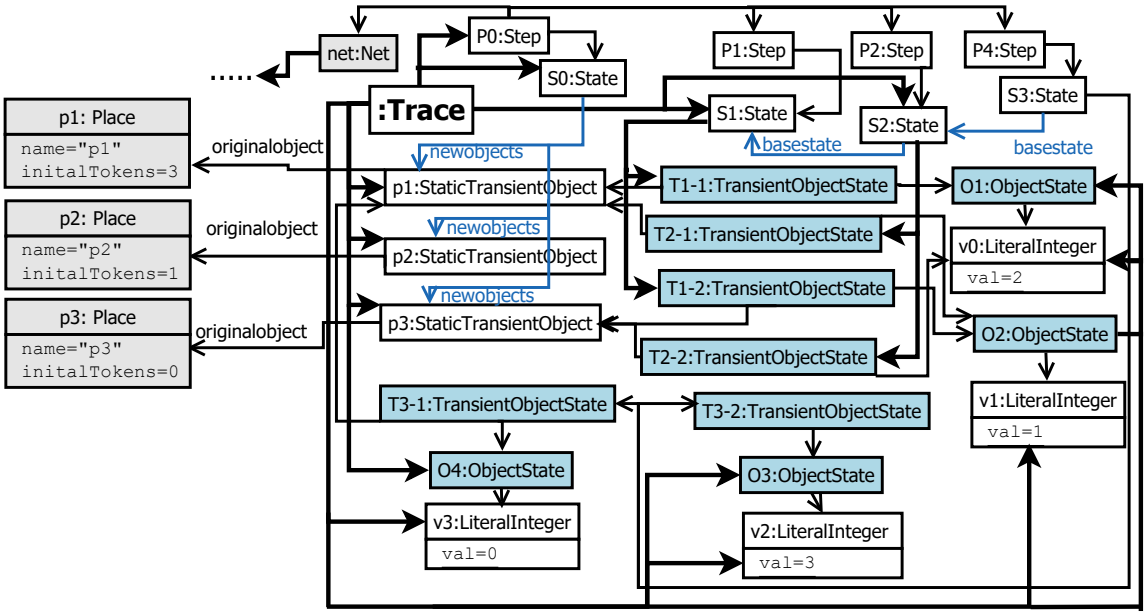

Figure 5.5: Excerpt of execution trace of the Petri net example (**State** with compaction)

compares the **ObjectState**s and **Value**s contained in the chosen **State** with those of the current **State**, and find the closet **State** object.

Finally, a **State** object stores the objects newly created in the state, as well as the objects deleted in the state, using the new references `newobjects` and `deletedobjects` pointing to the class **TransientObject**. Therefore, instead of creating **ObjectState** objects referenced by respective **State** object, new objects can be simply specified using the reference `newobjects`.

The direct benefit of this structure is that we avoid redundancies by creating a single **ObjectState** per value change. Another benefit is that the **ObjectState** objects can be shared between different **TransientObjects**. It also supports exploring previous states of an executed model.

Figure 5.5 shows an excerpt of the trace of the Petri net example model, conforming to the part of the CTM shown in Figure 5.4. The trace illustrated in Figure 5.5 is a compact version of the trace that was presented in Figure 5.3. The blue elements denote the elements used for representing states in a compact form. Four references are used to represent the new objects referencing to the first **State** object at the beginning of the execution. As compared to Figure 5.3, the first state is represented by using four `newobjects` references, and no **ObjectState** objects are cre-

ated. To represent the second **State** object, two **TransientObjectState** objects, one referring to the **ObjectState** of *p1* and the other one referring to the **ObjectState** of *p3* are linked to the state *S1*. Similarly, two **TransientObjectState** objects are used to represent the **ObjectState** for *p1* and *p3* after executing the next two execution steps. As shown in the figure, **ObjectState** objects are shared between different **State** objects. For instance, the **ObjectState** O2 is shared between two **State** objects by using the object T1-2 referring to *p3* for the first **State** (S1) and the object T2-1 referring to *p1* for the second **State** (S2).

In total, the new compact structure reduces the number of objects from 46 objects to 21, and the number of references from 70 references to 30.

5.3.2.2 Dealing with Repetitions in Step

The next part of our compaction strategy focuses on repetitions appearing due to the existence of loops and patterns of identical sequences of events, and recurring patterns. For our Petri net example, as we can see in Figure 5.3, there are three repetitions, caused when the transition *t1* is fired.

To achieve this, we adopted the *Flyweight design pattern* [160] and the *Composite design pattern* [160] to implement a hierarchical structure for **Step** objects in terms of a directed-acyclic graph with shared leaf nodes. The idea is to remove the repetitions by collapsing repeated nodes into one node, and storing the repeated parts only once. To better present our technique, we use the following definitions:

Step patterns (i.e., sequences of execution steps repeated consecutively in a trace) are represented using the StepPattern class. Two sequences are considered as instances of the same pattern if they contain the same steps in the same order.

RepeatingStep: A StepPattern includes a sequence of **Step** objects, named RepeatingSteps. We differentiate RepeatingStep from the **Step** class. A **Step** (as shown in Figure 5.2) represents its **StepType**, **State** and **ParameterValue** as well. In contrast, for a **RepeatingStep**, only its **StepType** is represented. In fact, because a **RepeatingStep** might be included in several **StepPattern**

objects, it can occur in different parts of an execution; each containing different **State** objects and **ParameterValue** objects. Therefore, to represent a **Step**, which belongs to a **StepPattern**, we use **RepeatingStep** (instead of **Step**).

PatternOccurrence: This class represents the instances of a step pattern. A StepPattern can occur more than once in a trace. PatternOccurrence objects are instances of StepPattern objects, which are the occurrences of the patterns invoked in the trace.

We introduced these concepts to act as a basis for supporting patterns in a trace. Indeed, a trace might include several **PatternOccurrence** objects; each referring to a **StepPattern** object. The instance of PatternOccurrence shows part of a trace that the pattern occurs as well as the starting point of the pattern. In each **StepPattern** object, there might exist several **RepeatingStep** objects. In the following, we briefly discuss how to apply the new concepts in the generic trace metamodel:

As shown in Figure 5.6, we define a new class **StepPattern** pointing to repeating patterns as a sequence of **RepeatingStep** objects repeated consecutively in the trace. We also add a new class **RepeatingStep**, referring to the **Step** objects included in a **StepPattern**. Similar to **Step**, **RepeatingStep** is a tree-like structure, which implies having a composite reference to represent `parent` and `children` references. In addition, we rely on the containment reference `steppattern` of the **Trace** class to remove redundancy in **StepPattern** objects. A **RepeatingStep** can be shared between several **StepPattern** objects by adding the containment reference `repeatingstep` to the **Trace** class.

The reference `repeatingstep` between the **StepPattern** and the **RepeatingStep** classes represents which **RepeatingStep** objects are included in each **StepPattern** object.

To support repetitive patterns within an execution trace, we need to distinguish between a normal step from a step that refers to an occurrence of a step pattern. We do this by extending the **Step** class with two subclasses, **NormalStep** and **PatternOccurrence**. The **PatternOccurrence** class represents occurrence of the patterns, and contains an attribute `rept` that is used to specify the number of repetitions of a pattern.

Each **PatternOccurence** object is related to a **StepPattern** object using the reference `pat-`

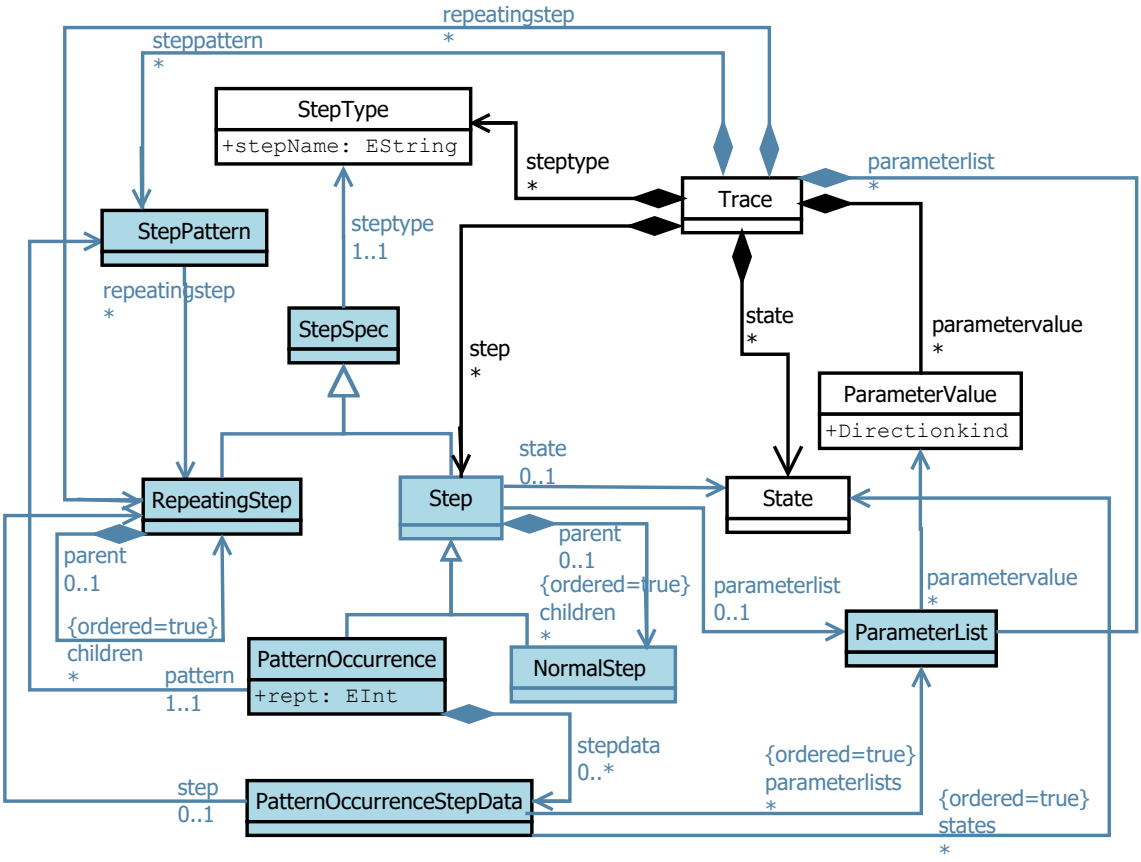

Figure 5.6: Excerpt of CTM with modeling concepts related to **Step**, with the changes highlighted in blue

`tern`. Besides, there is a reference `state` between the **Step** and **State** classes, pointing to the state of the model at any point in time for the respective **NormalStep**.

Despite the similarity of **Step** objects in a loop, they could lead to different states (**State** objects) and process/produce different parameters (**ParameterValue** objects). Because each **Step** might include more than one **ParameterValue**, we need a new class **ParameterList**, which refers to a list of corresponding **ParameterValue** objects. This class is used to merge **ParameterList** objects of **Step** objects included in a loop. Similar to the technique used by Taniguchi et al. [161] for abstracting repetition patterns, in order to replace the whole repetition, we make a representative by unifying **Step** objects (by adding a reference to the **RepeatingStep** class), and storing the corresponding **State**s and **ParameterValue**s in chronological order sequences. More precisely, a **PatternOccurrence** object points to a specific **StepPattern** object, which contains a set of **RepeatingStep** objects. In subsequent iterations of the pattern, a sequence of **State** objects and a sequence of **ParameterList** objects are obtained for each **RepeatingStep** object. This data is represented by using the class **PatternOccurrenceStepData**, having a reference to the **RepeatingStep** class, an ordered unbounded reference `state` to the **State** class, and an ordered unbounded reference `parameterlist` to the **ParameterList** class as well. All associated **State**s and **ParameterList**s are stored chronologically for a single **RepeatingStep**. Using this structure, we replace multiple redundant instances of steps by the references to a single step. By storing repeated steps only once, we are able to eliminate all repetitions of steps within the trace.

Since both **NormalStep** and **RepeatingStep** have a reference to the **StepType** class, we add the **StepSpec** superclass, which inherits either **NormalStep** or **RepeatingStep**. We also add the reference `steptype` from the **StepSpec** class to the **StepType** class.

Figure 5.7 presents part of the compact version of the trace of the Petri net example model that makes use of the introduced compaction of **Step** information. In this example, we make use of one **RepeatingStep** referring to the firing of *t1*, which is repeated three times. There is an instance of the **StepPattern** class that contains only one **RepeatingStep** object. We replace all **Step** objects representing the firing of *t1* with one instance of **PatternOccurrence** that refers to

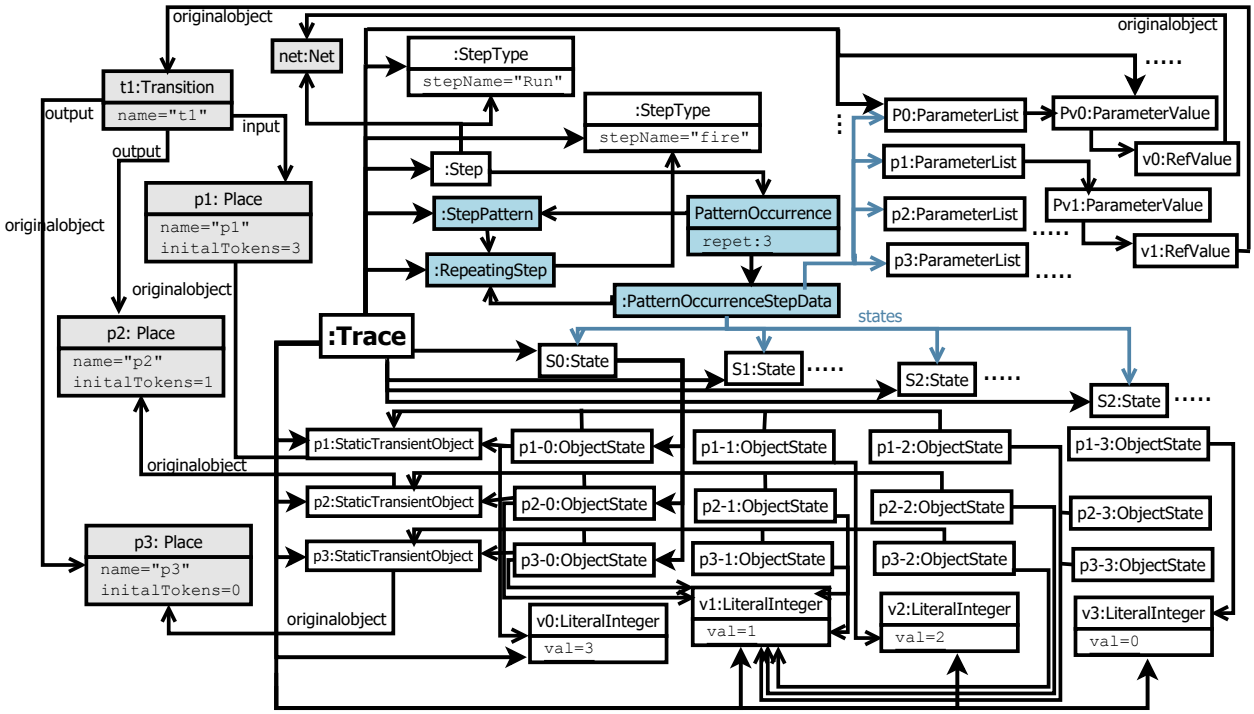

Figure 5.7: Excerpt of an execution trace of the Petri net example model including a loop (**Step** with compaction)

the **StepPattern** object, and stores the value 3 in the `rept` attribute. After each iteration, a **State** object and a **ParameterList** object are created (i.e., S1 and P1 after the first iteration, S2 and P2 after the second iteration, etc.). The **PatternOccurrenceStepData** object represents an order sequence of the **State** objects (i.e., S0, S1, S2, S3), an order sequence of the **ParameterList** objects (i.e., P0, P1, P2, P3) as well as the corresponding **RepeatingStep**.

Compared to the original version of the trace (Figure 5.3), the resulting compact trace requires only five objects and six references to represent the trace as opposed to four objects and 12 references when compaction is not used.

5.3.2.3 Dealing with Repetitions in ObjectState

Our third compaction strategy targets attributes of **ObjectState** objects. There may be redundancies among **ObjectState** objects regarding the values taken by different attributes of each object.

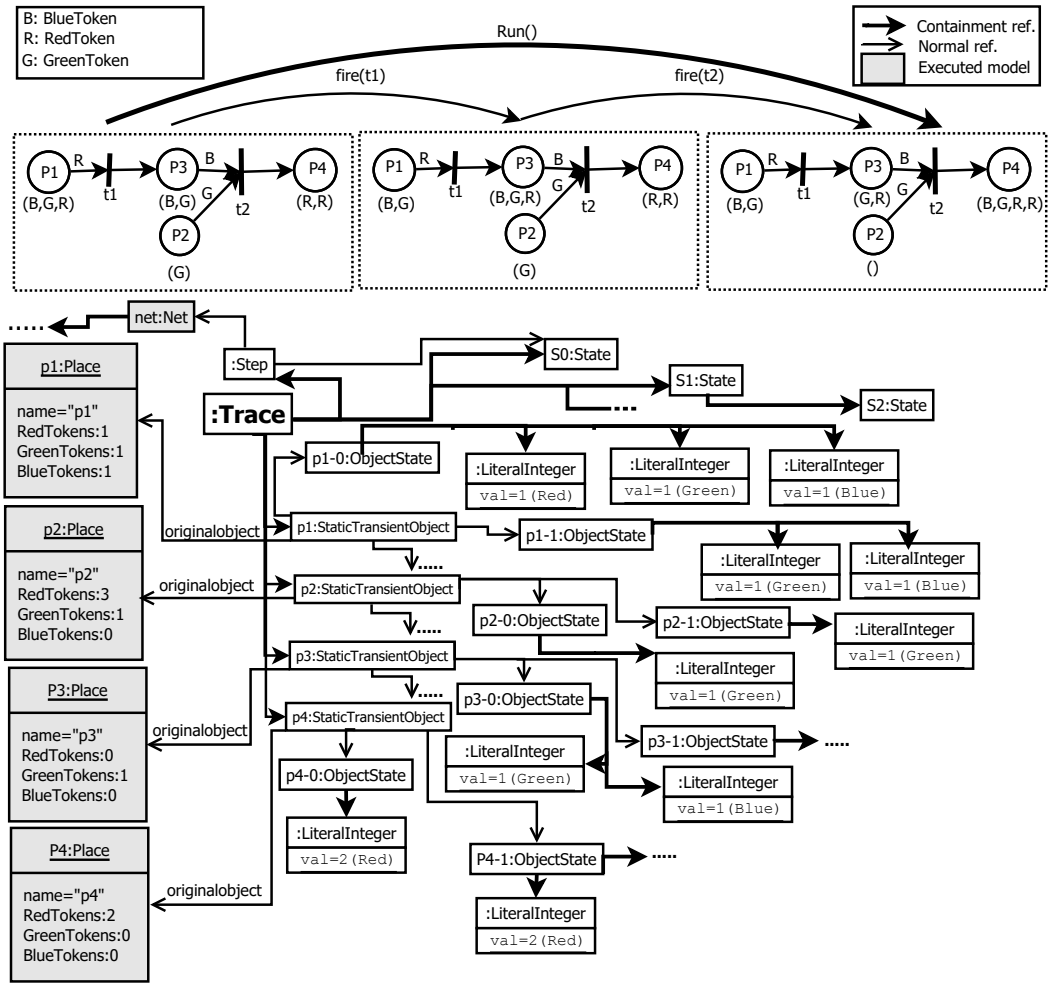

Figure 5.8: Excerpt of execution trace of the colored Petri net example model (**ObjectState** without compaction)

As an example we use a trace of a colored Petri net, shown in Figure 5.8. A colored Petri net is an extension of a Petri net in which each token carries a data value called the *token color*. For simplification reasons, in the figure, we considered both color and value in one object. The **Place** objects are specified with colour set stating the type of tokens. In this example, there are three tokens colors: Red (R), Green (G) and Blue (B). We use a simple representation of the concrete syntax of a colored Petri net to show its execution. The names of **Place** objects are represented inside the circles and the current number of tokens in a **Place** object are shown below the circle by specifying the color of the held tokens. As an example Place *p1* holds in the first state one

Blue, one Green, and one Red token. The **Transitions** among **Places** state which kind of tokens are required at the input **Places** to enable the **Transitions**. In our example model, *t1* is fired if *p1* contains a token with red color. In this case, when *t1* fires, it consumes one token with Red color, and adds one Red token to its output places.

Table 5.1: Excerpt of ObjectState data for the Place objects captured during the execution of the colored Petri net model shown in Figure 5.8

Id	BlueToken	GreenToken	RedToken
P1-0	1	1	1
p2-0	0	1	0
p3-0	1	1	0
p4-0	0	0	2
p1-1	1	1	0
p1-1	1	1	0
p2-1	0	1	0
p3-1	1	1	1
p4-1	0	0	2
p1-2	1	1	0
p2-2	0	0	0
p3-2	0	1	1
p4-2	1	1	2

Table 5.1 shows the partial data from the execution of the example model related to the **Place** object that includes three attributes. The first column (Id) shows the step of the execution and the respective **Place**. For instance, P1-0 refers to the **ObjectState** *p1* at the beginning of the model execution. The second to forth columns present the value of different tokens. The rows represent **ObjectState** objects of the corresponding **Place** objects with slight differences. Regardless of the similar rows (e.g., P2-0 and P2-1), there exists rows in the table that are partially similar. For instance, two values of P1-0, P3-0 and P4-2 (BlueToken and GreenToken) are identical. They are different in RedToken value. It is very likely that only a subset of the attributes of an object changes from one execution step to another. Also, there might exist **ObjectState** objects that are identical in two or more values during an execution. Therefore, we can identify frequent values in

ObjectState objects that can be shared and represented only once.

At the bottom of Figure 5.8, we show the content of the executed model and the generated trace, which conforms to our generic trace metamodel. The model is executed in three Steps, each providing a **State** object that contains four ObjectStates reached by *p1* to *p4*. Since each **Place** object contains three attributes, 36 objects are required to represent **Value** objects. For simplicity reasons, some parts of the trace are not shown in Figure 5.8, e.g., the **IntegerValue** objects with "zero" value and the links between **State** and **StaticTransientObject**. In total, 52 objects (12 ObjectState, 36 Value, 4 StaticTransientObject) and 64 references (24 objectstate, 36 value, 4 transientobject) are used for representing this part of the trace.

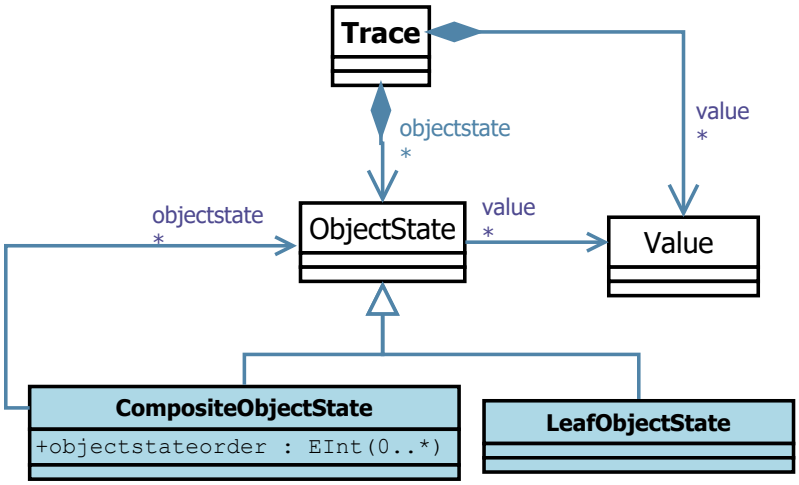

Figure 5.9: Excerpt of CTM with modeling concepts related to **ObjectState**, with the changes highlighted in blue

Figure 5.9 shows our solution for improving CTM generic metamodel by compacting repetitive values of **ObjectState** objects. Our solution was inspired by the Rainstore approach [90], introduced in Section 3.3. As explained in Section 3.3, every unique value in the Rainstor approach is stored only once, and each row of data is shown as a binary tree that allows rebuilding the original data using a breadth-first traversal of the tree. Following the Rainstor method, we decompose the class **ObjectState** into the class **CompositeObjectState** and the class **LeafObjectState**, each

having reference to the class **Value**. More precisely, an **ObjectState** object might consist of a subset of existing **ObjectState** objects or a set of **Value** objects. We model this using the *Composite design pattern* [160]. Each **ObjectState** can be constructed with little effort, by traversing the corresponding **ObjectState** and retrieving its contained **ObjectState**s recursively. Finally, instead of using the containment reference from **CompositeObjectState** to **ObjectState**, the containment reference `objectstate` of the class **Trace** provides the ability to reuse the existing **ObjectState** objects.

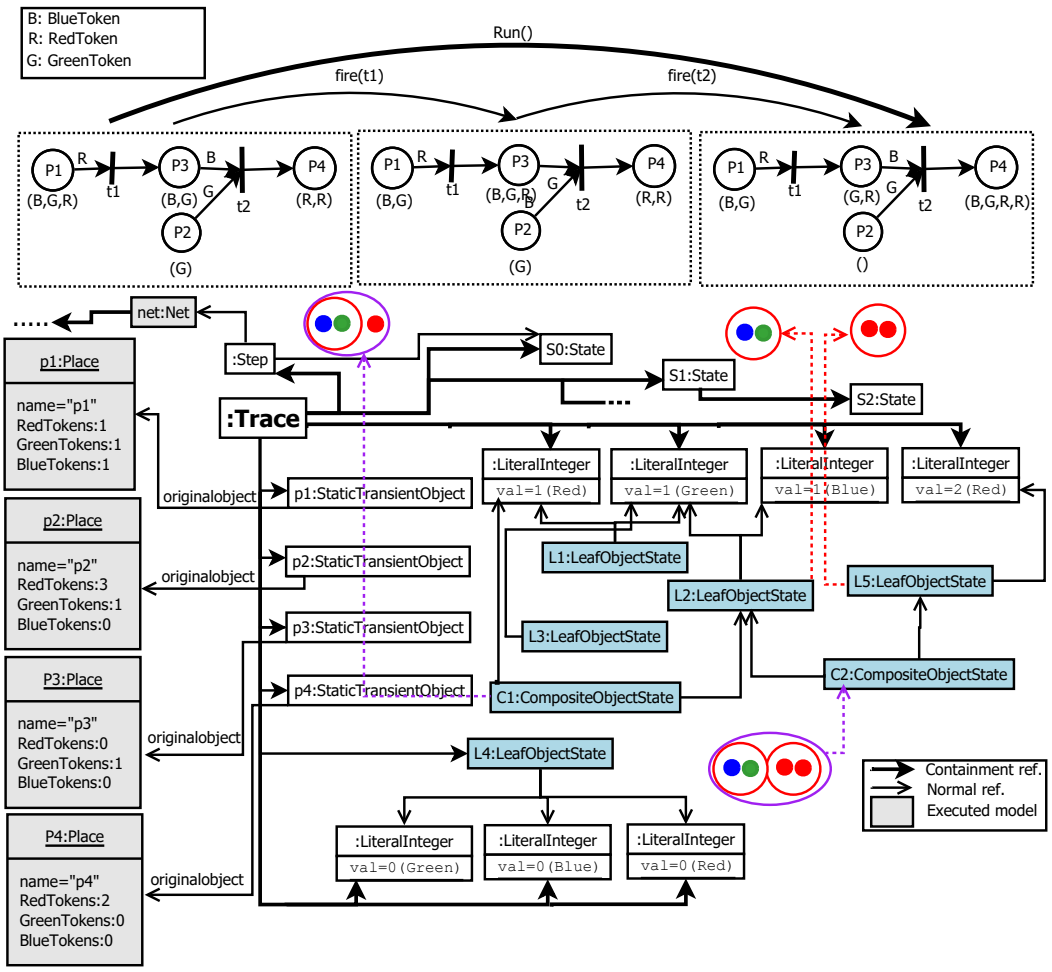

Figure 5.10: Excerpt of an execution trace of the colored Petri net example model (**Objectstate** with compaction)

Note that while two **ObjectStates** may have the same set of values, the order of the values may differ. As an example, consider an **ObjectState** A with values (c1, c2, c3), and an **ObjectState** B with values (c3, c1, c4) in a trace. (c1, c3) are common between A and B but because of the different orders in which they appear, A and B cannot be considered as a shared **ObjectState**. This can be handled by adding a new attribute to the class **CompositeObjectState** named `objectstateorder` defining the actual position of each value, which is obtained after exploring the corresponding **CompositsObjectState**. In our example, A can be represented by a **CompositeObjectState** that consists of a **LeafObjectState** containing (c1, c3) and **Value** c2. In this case, the resulting sequence of values for A is (c1,c3,c2) and the corresponding values taken by the `objectstateorder` attribute are (0,2,1) meaning that c1 in the position 0, c3 in the position 2, and c2 in the position 1 leading to the value order (c1,c2,c3). Similarly, B is represented by a **CompositeObjectState**, having a reference to the same **LeafObjectState** containing (c1, c3) and a reference to **Value** c4. The resulting sequence of values for B is (c1,c3,c4) and the corresponding values of the `objectstateorder` attribute are (1,0,2) meaning that c1 in the position 1, c3 in the position 0, and c4 in the position 2 leading to the value order (c3,c1,c4). To provide better compaction, we do not store any value for the `objectstateorder` attribute in the case that the order of values in the value sequence is the same as the order of values in the corresponding **ObjectState**. This means that in a **CompositeObjectState** object with empty `objectstateorder`, the order of values is the same as the order in which they are retrieved from the contained **ObjectState**s.

Figure 5.10 shows the compact version of the trace of the colored Petri net example model from Figure 5.8. We can see that in the last execution state, *p4* holds one Blue token, one Green token, and two Red tokens. Thereby, the pattern of one Blue token and one Green token can be observed multiple times in the Petri nets execution (see Table 5.1). For instance, *p1* holds one Blue and one Green token in all execution states, and *p3* holds the same kinds of tokens in the first and in the second execution state. To share these token specifications, we define one **LeafObjectState** with one Green token and one Blue token. This **LeafObjectState** is then used to represent all

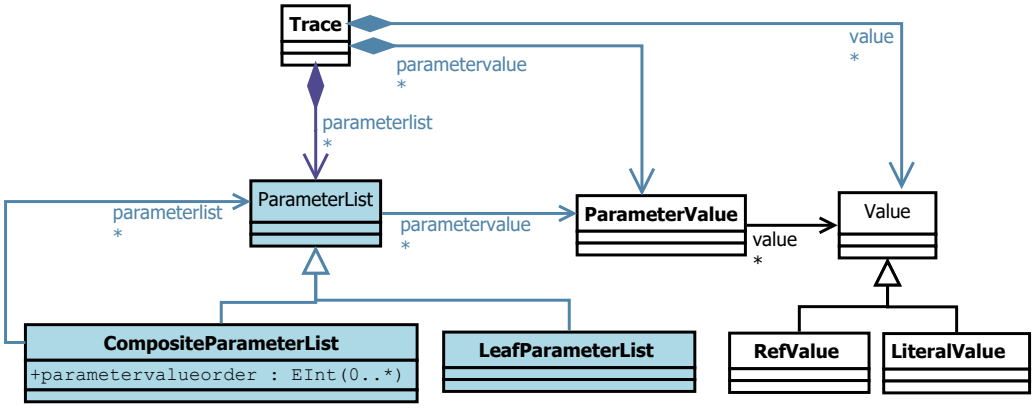

Figure 5.11: Excerpt of CTM with modeling concepts related to **ParameterList**, with the changes highlighted in blue

ObjectStates of **Place** objects where the **Place** objects hold one Green and one Blue token. To illustrate this, Figure 5.10 shows this for the last state of *p4* (B,G,R,R) and the second state of *p3* (B,G,R). For the last state of *p4* (B,G,R,R), we create a **CompositeObjectState** (C2) that points to the **LeafObjectState** (L2) that represents the combination of one Green and one Blue token. In addition, we create a second **LeafObjectState** (L5) that defines two read tokens. Similarly, to record the second state of *p3* (B,G,R), we also create a **CompositeObjectState** (C1) that points to the **LeafObjectState** (L2) for the Green and Blue tokens. In addition, it refers to a **LiteralInteger** that defines one Red token.

In comparison to the original trace, shown in Figure 5.8, applying the compaction mechanism to this part of the trace leads to a decrease in the number of objects from 48 to 14 and the number of references from 60 to 35, around 71% reduction in the number of objects and 42% in the number of references.

5.3.2.4 Dealing with Repetitions in ParameterList

The last part of our compaction strategy deals with redundancies among input and output parameters of **Step** objects with regard to their values. It is very likely that the values of parameters be repeated among different **Step** objects during execution. Hence, our approach determines the

parameters that can be shared and represents them only once. The problem of the repetitions in parameter values and its respective solution for avoiding redundancy is similar to those that were given in Section 5.3.2.4 for **ObjectState**. Due to space restrictions, we only present the relevant part of CTM in this section.

As shown in Figure 5.11, the class **ParameterList** is decomposed into two subclasses: **CompositeParameterList** and **LeafParameterList**, each might have a reference to the class **ParameterValue**. We add the containment references `parameterlist`, `parametervalue` and `value` to the class **Trace** to enforce storing similar objects only once. Using such structure, we can obtain the sequence of **ParameterValue** objects relevant to a **ParameterList** object by traversing its corresponding **ParamererList**s in a recursive way. Finally, similar to **ObjectState**, the order of **ParameterValue** can be stored in the `parametervalueorder` attribute of the class **CompositeParameterList**, in the case that the order of **ParameterValue** changes after retrieving the **ParameterValue** sequence.

5.4 Related Work

In this section, we first give an overview of existing approaches for defining model execution trace structures, then we look at existing business process mining approaches and finally, we briefly describe efforts on existing scalable model persistence approaches.

5.4.1 Model execution tracing approaches

There exist many studies that tackle the problem of large traces with a focus on code-centric development. Existing approaches fall into different categories including trace filtering, graph reduction, trace visualization, partitioning, and trace abstraction [94]. A considerably less emphasis was placed on managing traces generated from executable models. We surveyed the approaches conducted in tracing executable models [159] and found that, although each approach has its own advantages, only a few have been proposed to deal with the scalability of traces in memory usage.

In the following, we present existing model execution tracing approaches that are related to our work.

Hegedus et al. [85] proposed a generic execution trace metamodel that can be specialized to any given xDSML. This approach reduces traces size by only capturing state modifications and events related to state modifications. Although such a technique is slightly similar to State compaction applied in CTM, their trace metamodel only considers event occurrences in an execution, whereas CTM presents a complete representation of traces (including states, steps, and their corresponding inputs and outputs) in a compact form.

Kemper and Tepper [101] proposed a scalable approach to remove repetitive fragments from traces using heuristic methods such as cycle reduction. Similar to our approach, the authors focused on removing repetitions contained in the trace. Contrary to CTM, their approach represents traces as simple sequences of events and states in the form of message sequence charts, and no trace metamodel is presented.

In the TopCased project, Combemale et al. [51, 40, 12] proposed an approach to define an execution trace metamodel for discrete-event system modeling. They manually provided a metamodel specific to an xDSML. Similar to the work of HegedÃijs et al. [85], this approach considers a trace only as a sequence of execution steps. Another limitation of this approach is that the obtained metamodel does not take into account any sort of compaction scheme. Hence, it is not scalable to support large traces.

Meyers et al. [41] applied a generative approach as part of their ProMoBox framework, which generates domain-specific trace metamodels for xDSMLs. The obtained trace metmamodel is clone-based, and defines execution traces as sequences of events and states in which each state is a complete snapshot of the executing model. Although the resulted trace is more rich than the aforementioned approaches, it does not consider any technique to compact execution traces.

Similarly, Gogolla et al. [144] generate so-called filmstrip models that can be considered as domain-specific trace metamodels. Such trace metamodels are also clone-based and capture operation calls as well as state modifications during the execution. Thereby, the trace is defined as a

sequence of events and states. However, the whole model is cloned to store each execution state, meaning that a complete snapshot of an object is created at each execution step, and all static fields (that never change) and dynamic fields (that may not change in each step) are stored. Consequently, there is a lot of redundancies due to repetition in states, so that the resulting traces, even for small models, might be very large, hindering scalability.

Aljamaan and Lethbridge [123, 124, 125] applied a different approach to enable model execution tracing. They proposed Umple[1]—an action language in a fully executable platform—for textual modeling with UML. They defined trace directives that allow modelers to specify traces of UML attributes and state machines, representing attribute values as states and transitions as steps. This approach represents a trace in a textual format without applying any compaction techniques, hindering scalability.

Fuentes et al. [136, 137] provided a dynamic model weaver for executing aspect-oriented models, leading to the generation of execution traces. A trace represents the execution of an activity as well as the current status of objects and their attribute values. Likewise, this approach suffers from the same limitations as the other approaches. Their approach deals with a specific kind of xDSML, as opposed to CTM, which is more generic. There is no compaction for execution traces either.

Mayerhofer et al. [39] proposed an approach to capture execution traces of fUML models. The traces represent the execution of fUML activities and actions, snapshots of object values, as well as processed inputs and produced outputs. Despite providing a complete representation of a trace for UML models, their proposed metamodel differs from CTM in that it is specific to fUML, while CTM is completely generic supporting any xDSML. Furthermore, it does not consider the compaction of traced data.

Hendriks et al. [117, 118] proposed a graph-based representation of execution traces for performing automated analysis techniques that can be applied by different modeling and analysis tools. The authors suggested the TRACE tool to interpret system behavior and analyze execution traces by using two different analysis techniques. They emphasize the fact that the large size of

[1] https://github.com/umple/Umple

traces complicates the interpretation and analysis behavior of systems over time. However, they do not apply any compaction techniques for traces. Moreover, their approach represents only trace execution events by producing traces that are sequences of events that occur during a model execution.

Schivo et al. [149] proposed UPPAAL as a back-end analysis tool for real-time systems. The approach provides a systematic way to define metamodels for UppaalâĂŹs timed automata, queries, and traces, which are needed to construct UPPAAL' models, and to verify relevant properties and interpret the results. Although the authors proposed a generic trace metamodel that is independent of any xDSML, the metamodel represents a trace as a sequence of states and transitions, and no compaction for execution traces is provided.

Recently, Bousse et al. [10, 7] presented a generative approach to automatically derive multidimensional domain-specific trace metamodels for xDSMLs that provide facilities for efficiently processing traces. Such metamodels define an execution trace as a sequence of execution steps and execution states where execution states capture the values of the dynamic properties of all model elements. By providing one navigation dimension per dynamic property, the trace metamodels improve scalability in trace processing time. With the derivation of domain-specific trace metamodels, usability is improved. The approach also considers reducing redundancies in states by only capturing the values of dynamic properties when they change between states. However, no other compaction technique is considered.

Finally, Luay and Hamou-Lhadj [162] proposed MTF (MPI Trace Format) to model in a scalable way execution traces generated from multi-process systems based on the MPI (Message Passing Interface) standard. The authors discussed the scalability problems of MPI traces and the lack of practical solutions. The design of MTF is validated against well-known requirements for a standard exchange format, with the objective to work towards standardizing the way MPI traces are represented in order to allow better synergy among tools. Although MTF was developed in the context of multi-process systems, the compaction techniques in MTF can be used to extend CTM to support traces of executable models, designed with concurrency and parallel processing in mind.

5.4.2 Business process mining approaches

Process mining techniques focus on extracting knowledge from event logs (i.e., execution traces). In many cases, the high volume of data is captured in specific event logs, causing problem in process discovery. Accordingly, significant effort has been made to analyze business process execution traces by developing efficient and scalable techniques in process mining. Examples of such techniques include event log filtering [163], event log transformation [164], trace clustering [165] and discovery techniques such as fuzzy mining [166] and pattern discovery [167, 168, 164]. Most of these techniques are somehow similar to the methods, which are used for code-centric trace compaction approaches.

For example, trace clustering is an effective way of dealing with large event logs by splitting into correlative subsets of traces. Song et al.[165] used a trace clustering technique to partition execution traces into different groups. By dividing traces into different groups, process discovery techniques can be applied on subsets of behavior and thus improve the accuracy and comprehensibility.

Bose and Aalst [164] proposed a multi-phase approach that provides abstractions of activities in traces based on the patterns. The approach first determines and replaces the repeated occurrence of the loop constructs in traces with an abstract entity. Then, it identifies and replaces sub-processes or common functionality with abstract entities. This technique is similar to the Step compaction technique of CTM, which identifies patterns of identical sequences of Steps and replaces them with an abstract entity (i.e., PatternOccurrence).

Pattern mining techniques such as the one's proposed by Tax et al. [169] discover patterns of local behavior from event logs. Such patterns do not capture the behavior of the complete traces. For instance, Tax et al. [169] discovered more precise process models by abstracting events at a higher level of human activity. Although these techniques simplify the analysis of event logs, their applicability depends on the availability of a list of human behavior at the activity level. Contrary to CTM compaction techniques, which allow us to keep track of every detail in the trace, pattern mining techniques ignore low-level data from a trace to understand its main content.

5.4.3 Model persistence approaches

The interest in scalable persistence of large models has grown significantly in recent years. In the following, we present some existing approaches in different categories.

5.4.3.1 XMI-based approaches

There exist many MDE approaches that used XMI as a common import/export model persistence format. Although XMI allows interoperability between existing tools and their models, it provides limited support for lazy or partial loading of models in memory for persistence. It also lacks scalability when working with large models. Examples are the work of Mayerhofer et al. [14], Combemale et al. [12], and Schivo et al. [149, 170] that persist execution traces in the XMI format. Besides XMI, CTM uses EXI format to address XMI limitations and improves scalability both in terms of memory and time required to store/load trace models.

5.4.3.2 Relational-based approaches

Another idea for persisting huge models is storing models in a relational database. An example is Connected Data Objects (CDO)[2] project in which an Ecore metamodel derives a relational schema and allows developers interacting with models. Such approach supports on-demand and partial loading of models in memory. While relational-based approaches are better than XMI serialization, they are no longer effective for timely, scalable data management. Furthermore, such approaches are still inefficient due to the highly interconnected nature of models for providing complex queries. As an example of a relational-based approach, Dominguez et al. [130] persist execution traces in a database. In this approach, an UML profile is generated for tracing system execution using a UML statechart. A persistence component transmits the runtime data obtained from the execution of a model to the trace database.

[2] http://www.eclipse.org/cdo/

5.4.3.3 Graph-based NoSQL databases

NoSQL databases provide better scalability and performance compared to relational databases [171]. MORSA [172] is the first approach for scalable model persistence based on a NoSQL back-end and on-demand loading/caching mechanisms. MORSA uses a document store database to persist large models using the standard EMF mechanisms. While this approach leads to an effective memory footprint and system performance, due to highly interconnected references between model elements, the storage of models can be extremely complex.

Hartmann et al. [173] proposed a compact representation of time-evolving graphs for analyzing complex data. The authors incorporated time as a first-class property into a temporal graph structure to make each node an independent time series. A temporal graph is an efficient data structure for storing the history of data that frequently changes over time. In this approach, GREYCAT, a framework for time-evolving graphs, was implemented to support read and write mechanisms for a temporal graph. This structure provides substantial memory reduction rather than snapshots. It is efficient for analyzing large-scale graphs especially with partial changes along time. This approach is similar to CTM as both of them focuses on state changes instead of providing a complete or partial snapshot of data.

Another example is KMF runtime versioning [5], which stores the versions of each object of a model separately, allowing to enumerate the states of a specific object of the executed model. It efficiently supports the notion of model versioning by setting a specific version to each object or making a reference to a particular version of an object. This approach considers changes at the object level, allowing to browse an execution trace by navigating among the states of a model. The model elements can be stored from memory to a NoSQL database. Similar to this approach, we only save incremental changes rather than snapshots of a complete model. While this approach offers a considerable reduction in memory usage for small modifications, it induces a serious overhead for full model change storage.

5.5 Conclusion

Dynamic V&V of models requires the ability to capture execution traces for the execution of models. In order to support dynamic V&V, an efficient representation of trace information is required. We identified three main requirements for designing a trace metamodel: *genericity*, *scalability in space*, and *information preservation*. To design a trace metamodel for an xDSML, the following requirements have to be addressed. First, it must be generic to support any kind of xDSML. Second, it must provide good scalability in space. In addition, it should provide lossless representation of traces. In this thesis, we presented CTM, a metamodel for representing traces generated from executable models. The metamodel captures sequences of model states, execution steps, object values, and parameters, which are concepts that exist in most xDSMLs, making CTM generic enough to support traces generated from models of various xDSMLs. We designed CTM by embedding various compaction techniques that remove redundancies in model states, object states, values, steps, and parameters.

Part III

Applications and Tooling

Chapter 6

Tool Support in the Context of Gemoc

In this chapter, we present the software development that was achieved to implement our approaches and applications. Section 6.1 gives an overview of the Gemoc studio and its execution framework. Continuing, we present all the work that was performed in the Gemoc Studio. Section 6.2 explains the implementation of our trace metamodel that we presented in Chapter 5. Continuing, Section 6.3 presents techniques and algorithms that are used for the compaction of traces in CTM. Lastly, Section 6.4 evaluates CTM by defining several research questions, measuring evaluation criteria, and comparing the results with the existing trace metamodels.

6.1 Gemoc Studio Execution Framework

The Eclipse Gemoc Studio[1] is a framework for designing and integrating, and using different executable modeling languages. The framework provides generic components through Eclipse technologies to define discrete-event operational semantics into an execution engine specific to a metaprogramming approach. Figure 6.1 gives an overview of the Gemoc Studio.

The Gemoc Studio offers two workbenches including a language workbench and a modeling workbench. The language workbench is used by language designers and language integrator to build and compose new executable modeling languages. For this work, the designer should define the abstract syntax (using Ecore), the operational semantics, and the concrete syntax of an xDSML. Figure 6.2 shows a snapshot of the Gemoc language workbench that has been used for designing a Timed Finite State Machine (TFSM) example.

[1] http://gemoc.org/studio

The modeling workbench is used by domain designers to create, execute, and coordinate models conforming to executable modeling languages. An advanced execution engine also exists that can be used for the execution of any model.

Figure 6.3 presents an overview of the underlying architecture of the execution framework. On the left, an xDSML containing an abstract syntax, a concrete syntax and operational semantics is shown. At the middle, the model that is conformed to the execution metamodel of the xDSML is presented. The state of the model is modified by the transformation rules of the operational semantics. In addition, it provides an interface to define engine addons that contain components to control the progress of the execution, and can be used to support the execution. For example, an add-on may be used to provide synchronous notifications to control the execution. It is also possible to create queries, and to modify the model between execution steps (e.g., for implementing a debugger).

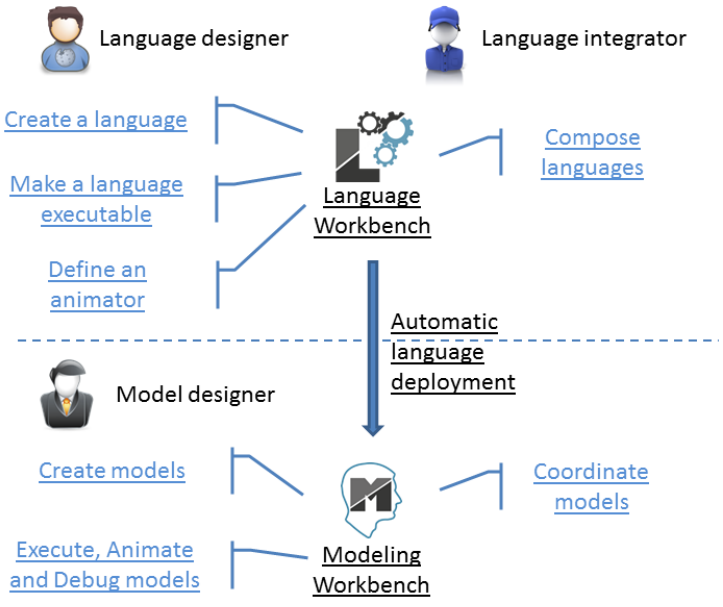

Figure 6.1: overview of the Gemoc studio [174]

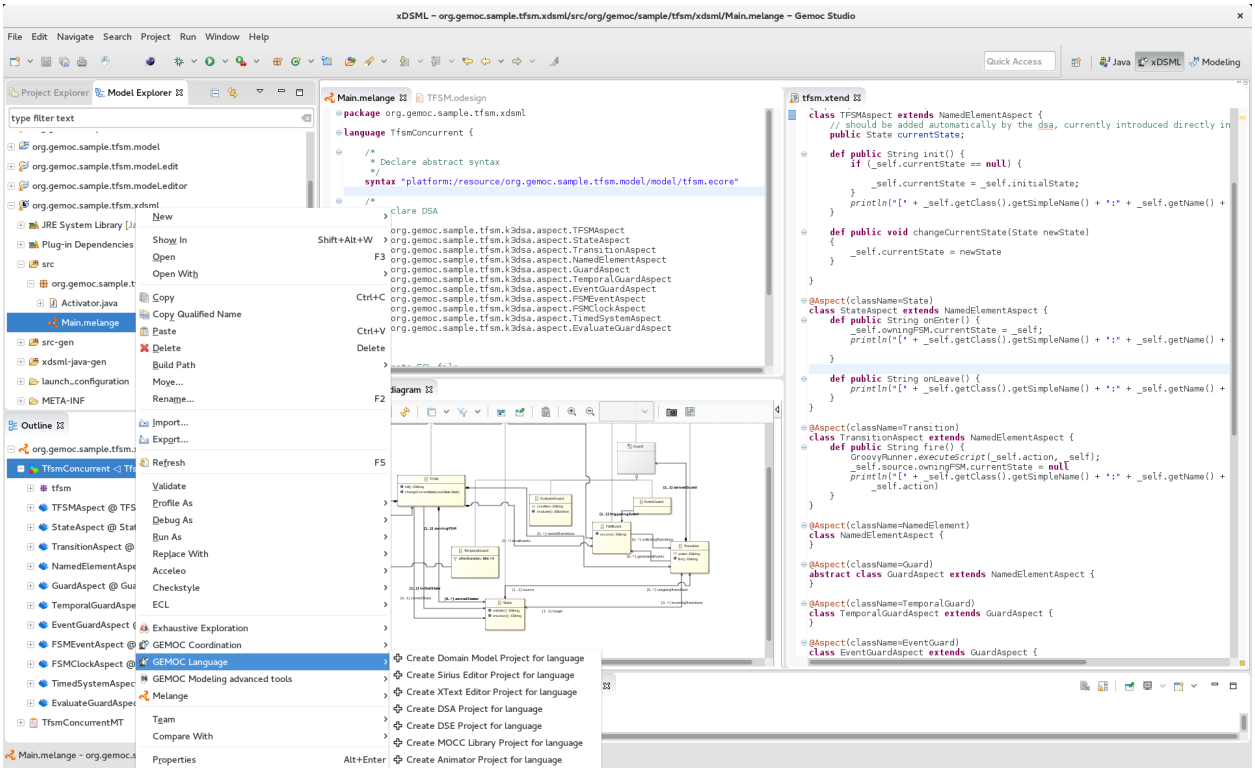

Figure 6.2: Screenshot of Gemoc language workbench showing the design of a TFSM example

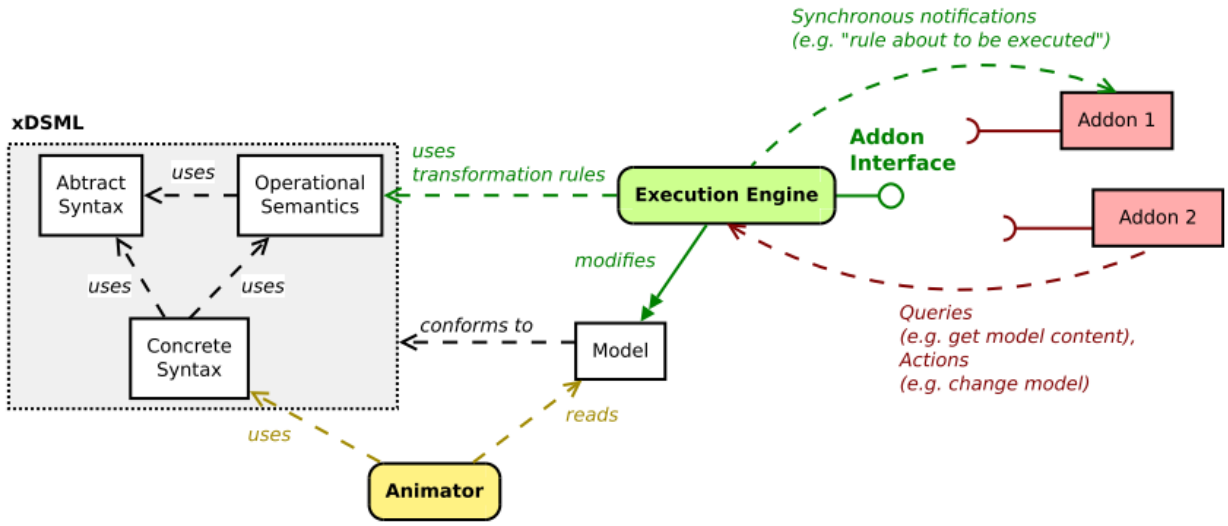

Figure 6.3: Overview of the Gemoc modeling workbench execution framework [174]

6.2 Implementation of CTM

We have implemented a prototype in EMF using the Xtend[2] and Java programming language. It was implemented as an add-on integrated in Eclipse Gemoc Studio. The prototype is a set of Eclipse plug-ins containing the generic and compact trace metamodels, and a *trace constructor* as well as a *trace decompactor*. We explain and discuss the different parts of the implementation in the following sections.

6.2.1 Generation of proposed trace metamodels in EMF

We employed the metamodeling language Ecore for defining the abstract syntax of the metamodels, and Kermeta and xMOF for the operational semantics. For each metamodel, we have generated three additional plug-ins which provide wizards for creating new model instances, an editor which allows to enter the model information, and templates to write tests for a model as well.

At the beginning, we created a project to define two trace metamodels (generic and CTM) by using EMF tooling, creating Ecore models (.ecore files), and an EMF generator model (.genmodel file) that configures the generation of the Java API corresponding to the metamodel. The EMF model code generator is called with the .genmodel file, and generates Java interfaces and Java classes. The genmodel defines various options needed for the code generation, e.g., the path and file information. The genmodel file also contains the control parameter how the code should be generated.

The generated code consist of three parts: model, model.impl, model.util. Every generated method is tagged with @generated. Although in some cases, we had to adjust the methods, and changed them to support special features. For instance, for the non-unique references in CTM, we added an additional code to the corresponding methods to support duplicated objects. Note that due to a bug in EMF, assigning *false* value to the flag **Unique** for a reference doesn't support

[2]https://www.eclipse.org/xtend/

duplicated objects. Listing 6.2.1 shows the updated version of the generated Java code of the method getParametervalue in ParameterListImpl class.

```
1    *  @generated Not FIXME workaround BUG 89325
2    *  In order to accept duplicates in parametervalues
3   @SuppressWarnings("supporting duplicats in object")
4   @Override
5   public EList<ParameterValue> getParametervalue() {
6     if (parametervalue == null) {
7       parametervalue = new EObjectResolvingEList<ParameterValue>(
     ParameterValue.class, this,       TracePackage.
     PARAMETER_LIST__PARAMETERVALUE){
8             @Override
9                 protected boolean isUnique() {
10                    return false;
11                }
12               };
13         }
14     return parametervalue;
15  }
```

Listing 6.2.1: The updated code of *getParametervalue* method for **ParameterListImpl** class to support duplicates in `parametervalue` reference, written in Java

The code was added for the following references in the respective methods.

- Patternoccurencestepdate.`states`
- Patternoccurencestepdate.`parameterlists`
- Objectstate.`values`
- Refvalue.`originalobjects`
- Parametervalue.`values`
- Steppattern.`repeatingstep`
- Compositeparameterlist.`parameterlist`
- Compositobjectstate.`objectstates`

Moreover, we relied on Kermeta aspects to add new properties and operations, and implemented the operational semantics, weaving them into the proposed metamodels. Kermeta is a language designed as an extension of the Xtend language, which is part of the Eclipse project.

Xtend is a programming language that provides same possibilities as Java, with a very powerful syntax, better switch expressions and template expressions as well. Xtend is extensible through so-called *active annotation*, which is a mechanism to allow developers to change the content of methods. Kermeta consists of a set of Xtend active annotations. **@Aspect** is an annotation that is used to add new properties and methods in them.

6.2.2 Creation of an xDSML

As shown in Figure 5.1, the input of our approach is an xDSML, which is defined using Ecore for the abstract syntax, and using either Kermeta [175] or xMOF [14] for the operational semantics. At the next step, we created the abstract syntax of the Petri net language, i.e., the ecore metamodel, also called "domain model" in Gemoc, and generated the code of the ecore metamodel using a genmodel file. Then, we created the operational semantics of the language, which is also called "DSA project" in Gemoc. Using Kermeta 3, we created an Xtend file containing a set of aspects, each being an extension of a class of the abstract syntax (e.g., Place or Transition). Each aspect allows to define (a) new dynamic properties, such as the number of tokens contained in a Place, and (b) methods, which are the transformation rules that will create execution steps, such as "fireTransition". The last step for the definition of an xDSML is creating the language definition project, which declares all the different components of the language using a DSL called Melange. In summary, for an xDSMl we have to create three projects:

- the abstract syntax (with the ecore)
- the operational semantics (with xtend)
- the language definition (with melange)

We validated our approach by all well known xDSMLs. For each xDSML, we created the three aforementioned projects for making the respective language executable.

6.2.3 Implementation of the *Trace Constructor*

The main element of our prototype is the *Trace Constructor* (shown by *d* and *e* elements of Figure 5.1), which is used for constructing traces during a model execution. It contains two components: 1) *regular trace constructor*, which constructs traces in regular form conforming to the generic trace metamodel. 2) *compact trace constructor* which has been developed so that each part of the trace (State, Step, ObjectState, ParameterList, Value) can be produced in both regular (*uncompact*) and compact form. Both of two trace constructors, as shown in Figure 5.1, use an xDSML and an input model for the execution as well. The output of the model execution is an execution trace either in regular or compact form. A user can choose the trace compaction techniques (i.e., State, Step, ObjectState, ParameterList) he or she wishes to apply by selecting the corresponding flag in the compact trace constructor tool. Thus, the trace can be partially or entirely compacted. In the case that all flags are false, the trace is constructed in a regular form using the *regular trace constructor*. In the case of generating the compact trace, we have applied several techniques, each being used to efficiently reduce the size of the corresponding part of the trace described in Section 5.3.2. We will detail each technique in the following sections.

6.2.4 Implementation of the *Trace Decompactor*

The second part of our prototype is a *trace decompactor*, which aims to prove that the trace constructor a lossless compact execution trace. The *trace decompactor* takes a serialized compact trace as an input, and produces the original one. As mentioned before, the *regular trace constructor* generates trace without compaction. Hence, we are able to compare such trace with the regular trace resulted from the *trace decompactor*. Note that similar to the trace construction process, the *trace de-compactor* relies on four boolean flags, each specifying the de-compaction state of each part of the trace (State, Step, ObjectState, ParameterList). Therefore, trace de-compaction can be done partially if there is no need for full de-compaction of the trace.

CTM was implemented as an engine add-on deployed in Eclipse Gemoc Studio. This simplifies the integration of the trace constructor with the execution engine, as the engine is responsible

for running the execution transformation, and no modifications of the execution transformation are required to enable construction of traces. It is worth noting that the CTM components (i.e., generic trace metamodel, compact trace metamodel, *trace constructor* and *trace de-comactor*) were implemented independently from any xDSML, and can be applied to any execution framework that supports execution of models. In this case, the considered xDSML containing its abstract syntax and its operational semantics should be supported by the execution framework.

Our prototype has been tested on a selection of xDSMLs that had been previously developed using the Gemoc Studio. Table 6.2 presents all the xDSMLs considered so far, with links to their source material and their semantics as well. The trace constructor and trace decompactor were successfully tested for these languages.

Both the *trace constructor* and the *trace de-compactor* have been implemented using the Xtend[3] and Java programming language. The *trace constructor* and the *trace de-compactor* share some part of the code and comprise 4107 and 744 lines of Java and Xtend codes, respectively. The source code (EPL 1.0 licensed) is available at our project web page[4].

6.3 Applying Compaction Techniques to CTM

In the following sub sections, we explain how compaction techniques are employed, and applied to CTM to produce a compact execution trace.

6.3.1 Implementation of Step Compaction

In this section, we present an algorithm for converting a tree structure into an ordered directed acyclic graph. We used an extension of Valiente's algorithm [176] to detect Step repetitions in a trace when the trace is represented in a tree. In the following, we first define Valiente's algorithm, then explain how it can be improved to abstract the Step part of the trace.

[3]https://www.eclipse.org/xtend/
[4]https://github.com/MDSEGroup/TraceCompaction

6.3.1.1 Overview

Valiente [176] presented an algorithm that traverses the tree in a bottom-up fashion (from the leaves to the root). In this algorithm, a *certificate* (positive integers between 1 and the size of the tree) are assigned to nodes so that the roots of two isomorphic sub-trees take the same *certificate*. For computing each *certificate*, the algorithm uses *signature*, which is obtained by concatenating the label node and the *certificates* of its direct children.

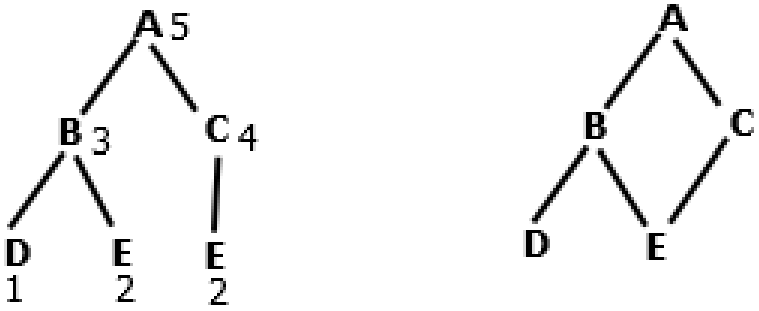

Figure 6.4: A sample tree (left) and its DAG (right) which is transformed using Valiente's algorithm

For example, in Figure 6.4, the signature of B is "B 1 2". For the leaf nodes, the signature is their labels (e.g. the signature of D is "D"). In the program, a hash table (or hash map) can be used to store the certificates and signatures, so that there is no duplication for the certificates. The results of applying Valiente's algorithm to the tree are shown in Table 6.1

Table 6.1: Result of applying Valiente's algorithm to the tree of Figure 6.4

Signature	Certificate
D	1
E	2
B12	3
C2	4
A34	5

Time complexity of the algorithm consists of the time of traversing the tree, the time needed to compare two sub-trees, and the time of calculating the signatures.

6.3.1.2 Extension of Valiente's Algorithm to Trace Compaction

To deal with the repetitions of Steps, we used an extension of Valiente's algorithm [176] proposed by Hamou-Lhadj and Lethbridge [64] to detect redundant patterns within execution steps. In our case, the execution trace would be a tree, containing the nodes corresponding to the Steps. The aim is to detect patterns involving contiguous repetitions of Steps existing due to loops or recursion. As mentioned in Section 5.3.2.2, the StepPattern class in the metamodel represents the repetition in steps. The idea is to be able to detect similar patterns, which can be either sub-trees or leaf nodes during the generation of the trace. At the end of execution, the node signatures are computed by detecting the patterns, and using the global hash table to find/add the respective entry. The final table represents the DAG version of the trace.

To develop the algorithm, let us consider the following items:

- We need to create a new structure as *nodesignature* for the nodes, including a string value as the signature, and a boolean value to show whether the node involved in a pattern or not. Each entry of the global hash table contains a *certificate* and a *nodesignature*. The signature composes of the **StepType** of the respective **Step**, and the *certificates* of its direct children.

- A node or a sub-tree is considered as a pattern only when they exist in a loop or recursion. Hence, the certificates of the same nodes are different, if they occur only once.

- The algorithm is done *offline* (i.e., after the execution of the model), due to the complexity of applying the technique during execution. We defer doing the step compaction *on the fly* to future work.

The algorithm simply takes a complete execution trace, and produces a *certificate* and *signature* for each node by traversing the tree of Steps in a bottom-up fashion (from the leaves to the root). The *certificates* are assigned to nodes so that the roots of two isomorphic sub-trees take

the same *certificate*. The signatures are computed by checking the global table, and finding the respective values if they exist. To carry out this work, we extended the **Step** class by adding two new properties (*signature*, *certificate*) in the trace metamodel through a Kermeta aspect (shown in Listing 6.3.1). Given a *signature*, we can recognize repetitions that might be included in the corresponding **Step**, and thus construct the trace in accordance to the trace metamodel described in Section 5.3.2.2. Algorithm 1 shows our compute signature procedure. It is a recursive procedure that is called in line 7.

```
1  @Aspect(className=StepSpec)
2  abstract class StepSpecAspect
3{
4     public String signature=""
5     public int certificate=0
6}
```

Listing 6.3.1: Defining two new properties in StepSpec aspect class Aspect

According to the algorithm (lines 8, 11), the node signature is a string value, which can contain contiguous repetitions of numbers, meaning that similar certificates occur contiguously. We used Java.Utill.Regex class and its methods (e.g., compile, matcher) to find patterns in a signature.

6.3.2 Implementation of State Compaction

As for the State part, we used an add-on named BatchModelChangeListener appended in Gemoc Studio to track the changes that are made to the objects during an execution. It can recognize new objects created, and the objects removed after execution of each execution step. In addition, the changes made on the collection fields and non collection fields are provided as well. This method helped us to represent only the modification between **State** objects as **TransientObjectState** objects. The value of `basestate` reference is determined by traversing the trace, comparing the states with the current state and finding the closest one. If there is no change in **Objectstate** values, no new **State** is created, and previous **Step** is assigned to the current **Step**.

Listing 6.3.1 shows how the BatchModelChangeListener method gets the changes made on the objects. This method takes an engine with type of IExecutionEngine as an input (line 1), and puts the model changes on the related output variables. It first defines a listener, assigns related resource model, and gets the list of changes by executing *getchanges* method (lines 6 -9). Then, the *for* statement iterates over the list of values, each being compared with the values for each case in the switch structure. If there is a match, the block of code associated with that case is executed. For the new and deleted objects, only the objects are added to the associated variables (lines 11 - 15, 28 - 30). For the changes made on a field (either collection or non collection), the respective feature is also added in the associated variables. Note that `ObjectFeature` (lines (17, 23)) is a new structure defined with two elements including Eobject and EStructuralFeature, which is used to store both objects and features of the model changes.

In addition to considering state modifications after executing steps, our trace constructor supports also recording of state modification that occur before starting a step. This is relevant when a step makes a change before calling another (sub-)step. In such case, the *trace constructor* creates an instance of **Step** object and its **StepType** is assigned to *"Implicit step"*.

Algorithm 1 Algorithm for calculating signature and certificate for each step in the execution trace

▷ %input: trace , step % ▷ % output: an integer value as certificate of step%

1: i=0, fnode=null
2: nodeSignature=”, subtreeSignature=”

▷ % nodeSignature is the signature of the node respective to step, subtreeSignature is obtained by concatinating the signature of all the children of the step %
3: globalTable=a Hash table
▷ % In the globalTable, the keys represent signatures and the values represent certificates%
4: **if** (step.children not null) **then**
5: **while** (i<step.children.size) **do**
6: fnode= step.children.get(i)
▷ %the algorithm is called in recursive way to compute certificate of the input node%
7: certificate= call the algorithm with fnode as an input
8: $subtreeSignature|$certificate
▷ % || : concatination sign %
9: i++
10: **end while**
11: $nodeSignature\|$node.lable
12: $nodeSignature\|$subtreeSignature
▷ % node label is step.StepType%

▷ % The findglobalTable method check globalTable, if there exist an entry with nodeSignature, the method return the respective certificate, if not, the certificate is increased by one, and an entry (certificate,nodeSignature) is added to globalTable %
13: certificate=findglobalTable(nodeSignature)
14: assign cerificate to step, assign nodeSignature to step
15: return certificate
16: **end if**
17: **if** (step children is null) **then**
18: $nodeSignature\|$node.lable
19: certificate=findglobalTable(nodeSignature)
20: assign cerificate to step, assign nodeSignature to step
21: return certificate
22: **end if**

```
def void preparechangelist(IExecutionEngine engine, Collection<EObject> newObjects,
  Collection<ObjectFeature> NonCollectionFieldobjects, Collection<ObjectFeature> PotentialCollectionFieldobjects,
  Collection<EObject> removedObjects) {
  // Defining a Listener and
  this.listenerAddon = new BatchModelChangeListener(
    EMFResource.getRelatedResources(engine.executionContext.resourceModel))
  listenerAddon.registerObserver(this)
  var List<ModelChange> changelist = listenerAddon.getChanges(this)
  // A ModelChange can be a new/removed object in the model, or a change in a field.
  for (c : changelist) {
    switch (c) {
          NewObjectModelChange: {
        newObjects.add(c.changedObject)
      }
      NonCollectionFieldModelChange: {
        val ObjectFeature of = new ObjectFeature()
        of.object = c.changedObject
        of.feature = c.changedField
        NonCollectionFieldobjects.add(of)
      }
      PotentialCollectionFieldModelChange: {
        val ObjectFeature of = new ObjectFeature()
        of.object = c.changedObject
        of.feature = c.changedField
        PotentialCollectionFieldobjects.add(of)
      }
      RemovedObjectModelChange: {
        removedObjects.add(c.changedObject)
      }
    }
  }
}
```

Listing 6.3.1: Using **BatchModelChangeListener** to get model changes

6.3.3 Implementation of Objectstate Compaction

Regarding the ObjectState compaction, the main challenge was how to efficiently identify the similar values of ObjectStates that can be shared among different ObjectStates. To this work, we used LCM (Linear time Closed item set Miner) [177], a powerful algorithm for enumerating frequent closed item sets, which creates a set of ObjectStates including the values that occur more frequently than a certain threshold. We chose this algorithm due to efficiency in memory saving and computation time. We defined the threshold value (between 0 and 1) as minimum support by executing several fUML models multiple times, and selected the value that leads to less memory

Table 6.2: xDSMLs applied to test our prototype

xDSML	Link	Description	Semantics
Petri nets	link[a]	Simple Petri nets (see Figure 2.1)	xMOF
PetrinetComplex	link[b]	Petri nets with token objects	Kermeta
IML	link[c]	AutomationML Intermediate	Kermeta
TFSM	link[d]	Time finite state machine	Kermeta
fUML	link[e]	Complete fUML	xMOF

[a] https://github.com/MDSEGroup/TraceCompaction/tree/master/Traceconstruction/Petrinet
[b] https://github.com/MDSEGroup/TraceCompaction/tree/master/Traceconstruction/PetrinetComplex
[c] https://github.com/MDSEGroup/TraceCompaction/tree/master/Traceconstruction/IML
[d] https://github.com/MDSEGroup/TraceCompaction/tree/master/Traceconstruction/TFSM
[e] https://github.com/MDSEGroup/TraceCompaction/tree/master/Traceconstruction/fUML

consumption during the execution.

6.3.4 Implementation of Parametervalue Compaction

As the compaction technique used for Parametervalue is the same as Objectstate, we used the same algorithm for finding frequent ParameterValues within ParameterList objects, and consequently creating **CompositParameterList** and **LeafParameterList** representing the *Composite design pattern* [160]. Note that in both cases, the compaction can be done both *on the fly* and *offline*.

6.4 Evaluation of CTM

In this section, we present the evaluation of our approach. We first provide background information on fUML, then we evaluate the genericity of CTM with respect to different xDSMLs and different metaprogramming approaches. Thereafter, we present the conducted experiments and a set of metrics regarding to the scalability of traces created with our approach. We have also evaluated the overhead caused by CTM regarding execution time and memory consumption. Continuing, we present the evaluation of CTM with regard to information preservation. Finally, we discuss the evaluation results.

6.4.1 Overview on fUML

Foundational UML (fUML) [178] is an OMG standard, which defines the execution semantics of a subset of UML through an operational approach. It provides a virtual machine for executing fUML-compliant models. The fUML subset contains parts of the abstract syntax of UML including structural concepts for defining UML classes and behavioral concepts for defining the behavior of these classes using UML activities. The fUML execution model is a model which defines the execution semantics of the fUML subset, and specifies how fUML models are executed. fUML basically enables the execution of UML activities. For the execution, the fUML virtual machine takes an fUML activity and the activity's input parameter values as input, and produce values for the activity's output parameters. The execution semantics of fUML activities is similar to the one of the Petri net xDSML; both are based on offering and consuming tokens, except that tokens in an fUML activity, can specify either control or data. Control tokens define the beginning and the end of an activity, as well as conditionals or concurrency among nodes. Object tokens represent the passing of data between actions. In some cases (i.e., join node) both control and object tokens may flow among actions.

6.4.2 Experiments on CTM

We evaluated CTM with respect to several research questions, which have been defined based on all targeted criteria.

6.4.2.1 Genericity

To evaluate the genericity of the CTM, we considered the following research questions.

RQ #1: Can CTM be used with different xDSMLs?

RQ #2: Can CTM be used with xDSMLs implemented using different metaprogramming approaches?

To answer these questions, we tested CTM with a selection of different xDSMLs that had previously been developed using Gemoc Studio. Table 6.2 presents all the xDSMLs considered in this study, with links to their source material. The trace constructor and trace decompactor were successfully tested for these languages. For each xDSML, we executed several example models with different parameters, and generated execution traces using trace constructor.

For each xDSML, we implemented a set of plugins containing the abstract syntax (implemented with the Ecore), and the execution semantics (implemented using either Kermeta or xMOF). Then, we executed several example models with given parameters, and created traces (in both regular and compact forms) with the *trace constructor*. The regular traces were compared with traces that were produced from the domain-specific trace metamodels proposed by Bousse et al. [7], for number of **Step**, **State** and **Value** objects, and observed the same results. We also reconstructed uncompacted version of the compact traces using the *trace decompactor*, compared them with the uncompacted ones produced by the regular trace constructor, and observed similar results.

6.4.2.2 Scalability

To evaluate scalability of CTM, we compared the memory and disk space used by the trace generated by CTM with the trace obtained from the domain-specific trace metamodels proposed by Bousse et al. [7]. We chose to compare our approach to Bousse's method because, to our knowledge, this is the only method that supports compaction of traces of executable models.

To proceed, first, we chose the set of 16 fUML models, which have been selected by Maoz et al. [61] from different industrial sources (e.g., IBM, Nokia), and have been already used for similar case studies (e.g., [122], [45], [7]). We chose these fUML models because they have also been used by Bousse et al. [7] for the generation of traces. Note that the experiments contain a total of 38 model execution of the considered fUML models, with the number of execution states ranging between 180 and 340, and different parameter setting. We examined the following research question to evaluate the scalability of CTM.

RQ #3: Does CTM reduce the size of traces in memory and disk space, as compared to

existing trace metamodels?

For answering RQ #3, we have first defined a set of simple and practical metrics that aim to measure how scalable the trace is compared to the other existing tracing approach. We then show the results of applying these metrics to several traces generated for fUML models. The first metric is used for the disk space measurement, and the two other ones measure the memory used by the trace at the conceptual level.

File size [S] is the size of a trace serialized in the XMI standard format. It specifies how much storage space we need to store a trace.

Number of Objects [Nobj] is the number of objects used to represent the trace. It is important to notice that in practice the number of objects is equal to the number of nodes within the trace, specified as either a graph or a digraph object.

Number of References [Nref] is the number of references in the trace. This number is equal to the number of edges in the graph made from the trace.

We define the memory size, A, of the CTM trace as the total number of objects and references in the trace:

$$A = Nref_{CTM} + Nobj_{CTM} \tag{6.1}$$

Similarly, we define the memory size, B, of the obtained trace from the domain-specific trace metamodels as the total number of objects and references in the trace. Note that we are using the subscript DS to mean 'Domain-Specific trace'.

$$B = Nref_{DS} + Nobj_{DS} \tag{6.2}$$

We measure the memory compaction rate as follows:

$$CompactionRate = (1 - A/B) * 100\% \tag{6.3}$$

Besides, we measure the disk usage compaction rate as follows:

$$CompactionRate_{disk} = (1 - A/B) * 100\% \tag{6.4}$$

where A and B refers to the disk usage of the CTM trace and the domain specific trace, respectively.

Using the aforementioned metrics, we have measured the scalability of the execution traces constructed with our CTM add-on, in terms of memory and disk space.

Likewise, the compaction rate corresponding to each part of the trace (i.e., State, Step, Object-State, and ParameterList) can be determined using the same formula. For instance, we measure the compaction rate for State using the memory compaction rate formula, except that A and B now refer to the total number of objects and references in the trace with the State-based compaction technique and without it, respectively.

We used yEd Graph Editor [5] (Version 3.17.1) (for the very large trace, we used Gephi [6] (Version 0.9.1)) and Advanced XML Converter [7] (Version 3.02.0.12) to generate graphs of the serialized traces, and prepared several SQL scripts running in SQL Server 2016 to determine the generated graph's nodes and edges.

6.4.2.3 Information Preservation

We define the following research question to demonstrate information preservation of CTM.

RQ #4: Can CTM provide a lossless representation of traces?

For answering RQ #4, we used the *trace decompactor* to reconstruct an uncompacted version of a compact trace. This uncompacted version was then compared with the trace produced by the regular trace constructor (i.e., an uncompacted trace). Both traces were serialized as XMI files, and compared using EMF Compare.

[5] https://www.yworks.com/products/yed
[6] https://gephi.org/
[7] www.xml-converter.com

6.4.2.4 Performance overhead

We evaluate the performance of CTM by considering the following research question.

RQ #5: How much performance overhead is caused by CTM?

To answer RQ #5, we measured the runtime overhead induced by the trace construction using CTM, and compared it with the execution time needed to construct traces of domain-specific trace metamodels. The runtime overhead is obtained by comparing each execution time with the time needed for the model execution where no trace was constructed.

The experiments for answering the research questions were performed on the following hardware and software environment.

- **Hardware**: Intel Core i7-2620M CPU 2.5 GHz, 12 GB RAM
- **Operating system**: Windows 10 Professional 64-Bit
- **Eclipse Gemoc Studio**: Eclipse Oxygen 3, Build 2018-07-17
- **Java**: Version 8, Build 1.8.0_60
- **Eclipse Memory Analyzer**: Released Version 1.8.1

6.4.3 Results of the Evaluation

In the following, we present the results obtained from the experiments, and give the answers to the research questions.

RQ #1 and RQ #2: Genericity of CTM. The results obtained regarding the genericity of CTM shows that the compact trace can be generated for any given xDSML and regardless of the metaprogramming approach used for the implementation of execution semantics. Therefore, to answer RQ #1 and RQ #2, we observe that CTM is generic enough to support different xDSMLs, and different metaprogramming approaches.

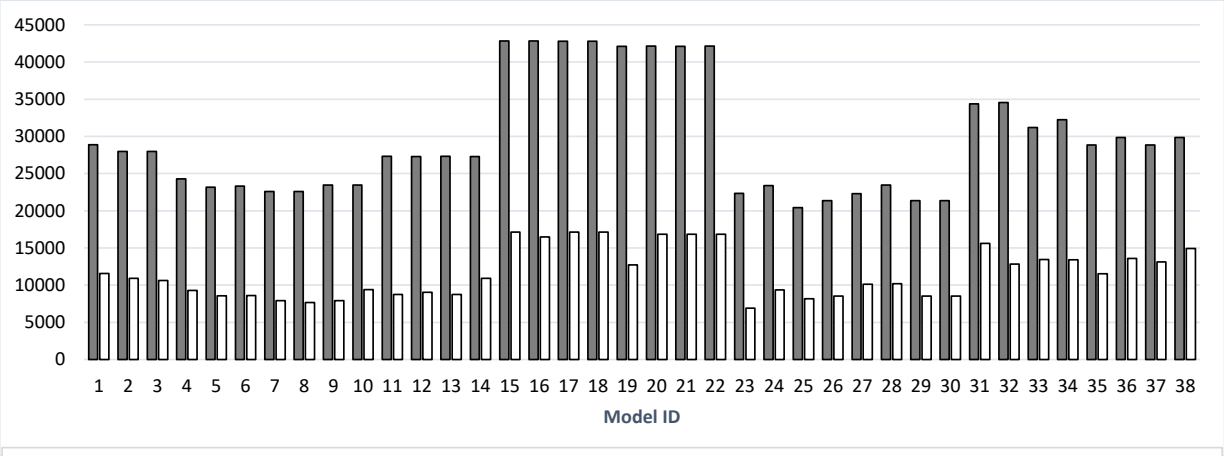

Figure 6.5: Number of objects used by both CTM traces and domain-specific traces

RQ #3: Scalability. Figure 6.5 shows the number of objects used to represent the trace with CTM and the domain-specific trace metamodels generated with [7]. The X-axis shows the used example model, while the Y-axis shows the number of objects contained in the trace recorded for the model's execution. Likewise, Figure 6.6 shows the number of references used to represent the trace obtained by CTM and the domain-specific trace metamodels. These two measures are related to memory consumption. As we can see, domain-specific traces require 1.7 to 2.2 times more objects than CTM traces with an average of 1.9. In addition, we observe that less references are created using CTM compared to domain-specific traces. As shown in Figure 6.6, domain-specific traces require 2.1 to 3.3 times more references than CTM traces with an average of 2.5.

Furthermore, regarding the disk usage, Figure 6.7 shows the disk space usage of the execution traces with CTM serialized in both XML and EXI, and the domain-specific trace metamodels. The Y-axis shows the amount of disk used by the trace in kilobytes (kB). We observe a significant reduction of the disk usage ranging from 92% to 96% for the CTM trace serialized in EXI, and 65% to 73% for the CTM trace serialized in XML. This means that domain-specific traces require 13.4 to 28.5 times more disk usage than EXI-based CTM traces with an average of 18.2 and needed 2.8 to 3.7 times more disk usage than XML-based CTM traces with an average of 3.3.

Therefore, to answer RQ #3, we observe that CTM traces are more efficient in both memory

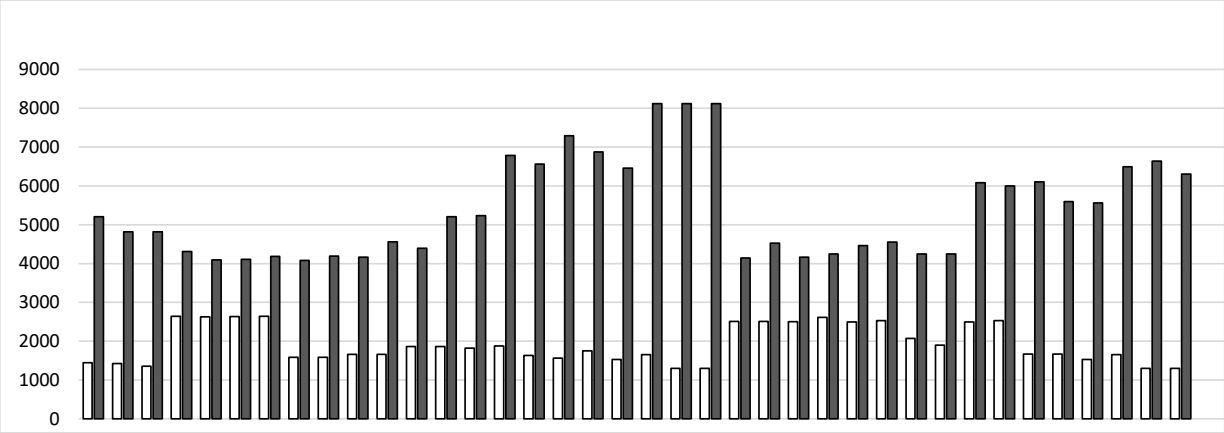

Figure 6.6: Number of references used by both CTM traces and domain-specific traces

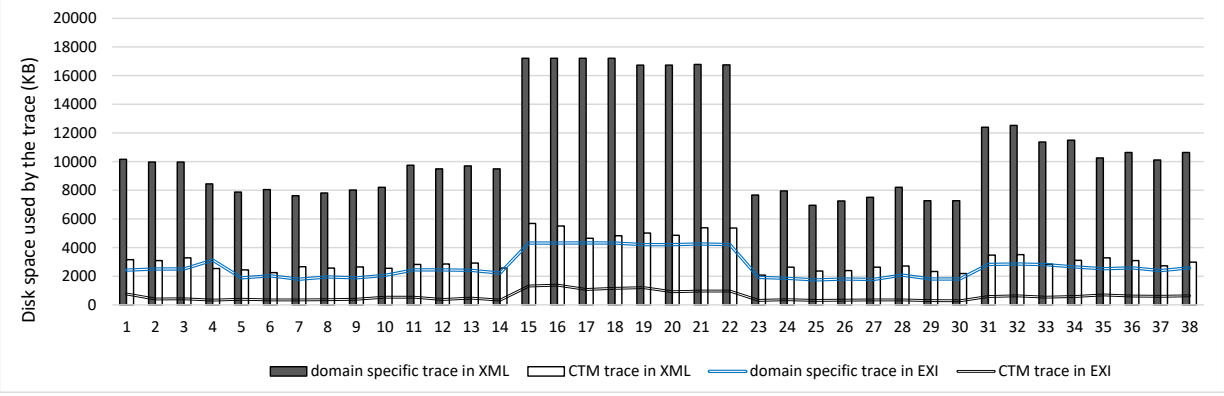

Figure 6.7: Disk space used by both CTM traces and domain-specific traces

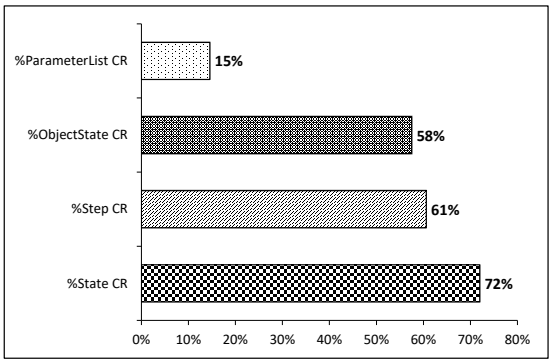

Figure 6.8: Compaction rate of CTM trace elements

and disk usage than traces obtained by domain-specific trace metamodels as defined in [7]. In Summary, CTM achieves an average compaction rate of 59% in memory usage and 95% in disk space for EXI-based CTM traces.

Regarding the memory compaction rate, corresponding to each part of the trace, we also did an empirical study on ten selected fUML models. First, we measured the memory compaction in terms of the number of objects and references, which are relevant to each of trace element instances (i.e., State, Step, ObjectState, ParameterLists). Then, we measured the average of the compaction rates of the selected models for every trace element separately. Figure 6.8 shows a chart depicting the distribution of the memory compaction measurements corresponding to each trace element. The figure shows that the State element obtained the highest ratio of memory gain (%35) over the total compaction rate, while the ParameterList element has the least (%7). These results are due to the fact that the State of a trace contains the states of all objects in the executed model after each execution step that occurs. Hence, storing only states modification using State compaction technique can provide a remarkable result. The State compaction rate heavily depends on the number of static or dynamic objects and the number of execution steps as well. Although the compaction technique used for ParameterList is similar to the technique used for ObjectState, in most cases such as what we had in the selected fUML models, the redundancies among input and output parameters are less than the value repetitions in ObjectState objects. Therefore, in our experiments, ObjectState obtained more memory gain than ParameterList.

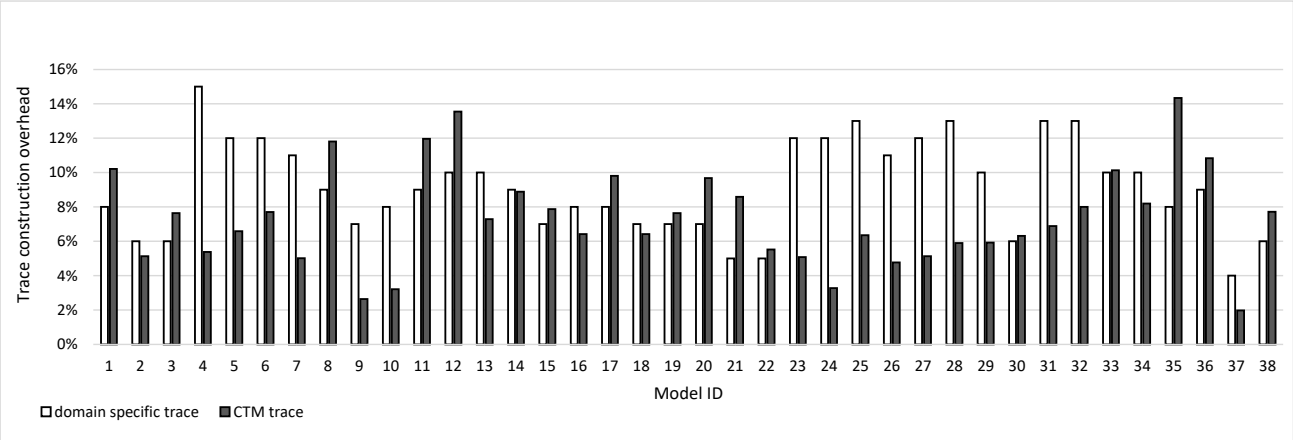

Figure 6.9: Runtime overhead of the CTM and domain-specific trace construction, for each executed model

RQ #4: Information Preservation. For answering this research question, we conducted the experiments on the same 10 fUML models. The compact traces of these models were uncompacted using the *trace de-compactor*, and then compared with the regular ones produced by the regular trace constructor. The results show that compact traces created with our compaction techniques contain the same information as their not-compacted counterparts. Indeed, using the developed trace de-compactor on traces recorded with the compact trace constructor, the same uncompacted traces could be obtained as the ones recorded with the regular trace constructor. Thus, it can be concluded that CTM offers a lossless representation of traces. In particular, no information is lost in the compaction of traces, and the regular trace can be perfectly reconstructed from the compact one without losing data.

RQ #5:Performance Overhead. Figure 6.9 shows the runtime overhead induced by constructing execution traces, i.e., the percentage of additional execution time spent on building a trace, using CTM and domain-specific trace metamodels. The X-axis shows the used example model, while the Y-axis shows the percentage of runtime overhead induced by the construction of execution traces. Although the runtime overhead for constructing traces heavily depends on the considered execution, the results show that on average, the runtime overhead comprises 8.9% for constructing

a CTM trace and 7.25% for building a domain-specific trace. We observe that the construction of a domain-specific trace is faster than the CTM construction. This is expected since the CTM construction process involves different compaction techniques, i.e., the notification framework and the LCM algorithm on the fly and Valiente's tree pattern matching algorithm offline, which causes more overhead on the execution. However, the median overhead remains quite low and under 10%.

Furthermore, for each compaction technique, we measured the execution time needed by the respective operation for executing it, as well as the respective memory consumption. These experiments were carried out on the same 10 fUML models[8]. The execution time was measured by taking timestamps right before the operation for each compaction technique starts, and right after the operation finishes. We repeated the measurements three times and used the arithmetic mean for answering this research question. First, we measured the runtime overhead induced by only constructing execution traces associated to each of the trace compaction techniques (i.e., State, Step, ObjectState, ParameterList). Then, we measured the percentage of the time overhead of the selected models for each compaction technique separately. Figure 6.10 and Table 6.3 show the execution times measured for applying each of the models distinguished between different compaction techniques. Due to space limitations, we only present the total time of the trace construction consumed by each execution. The last column of Table 6.3 shows the percentage of the time required for executing some additional operations not related to the trace compaction. The figure shows that the Step compaction technique obtained the highest ratio of the time overhead (%38) over the total trace construction time. This result is due to the complexity of the Valentine's algorithm, which consists of the time to traverse the tree, the time to compare two subtrees, and the time to compute the signatures.

We used the Eclipse Memory Analyzer (MAT)[9] to measure the memory usage associated to each compaction operation. First, we created a heap dump at the end of each operation run. Then, we measured the number and size of objects allocated on the heap, relevant to trace elements in-

[8] https://github.com/MDSEGroup/TraceCompaction/tree/master/runtime-modelingworkbench/examples.fuml.models

[9] http://www.eclipse.org/mat/

Table 6.3: Time measurement, corresponding to each compaction technique

Model ID	Total time	State	Step	ObjectState	ParameterList	others
1	1.57E+09	%15	%37	%29	%12	%7
2	1.53E+09	%17	%40	%28	%10	%5
3	1.25E+09	%17	%38	%29	%11	%5
4	9.43E+08	%16	%36	%30	%12	%6
5	9.30E+08	%16	%33	%33	%13	%5
6	1.28E+09	%17	%35	%32	%11	%5
7	1.05E+09	%18	%38	%29	%10	%5
8	1.33E+09	%15	%40	%27	%11	%7
9	5.17E+09	%19	%44	%20	%9	%8
10	4.08E+09	%19	%43	%21	%8	%9
Average	1.91E+09	%17	%38	%28	%11	%6

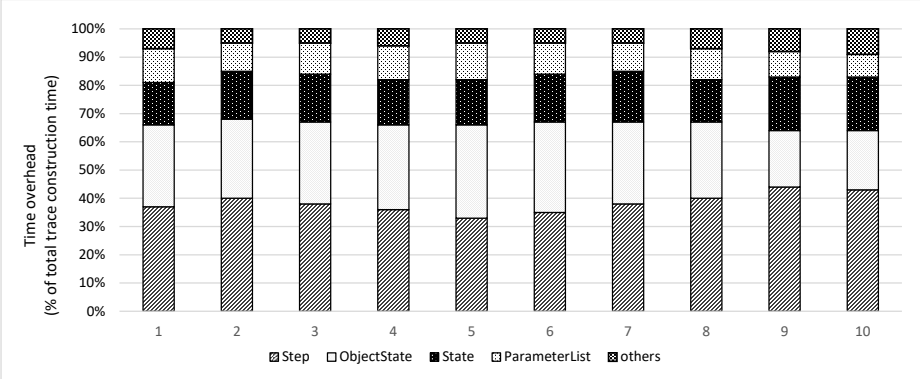

Figure 6.10: Time measurements for CTM trace compaction techniques

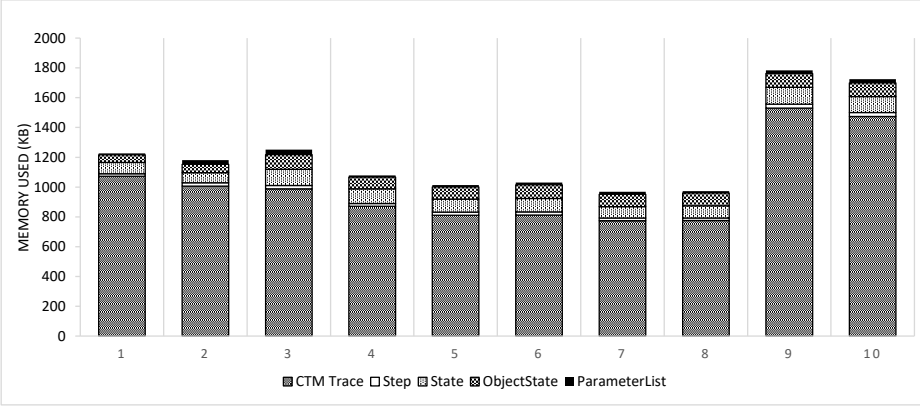

Figure 6.11: Memory consumption measurements for CTM trace elements

Table 6.4: Memory consumption measurement associated to compaction techniques (all measurements are in KBs)

Model ID	Total trace	State	Step	ObjectState	ParameterList
1	1073	17(%2)	76(%7)	50(%5)	11(%1)
2	1007	23(%2)	67(%7)	56(%6)	29(%3)
3	988	22(%2)	109(%11)	98(%10)	34(%3)
4	871	19(%2)	98(%11)	78(%9)	12(%1)
5	811	21(%3)	87(%11)	82(%10)	13(%2)
6	813	20(%2)	91(%11)	91(%11)	16(%2)
7	774	18(%2)	76(%10)	84(%11)	17(%2)
8	776	17(%2)	81(%10)	86(%11)	13(%2)
9	1530	27(%2)	112(%7)	93(%6)	21(%1)
10	1474	27(%2)	107(%7)	91(%6)	25(%2)
Average	1012	21(%9)	90(%8)	81(%2)	19(%2)

stances, and consequently calculated the additional memory consumption caused by various types of compaction, i.e., change notification framework, the LCM algorithm, and the Valiente's tree pattern matching algorithm. Note that for the Step compaction, we created a heap dump before and after it is run, and computed the difference. Figure 6.11 and Table 6.4 show the results of the memory consumption measurements for each of the models distinguished between Step, State, ObjectState, and ParameterList. The results show only the size (in KB) of the objects allocated on the heap for executing each of the compaction techniques. It amounts to 21KB, 90KB, 81KB, and 19KB compared to 1,012 KB allocated for the trace size. Thus, the measured memory consumption overhead lies between 2% and 9%. Such overhead is due mostly to the use of collections such as ArrayList, Hashtable, and HashMap. For example, the LCM algorithm uses ArrayLists of ObjectState and Value instances to deal with frequent ObjectStates.

Overall, from these results, we conclude that CTM compaction techniques cause only a marginal memory overhead. Nevertheless, the memory overhead and the size of captured traces grow linear with the number of executed model elements.

Part IV

Conclusion and Perspectives

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

Dynamic V&V of models requires the ability to capture execution traces for the execution of models. We identified three main requirements regarding execution trace data structures. First, *genericity* is required for a trace structure to support a wide range of xDSMLs, independent of the metaprogramming approaches used for their implementation. Second, *scalability in space* is required to handle large execution traces. Third, *information preservation* must be considered for providing a lossless representation of traces.

In this thesis, we aimed at defining a new trace structure addressing the identified requirements. Hence, we made the following two contributions.

First, in order to understand the state of research on model execution tracing, we conducted a systematic literature review. Our search yielded a classification of 64 existing approaches from 645 research studies found through automatic searches in popular academic online libraries. We analyzed the identified approaches, and classified them based on the following facets: supported types of models, supported execution semantics definition technique, traced data, purpose, data extraction technique, trace representation format, trace representation method, language specificity, data carrier format, and maturity.

Our reviews of the literature shows that there is a lack of approaches that address the three mentioned requirements. The results suggest that more research work is needed particularly on suitable trace representations and broad applicability of approaches with scalability and interoperability being two concerns that have been mostly neglected so far.

As the second contribution, we defined CTM, a metamodel for representing traces generated from executable models that is built with genericity and scalability in space. For this, we presented four complementary contributions:

1. A generic trace metamodel that is defined by identifying a set of key generic concepts needed to express traces for models created with any xDSML. Examples of such generic concepts include the *execution steps* occurring during the execution, *execution states*, *object states*, and processed *parameters*.

2. A generic scalable trace metamodel called the Compact Trace Metamodel (CTM), which relies on a set of compaction techniques to provide a representation of traces in a compact form. CTM is built with scalability in mind, supporting trace compaction techniques at the metamodel level.

3. A process for compressing a regular trace into a compact trace. The process is lossless, meaning that the regular trace can be fully reconstructed from its compact version.

4. A process to uncompress a trace compacted with CTM into its original format.

CTM was applied to five different xDSMLs: two variants of Petri nets, IML, TFSM and fUML. Furthermore, we compared the scalability of CTM traces with traces created using the approach by Bousse et al. [7], which is the only model execution tracing approach that considers some kind of compaction. The results show that the compaction gain reached by a trace represented in CTM is in average 59% in memory usage and 95% disk space.

Overall, we addressed the challenges identified in model execution trace structures. Our contributions not improve the state of the art of solutions for model execution tracing, but illustrate the concrete benefits of CTM as a new trace structure concerning to execution trace management.

7.2 Perspectives

In the following, we discuss interesting directions for future work building upon the research conducted in this study.

7.2.1 Extended pattern detection

There exist two kinds of behavioral patterns in a trace. The first type involves contiguous repetitions of sequences of events due to loops. The second type consists of behavioral patterns that occur in a non-contiguous way in the trace. Our graph reduction technique, used for dealing repetitions in Steps, supports only the first one in which patterns are considered as the sequence of Steps repeated contiguously in the trace. It dose not take into account the non-contiguous repetitions that occur in a trace. In other words, two identical sub-trees that occur in a non contiguous way will be counted twice. This would require applying relevant techniques to the *compact trace constructor* to support non-contiguous patterns.

In addition, as mentioned in Section 5.3.2.2, the step compaction including the pattern detection is performed *offline* (i.e., after the execution of the model). In order to gain better memory-efficiency, such technique must be carried out *on the fly*.

7.2.2 Further evaluation

We intend to test our trace constructor with producing numerous execution traces of real world models to further evaluate the efficiency of CTM. By efficiency, we mean the ability of the construction of execution traces as compactly as possible with minimal run-time overhead.

7.2.3 Combining compaction with compression techniques

CTM uses several compaction techniques that are tailored to remove redundancies, which exist in different parts of a trace. Our approach employed the common compaction techniques that are mostly used for the compaction of trace in code-centric approaches. On the other hand, there

exist various data compression techniques to reduce redundancies in data representation in order to decrease the size of data. Compression can be either lossy or lossless. An interesting topic for future research could be combining data compression techniques with compaction techniques, for providing more scalable and efficient trace.

7.2.4 Applying lens-like abstraction

. In this research, the compaction techniques have been used to traces with the aim of removing repetitions of trace elements. An efficient technique would be lens-like abstraction, which can reduce the size of traces by ignoring details not relevant to the property under study. In fact, an abstract trace model is constructed and, during its analysis, only the main concepts are considered and all details about the system are ignored. Hence, it is possible to analyze the behavior of a system and understand its main content through the analysis of a smaller more compact trace.

7.2.5 Applying process mining abstraction techniques.

As mentioned in Section. 5.4, many abstraction techniques have been proposed in process mining approaches to reduce the size and complexity of traces. In future investigations, it might be possible to use these techniques for abstracting and exploring the content of large model-based traces.

7.2.6 A Tool Suite

The techniques presented in this thesis need to be integrated with trace analysis tools. We need to investigate how existing V&V techniques, specially dynamic analysis, can be used to support such techniques. Finally, we need to work more towards the adoption of CTM by tool builders in academia as well as real industry.

Author's publications

1 F. Hojaji, B. Zamani, and A. Hamou-Lhadj, Towards a tracing framework for Model-Driven software systems, in Proceedings of the 6th International Conference on Computer and Knowledge Engineering (ICCKE), pp.298-303, IEEE, 2016, doi = 10.1109/ICCKE.2016.7802156.

2 F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse, Tracing Executable Models: A Systematic Mapping Study, *Software & Systems Modeling*, 2019, doi = 10.1007/s10270-019-00724-1.

3 F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse, Lossless Compaction of Model Execution Traces, *Software & Systems Modeling*, 2019, doi = 10.1007/s10270-019-00724-1.

Bibliography

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, second ed., 2017.

[2] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, "Neo4EMF, a scalable persistence layer for EMF models," in *Proceedings of the European Conference on Modelling Foundations and Applications*, vol. 8569 of *Lecture Notes in Computer Science*, pp. 230–241, Springer, 2014.

[3] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pp. 42–55, IBM Press, 2004.

[4] A. Hamou-Lhadj and T. C. Lethbridge, "A Metamodel for the compact but lossless Exchange of Execution Traces," *Software & Systems Modeling*, vol. 11, no. 1, pp. 77–98, 2012.

[5] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, O. Barais, and Y. Le Traon, "A native Versioning Concept to Support Historized Models at Runtime," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, vol. 8767 of *Lecture Notes in Computer Science*, pp. 252–268, Springer, 2014.

[6] P. Langer, T. Mayerhofer, and G. Kappel, "Semantic model differencing utilizing behavioral semantics specifications," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, vol. 8767 of *Lecture Notes in Computer Science*, pp. 116–132, Springer, 2014.

[7] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry, "Advanced and efficient execution trace management for executable domain-specific modeling languages," *Software & Systems*

Modeling, vol. 18, pp. 385–421, Feb 2019.

[8] K. Peffers, T. Tuuanen, M. A. Rothenberger, and S. Chatterjee, "A Design Science Research Methodology for Information Systems Research," *Journal of Management Information Systems*, pp. 45–77., 2007.

[9] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[10] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry, "A Generative Approach to Define Rich Domain-Specific Trace Metamodels," in *European Conference on Modelling Foundations and Applications*, vol. 9153 of *Lecture Notes in Computer Science*, pp. 45–61, Springer, 2015.

[11] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, pp. 19–25, Sep. 2003.

[12] B. Combemale, X. Crégut, and M. Pantel, "A Design Pattern to build Executable DSMLs and associated V&V tools," in *Proceedings of the 19th Asia-Pacific on Software Engineering Conference (APSEC)*, vol. 1, pp. 282–287, IEEE, 2012.

[13] A. Hegedus, G. Bergmann, I. Ráth, and D. Varró, "Back-annotation of simulation traces with change-driven model transformations," in *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 145–155, IEEE, 2010.

[14] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel, "xMOF: Executable DSMLs based on fUML," in *Proceedings of the International Conference on Software Language Engineering*, vol. 8225 of *Lecture Notes in Computer Science*, pp. 56–75, Springer, 2013.

[15] J. Tatibouet, A. Cuccuru, S. Gérard, and F. Terrier, "Formalizing Execution Semantics of UML Profiles with fUML Models," in *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, vol. 8767 of *Lecture Notes in Computer Science*, pp. 133–148, Springer, 2014.

[16] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. DeAntoni, and B. Combemale, "Execution framework of the GEMOC studio (tool demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16)*, pp. 84–89, ACM, 2016.

[17] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of UML models: a systematic review of research and practice," *Software & Systems Modeling*, 2018.

[18] Object Management Group, "Meta object facility (mof) core specification," 2014.

[19] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[20] F. Jouault and I. Kurtev, "Transforming models with atl," in *International Conference on Model Driven Engineering Languages and Systems*, vol. 3844 of *Lecture Notes in Computer Science*, pp. 128–138, Springer, 2005.

[21] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet, "Mashup of metalanguages and its implementation in the kermeta language workbench," *Software & Systems Modeling*, vol. 14, no. 2, pp. 905–920, 2015.

[22] A. Schürr, "Specification of graph translators with triple graph grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*.

[23] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[24] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle, "On the use of graph transformation in the formal specification of model interpreters," *J. UCS*, vol. 9, no. 11, pp. 1296–1321, 2003.

[25] L.-Å. Fredlund, B. Jonsson, and J. Parrow, "An implementation of a translational semantics for an imperative language," in *International Conference on Concurrency Theory*, pp. 246–262, Springer, 1990.

[26] E. Lepore and B. Loewer, "Translational semantics," *Synthese*, vol. 48, no. 1, pp. 121–133, 1981.

[27] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry, "Advanced and efficient execution trace management for executable domain-specific modeling languages," *Software & Systems Modeling*, pp. 1–37, 2017.

[28] J. Kraft, A. Wall, and H. M. Kienle, "Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects," in *Proceedings of the International Conference on Runtime Verification*, vol. 6418 of *Lecture Notes in Computer Science*, pp. 315–329, Springer, 2010.

[29] K. Mehner, "JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs," *Software Visualization*, vol. 2269, pp. 163–175, 2002.

[30] T. Ball, "The Concept of Dynamic Analysis," in *ACM SIGSOFT Software Engineering Notes*, vol. 24 of *Lecture Notes in Computer Science*, pp. 216–234, Springer, 1999.

[31] L. Alawneh and A. Hamou-Lhadj, *Execution traces: A new Domain that requires the Creation of a Standard Metamodel*, vol. 59 of *Lecture Notes in Communications in Computer and Information Science book series*, pp. 253–263. Springer, 2009.

[32] E. Bousse, *Execution trace management to support dynamic V&V for executable DSMLs*. Thesis, 2015.

[33] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries," in *International Conference on Parallel Computing*, vol. 22, pp. 481–490, 2011.

[34] X. Zhang and R. Gupta, "Whole execution traces and their applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 3, pp. 301–334, 2005.

[35] STMicroelectronics, "KPTrace Specification," 2012.

[36] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[37] J. DeAntoni, F. Mallet, F. Thomas, G. Reydet, J.-P. Babau, C. Mraidha, L. Gauthier, L. Rioux, and N. Sordon, "RT-simex: retro-analysis of execution traces," in *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 377–378, ACM, 2010.

[38] M. Desnoyers, "Common trace format (ctf) specification (v1.8.2)," 2013.

[39] T. Mayerhofer, P. Langer, and G. Kappel, "A runtime model for fUML," in *Proceedings of the 7th Workshop on Models@ run. time*, pp. 53–58, ACM, 2012.

[40] X. Crégut, B. Combemale, M. Pantel, R. Faudoux, and J. Pavei, "Generative Technologies for Model Animation in the TopCased Platform," *ECMFA*, vol. 6138, pp. 90–103, 2010.

[41] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer, "ProMoBox: A Framework for Generating Domain-specific Property Languages," in *Proceedings of the International Conference on Software Language Engineering (SLE)*, vol. 8706 of *Lecture Notes in Computer Science*, pp. 1–20, Springer, 2014.

[42] P. Kemper and C. Tepper, "Automated trace analysis of discrete-event system models," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 195–208, 2009.

[43] Object Management Group, "Uml testing profile (utp)," 2013.

[44] L. Alawneh, A. Hamou-Lhadj, and J. Hassine, "Towards a common metamodel for traces of high performance computing systems to enable software analysis tasks," in *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 111–120, IEEE, 2015.

[45] S. Maoz and D. Harel, "On tracing reactive systems," *Software & Systems Modeling*, vol. 10, no. 4, pp. 447–468, 2011.

[46] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of java program," in *Eighth International Workshop on Principles of Software Evolution*, pp. 148–151, IEEE, 2005.

[47] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (OTF)," in *International Conference on Compuatational Science*, vol. 3992 of *Lecture Notes in Computer Science*, pp. 526–533, Springer, 2006.

[48] G. Pagano, D. Dosimont, G. Huard, V. Marangozova-Martin, and J.-M. Vincent, "Trace management and analysis for embedded systems," in *7th International Symposium on Embedded Multicore Socs (MCSoC)*, pp. 119–122, IEEE, 2013.

[49] R. Aydt, "The Pablo self-defining data format," 1992.

[50] L. M. Schnorr, O. Stein, and J. Chassin, "Paje Trace File Format," report, 2013.

[51] B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, and M. Pantel, "Introducing simulation and model animation in the MDE Topcased toolkit," in *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS)*, 2008.

[52] A. Hegedus, G. Bergmann, I. Ráth, and D. Varró, "Replaying execution trace models for dynamic modeling languages," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 56, no. 3, pp. 71–82, 2013.

[53] F. Hilken, L. Hamann, and M. Gogolla, "Transformation of UML and OCL models into Filmstrip Models," in *International Conference on Theory and Practice of Model Transformations*, vol. 8568 of *Lecture Notes in Computer Science*, pp. 170–185, Springer, 2014.

[54] F. Hilken and M. Gogolla, "Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping," in *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pp. 708–713, IEEE, 2016.

[55] O. M. G. (OMG), "XML Metadata Interchange specification, version 2.5.1," 2011.

[56] Isocpp.org, "Serialization and unserialization," [April 1, 2015].

[57] W3C, "Efficient Extensible Markup Language (XML) Interchange (EXI), Format 1.0," standard, IJIS Institute Technical Advisory Committee, 2014.

[58] D. Crockford, "The application/JSON media type for javascript object notation (JSON)," RFC 4627, 2006.

[59] K. Varda, "Google Protocol Buffers: Google's Data Interchange Format," tech. rep., 2008.

[60] W. De Pauw, D. H. Lorenz, J. M. Vlissides, and M. N. Wegman, "Execution patterns in object-oriented visualization," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, vol. 4, pp. 219–234, 1998.

[61] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: semantic differencing for activity diagrams," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 179–189, ACM, 2011.

[62] R. Sharp and A. Rountev, "Interactive exploration of uml sequence diagrams," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pp. 1–6, IEEE, 2005.

[63] C. Prada-Rojas, M. Santana, S. De-Paoli, and X. Raynaud, "Summarizing embedded execution traces through a compact view," in *Conference on System Software, SoC and Silicon Debug (S4D)*, 2010.

[64] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu, "Seat: A usable trace analysis tool," in *13th International Workshop on Program Comprehension, IWPC 2005*, pp. 157–160, IEEE, 2005.

[65] A. Hamou-Lhadj, *Techniques to simplify the analysis of execution traces for program comprehension*. Thesis, 2005.

[66] K. Noda, T. Kobayashi, and K. Agusa, "Execution trace abstraction based on meta patterns usage," in *Working Conference on Reverse Engineering (WCRE)*, pp. 167–176, IEEE, 2012.

[67] K. Noda, T. Kobayashi, K. Agusa, and S. Yamamoto, "Sequence diagram slicing," in *Asia-Pacific Software Engineering Conference, APSEC'09.*, pp. 291–298, IEEE, 2009.

[68] J. Quante and R. Koschke, "Dynamic object process graphs," *Journal of Systems and Software*, vol. 81, no. 4, pp. 481–501, 2008.

[69] P. Dugerdil and J. Repond, "Automatic generation of abstract views for legacy software comprehension," in *3rd India software engineering conference*, pp. 23–32, ACM, 2010.

[70] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency," in *Eighth European Conference on Asia-Pacific Software Maintenance and Reengineering, CSMR 2004*, pp. 329–338, IEEE, 2004.

[71] R. J. C. Bose and W. M. van der Aalst, "Trace clustering based on conserved patterns: Towards achieving better process models," in *Business Process Management Workshops*, vol. 43, pp. 170–181, Springer, 2009.

[72] M. Song, C. W. GÃijnther, and W. M. Van der Aalst, "Trace clustering in process mining," in *Business Process Management Workshops*, pp. 109–120, Springer, 2009.

[73] B. Korel and J. Rilling, "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, no. 11, pp. 647–659, 1998.

[74] R. Smith and B. Korel, "Slicing event traces of large software systems," *arXiv preprint cs/0101005*, 2001.

[75] D. M. Dhamdhere, K. Gururaja, and P. G. Ganu, "A compact execution history for dynamic slicing," *Information Processing Letters*, vol. 85, no. 3, pp. 145–152, 2003.

[76] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of Java programs," *Software Visualization*, vol. 2269, pp. 647–650, 2002.

[77] K. Sartipi and H. Safyallah, "An environment for pattern based dynamic analysis of software systems," *Comprehension through Dynamic Analysis*, p. 12, 2006.

[78] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Eleventh International Conference on Data Engineering*, pp. 3–14, IEEE, 1995.

[79] M. Crochemore, "An optimal algorithm for computing the repetitions in a word," *Information Processing Letters*, vol. 12, no. 5, pp. 244–250, 1981.

[80] D. B. Lange and Y. Nakamura, "Object-oriented program tracing and visualization," *Computer*, vol. 30, no. 5, pp. 63–70, 1997.

[81] F. Francois, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "Kevoree modeling framework (kmf): Efficient modeling techniques for runtime use," *arXiv preprint arXiv:1405.6817*, 2014.

[82] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "An eclipse modelling framework alternative to meet the models@ runtime requirements," in *International Conference on Model Driven Engineering Languages and Systems*, vol. 7590, pp. 87–101, Springer, 2012.

[83] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesely, 2009.

[84] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[85] A. Hegedus, G. Bergmann, I. Ráth, and D. Varró, "Back-annotation of simulation traces with change-driven model transformations," in *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 145–155, IEEE, 2010.

[86] G. Graefe and L. D. Shapiro, "Data compression and database performance," in *Symposium on Applied Computing*, pp. 22–27, IEEE, 1991.

[87] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a column-oriented DBMS," in *Proceedings of the 31st international conference on Very large data bases*, pp. 553–564, VLDB Endowment, 2005.

[88] C. Dunn, "Smile! you're on RLE," *The Transactor*, vol. 7, no. 6, pp. 16–18, 1987.

[89] S. Kodituwakku and U. Amarasinghe, "Comparison of lossless data compression algorithms for text data," *Indian journal of computer science and engineering*, vol. 1, no. 4, pp. 416–425, 2010.

[90] D. Abadi, "Teradata RainStor's Compression and Performance Technology." http://blogs.teradata.com/data-points/teradata-rainstors-compression-performance-technology/, [April 1, 2015].

[91] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "LePetersen2015ssons from Applying the Systematic Literature Review process within the Software Engineering Domain," *Journal of systems and software*, vol. 80, no. 4, pp. 571–583, 2007.

[92] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[93] R. J. Adams, P. Smart, and A. S. Huff, "Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies," *International Journal of Management Reviews*, vol. 19, no. 4, pp. 432–454, 2017.

[94] F. Hojaji, B. Zamani, and A. Hamou-Lhadj, "Towards a tracing framework for Model-Driven software systems," in *Proceedings of the 6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 298–303, IEEE, 2016.

[95] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," report, Software Engineering Group, School of Computer Science and Mathematics, Keele University, 2000.

[96] I. Santiago, Ã. JimÃl'nez, J. M. Vara, V. De Castro, V. A. Bollati, and E. Marcos, "Model-Driven Engineering as a new landscape for traceability management: A systematic literature review," *Information and Software Technology*, vol. 54, no. 12, pp. 1340–1356, 2012.

[97] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transaction on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[98] Á. J. Cuadros López, C. Galindres, and P. Ruiz, "Project maturity evaluation model for SMEs from the software development sub-sector," *AD-minister*, no. 29, pp. 147–162, 2016.

[99] J. P. Calvez, *Embedded Real-time Systems. A specification and Design Methodology*. John Wiley, 1993.

[100] O. Pasquier and J. P. Calvez, "An object-based executable model for ation of real-time Hw/Sw systems," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 782–783, IEEE, 1999.

[101] P. Kemper and C. Tepper, "Automated analysis of simulation traces-separating progress from repetitive behavior," in *Proceedings of the Fourth International Conference on the Quantitative Evaluation of Systems.QEST 2007*, pp. 101–110, IEEE, 2007.

[102] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, *et al.*, "COLA–The component language," tech. rep., 2007.

[103] W. Haberl, J. Birke, and U. Baumgarten, "A middleware for model-based embedded systems," in *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, pp. 253–259, 2008.

[104] W. Haberl, M. Herrmannsdoerfer, J. Birke, and U. Baumgarten, "Model-level debugging of Embedded Real-time Systems," in *Proceedings of the 10th international conference on Computer and information technology (CIT)*, pp. 1887–1894, IEEE, 2010.

[105] J. DeAntoni and F. Mallet, "Timesquare: Treat your models with logical time," in *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS)*, vol. 7304, pp. 34–41, Springer, 2012.

[106] K. Garcés, J. Deantoni, and F. Mallet, "A model-based approach for reconciliation of polychromous execution traces," in *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 259–266, IEEE, 2011.

[107] A. Krasnogolowy, S. Hildebrandt, and S. Wätzoldt, "Flexible debugging of behavior models," in *IEEE International Conference on Industrial Technology (ICIT)*, pp. 331–336, IEEE, 2012.

[108] L. Li, X. Li, and S. Tang, "Research on web application consistency testing based on model simulation," in *Proceedings of the 9th International Conference on Computer Science and Education (ICCSE)*, pp. 1121–1127, IEEE, 2014.

[109] A. Intana, M. R. Poppleton, and G. V. Merrett, "A model-based trace testing approach for validation of formal co-simulation models," in *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pp. 181–188, Society for Computer Simulation International, 2015.

[110] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux, "Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification," *Journal of Software (JSW)*, vol. 4, no. 9, pp. 943–958, 2009.

[111] C. A. Fernández-Fernández and A. J. Simons, "An Algebra to Represent Task Flow Models," *International Journal of Computational Intelligence: Theory and Practice*, vol. 6, no. 2, pp. 63–74, 2011.

[112] C. Fernández-Fernández and A. Simons, "An Implementation of the Task Algebra, a Formal Specification for the Task Model in the Discovery Method," *Journal of applied research and technology*, vol. 12, no. 5, pp. 908–918, 2014.

[113] T. Mayerhofer, "Testing and debugging UML models based on fUML," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 1579–1582, IEEE, 2012.

[114] S. Mijatov, P. Langer, T. Mayerhofer, and G. Kappel, "A framework for testing UML activities based on fUML," in *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, vol. 1069, pp. 1–10, Springer, 2013.

[115] S. Mijatov, T. Mayerhofer, P. Langer, and G. Kappel, "Testing functional requirements in UML activity diagrams," in *International Conference on Tests and Proofs*, vol. 9154 of *Lecture Notes in Computer Science*, pp. 173–190, Springer, 2015.

[116] L. Yilmaz, "Automated object-flow testing of dynamic process interaction models," in *Proceedings of the ation Conference, Proceedings of the Winter*, vol. 1, pp. 586–594, IEEE, 2001.

[117] M. Hendriks and F. W. Vaandrager, "Reconstructing Critical Paths from Execution Traces," in *Proceedings of the 15th International Conference on Computational Science and Engineering (CSE)*, pp. 524–531, IEEE, 2012.

[118] M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers, "Analyzing Execution Traces: Critical-path Analysis and Distance Analysis," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 4, pp. 487–512, 2016.

[119] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen, *et al.*, *Object-oriented modeling and design*, vol. 199. Prentice-hall Englewood Cliffs, NJ, 1991.

[120] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting efficient and advanced omniscient debugging for xDSMLs," in *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering*, pp. 137–148, ACM, 2015.

[121] E. Bousse, B. Combemale, and B. Baudry, "Towards Scalable Multidimensional Execution Traces for xDSMLs," in *Proceedings of the 11th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE (MoDeVVa 2014)*, vol. 1235, pp. 13–18, 2014.

[122] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry, "Omniscient debugging for Executable DSLs," *Journal of Systems and Software*, vol. 137, pp. 261–288, 2018.

[123] H. Aljamaan and T. C. Lethbridge, "Towards Tracing at the Model Level," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pp. 495–498, IEEE, 2012.

[124] H. Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying trace directives for UML attributes and state machines," in *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pp. 79–86, IEEE, 2014.

[125] H. I. Aljamaan, T. Lethbridge, M. Garzón, and A. Forward, "UmpleRun: a dynamic analysis tool for textually modeled state machines using Umple," in *Proceedings of the First International Workshop on Executable Modeling co-located with MODELS 2015*, pp. 16–20, 2015.

[126] H. Aljamaan, T. C. Lethbridge, and M. A. Garzón, "MOTL: A Textual Language for Trace Specification of State Machines and Associations," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, (Riverton, NJ, USA), pp. 101–110, IBM Corp., 2015.

[127] S. Maoz, "Using model-based traces as runtime models," *IEEE Computer Society*, vol. 42, pp. 28–36, 2009.

[128] S. Maoz, "Model-based traces," in *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, vol. 5421 of *Lecture Notes in Computer Science*, pp. 109–119, springer, 2009.

[129] S. Maoz, J. O. Ringert, and B. Rumpe, "Summarizing semantic model differences," *arXiv preprint arXiv:1409.2307*, 2014.

[130] E. Domínguez, B. Pérez, and M. A. Zapata, "A UML profile for dynamic execution persistence with monitoring purposes," in *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pp. 55–61, IEEE, 2013.

[131] Z. Hu and S. M. Shatz, "Mapping UML Diagrams to a Petri Net Notation for System ation," in *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pp. 213–219, Citeseer, 2004.

[132] J. Lian, Z. Hu, and S. M. Shatz, "ation-based analysis of UML statechart diagrams: methods and case studies," *Software Quality Journal*, vol. 16, no. 1, pp. 45–78, 2008.

[133] L. Wang, E. Wong, and D. Xu, "A Threat Model Driven Approach for Security Testing," in *Proceedings of the 3th International Workshop on Software Engineering for Secure Systems, ICSE Workshops*, pp. 10–17, IEEE, 2007.

[134] L. Fuentes and P. Sánchez, "Designing and Weaving Aspect-Oriented Executable UML Models," *Journal of Object Technology*, vol. 6, no. 7, pp. 109–136, 2007.

[135] L. Fuentes and P. Sánchez, "Towards Executable Aspect-Oriented UML Models," in *Proceedings of the 10th international workshop on Aspect-oriented modeling*, pp. 28–34, ACM, 2007.

[136] L. Fuentes, J. Manrique, and P. Sánchez, "Execution and ation of (profiled) UML models using Populo," in *Proceedings of the international workshop on Models in software engineering*, pp. 75–81, ACM, 2008.

[137] L. Fuentes and P. Sánchez, "Dynamic Weaving of Aspect-Oriented Executable UML Models," *Transactions on Aspect-Oriented Software Development*, vol. 5560, pp. 1–38, 2009.

[138] M. L. Crane and J. Dingel, "Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities," in *Conference of the center for advanced studies on collaborative research*, pp. 96–110, ACM, 2008.

[139] B. Combemale, L. Gonnord, and R. Rusu, "A Generic Tool for Tracing Executions back to a DSMLs Operational Semantics," in *European Conference on Modelling Foundations and Applications*, vol. 6698, pp. 35–51, springer, 2011.

[140] A. Goel, B. Sengupta, and A. Roychoudhury, "Footprinter: Round-trip engineering via scenario and state based models," in *Proceedings of the 31st International Conference on Software Engineering - Companion Volume, ICSE-Companion*, pp. 419–420, IEEE, 2009.

[141] A. Derezinska and M. Szczykulski, "Tracing of state machine execution in the model-driven development framework," in *Proceedings of the 2nd International Conference on Information Technology, ICIT 2010*, pp. 517–524, IEEE, 2010.

[142] M. A. Wehrmeister, J. G. Packer, and L. M. Ceron, "Framework to simulate the behavior of embedded real-time systems specified in UML models," in *Brazilian Symposium on Computing System Engineering (SBESC)*, pp. 1–7, IEEE, 2011.

[143] M. A. Wehrmeister, J. G. Packer, L. M. Ceron, and C. E. Pereira, "Towards Early Verification of UML Models for Embedded and Real-Time Systems," *Embedded Systems, Computational Intelligence and Telematics in Control*, vol. 45, no. 4, pp. 25–30, 2012.

[144] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France, "From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics," in *Modellierung*, vol. 225, pp. 273–288, 2014.

[145] R. Deshayes, B. Meyers, T. Mens, and H. Vangheluwe, "ProMoBox in Practice: A Case Study on the GISMO Domain-Specific Modelling Language," in *Proceedings of the 8th Workshop on Multi-Paradigm Modelling (MPM)*, pp. 21–30, 2014.

[146] Scopus, *A Generic Framework for Realizing Semantic Model Differencing Operators*, vol. 1258, 2014.

[147] J. P. Faria and A. C. Paiva, "A Toolset for Conformance Testing against UML sequence diagrams based on event-driven colored Petri nets," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 285–304, 2016.

[148] B. Lima and J. P. Faria, "An approach for automated scenario-based testing of distributed and heterogeneous systems," in *Proceedings of the 10th International Joint Conference on Software Technologies (ICSOFT)*, vol. 1, pp. 1–10, IEEE, 2015.

[149] S. Schivo, B. M. Yildiz, E. Ruijters, C. Gerking, R. Kumar, S. Dziwok, A. Rensink, and M. Stoelinga, "How to Efficiently Build a Front-End Tool for UPPAAL: A Model-Driven Approach," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, vol. 10606 of *Lecture Notes in Computer Science*, pp. 319–336, Springer, 2017.

[150] A. Hamou-Lhadj and T. Lethbridge, "A metamodel for the compact but lossless exchange of execution traces," *Software & Systems Modeling*, vol. 11, no. 1, pp. 77–98, 2012.

[151] A. Hamou-Lhadj and T. Lethbridge, "A metamodel for dynamic information generated from object-oriented systems," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 59–69, 2004.

[152] L. Alawneh and A. Hamou-Lhadj, "An exchange format for representing dynamic information generated from High Performance Computing applications," *Future Generation Computer Systems*, vol. 27, no. 4, pp. 381–394, 2011.

[153] M. Szvetits and U. Zdun, "Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime," *Software & Systems Modeling*, vol. 15, no. 1, pp. 31–69, 2013.

[154] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: a Systematic Review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pp. 31–36, ACM, 2007.

[155] H. G. Gurbuz and B. Tekinerdogan, "Model-based Testing for Software Safety: a Systematic Mapping Study," *Software Quality Journal*, pp. 1–46, 2017.

[156] P. H. Nguyen, M. Kramer, J. Klein, and Y. Le Traon, "An Extensive Systematic Review on the Model-Driven Development of Secure Systems," *Information and Software Technology*, vol. 68, pp. 62–81, 2015.

[157] L. M. do Nascimento, D. L. Viana, P. A. S. Neto, D. A. Martins, V. C. Garcia, and S. R. Meira, "A Systematic Mapping Study on Domain Specific Languages," in *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA'12)*, pp. 179–187, 2012.

[158] F. D. Giraldo, S. Espana, and O. Pastor, "Analyzing the concept of Quality in Model-Driven Engineering Literature: A systematic review," in *Proceedings of the 8th International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–12, IEEE, 2014.

[159] F. Hojaji, T. Mayerhofer, B. Zamani, A. Hamou-Lhadj, and E. Bousse, "Model Execution Tracing: A Systematic Mapping Study," *Software & Systems Modeling*, pp. 1–25, Feb 2019.

[160] G. Erich, H. Richard, J. Ralph, and V. John, *Design patterns: Elements of reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[161] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence diagram from Execution Trace of Java program," in *Eighth International Workshop on Principles of Software Evolution*, pp. 148–151, IEEE, 2005.

[162] L. Alawneh and A. Hamou-Lhadj, "An exchange format for representing dynamic information generated from high performance computing applications," *Elsevier Journal of Future Generation Computer Systems*, vol. 27, no. 4, pp. 381–394, 2011.

[163] N. Tax, N. Sidorova, and W. M. P. van der Aalst, "Discovering more precise process models from event logs by filtering out chaotic activities," *Journal of Intelligent Information Systems*, vol. 52, pp. 107–139, Feb 2019.

[164] R. P. Jagadeesh Chandra Bose and W. M. P. van der Aalst, "Abstractions in Process Mining: A Taxonomy of Patterns," in *Business Process Management* (U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, eds.), vol. 5701 of *Lecture Notes in Computer Science*, (Berlin, Heidelberg), pp. 159–175, Springer Berlin Heidelberg, 2009.

[165] M. Song, C. W. Günther, and W. M. Van der Aalst, "Trace clustering in process mining," in *International Conference on Business Process Management*, vol. 17 of *Lecture Notes in Business Information Processing*, pp. 109–120, Springer, 2008.

[166] C. W. Günther and W. M. Van Der Aalst, "Fuzzy mining–adaptive process simplification based on multi-perspective metrics," in *International conference on business process management*, vol. 4714 of *Lecture Notes in Computer Science*, pp. 328–343, Springer, 2007.

[167] C. Diamantini, L. Genga, and D. Potena, "Behavioral process mining for unstructured processes," *Journal of Intelligent Information Systems*, vol. 47, pp. 5–32, Aug 2016.

[168] V. Liesaputra, S. Yongchareon, and S. Chaisiri, "Efficient Process Model Discovery Using Maximal Pattern Mining," in *Business Process Management* (H. R. Motahari-Nezhad, J. Recker, and M. Weidlich, eds.), vol. 9253 of *Lecture Notes in Computer Science*, (Cham), pp. 441–456, Springer International Publishing, 2015.

[169] N. Tax, N. Sidorova, R. Haakma, and W. M. P. van der Aalst, "Event abstraction for process mining using supervised learning techniques," *Lecture Notes in Networks and Systems*, pp. 251–269, Aug 2017.

[170] S. Schivo, J. Scholma, B. Wanders, R. A. U. Camacho, P. E. van der Vet, M. Karperien, R. Langerak, J. van de Pol, and J. N. Post, "Modeling biological pathway dynamics with timed automata," *IEEE journal of biomedical and health informatics*, vol. 18, no. 3, pp. 832–839, 2014.

[171] K. Barmpis and D. S. Kolovos, "Comparative analysis of data persistence technologies for large-scale models," in *Proceedings of the 2012 Extreme Modeling Workshop*, pp. 33–38, ACM, 2012.

[172] J. E. Pagán, J. S. Cuadrado, and J. G. Molina, "Morsa: A scalable approach for persisting and accessing large models," in *International Conference on Model Driven Engineering Languages and Systems*, vol. 6981 of *Lecture Notes in Computer Science*, pp. 77–92, Springer, 2011.

[173] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon, "Analyzing complex data in motion at scale with temporal graphs," in *The 29th International Conference on Software Engineering and Knowledge Engineering (SEKE'17)*, p. 6, KSI Research, 2017.

[174] GEMOC, "Gemoc studio documentation." http://download.eclipse.org/gemoc/docs/nightly/index.html.

[175] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet, "Mashup of metalanguages and its implementation in the kermeta language workbench," *Software & Systems Modeling*, vol. 14, no. 2, pp. 905–920, 2015.

[176] G. Valiente, "Simple and efficient tree pattern matching," report, Technical University of Catalonia, 2000.

[177] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," in *Proceedings of Workshop on Frequent itemset Mining Implementations (FIMI'03)*, vol. 90, 2003.

[178] Object Management Group, "Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.3," July 2017.

Appendix A

CTM Application and Setup

This appendix provides the required steps to setup the CTM prototype. Using this prototype, language designers are able to generate scalable traces of executable models for any xDSMLs. To clarify the application of CTM in practice, its usage as an xDSML is described step by step.

A.1 Introduction

This chapter helps you to setup, configure, and run the CTM prototype in order to generate a compact trace for a given xDSML. Given an xDSML and an executable model conforming to the xDSML, the model can be executed, and its trace generated in a compact form. To show the applicability and usage of the prototype, the following directions will help you to configure, and run the prototype for several xDSMLs.

A.2 Install Eclipse Gemoc Studio

Download and install the latest Eclipse Gemoc Studio by following the directions from its supporting web page[1]. Check the installation details from the menu Help>About Gemoc Studio>Installation Details (see Figure A.1). It must be mentioned that the used Eclipse Gemoc Studio tool suite version is 2.3.0-SNAPSHOT here.

The least requirements for Eclipse packages are Java 8 and JavaFX. JavaFX is required for the Multidimensional timeline features. you can also install additional components using the component discovery service: Help > Install additional Gemoc components.

[1] http://download.eclipse.org/gemoc/packages/nightly/

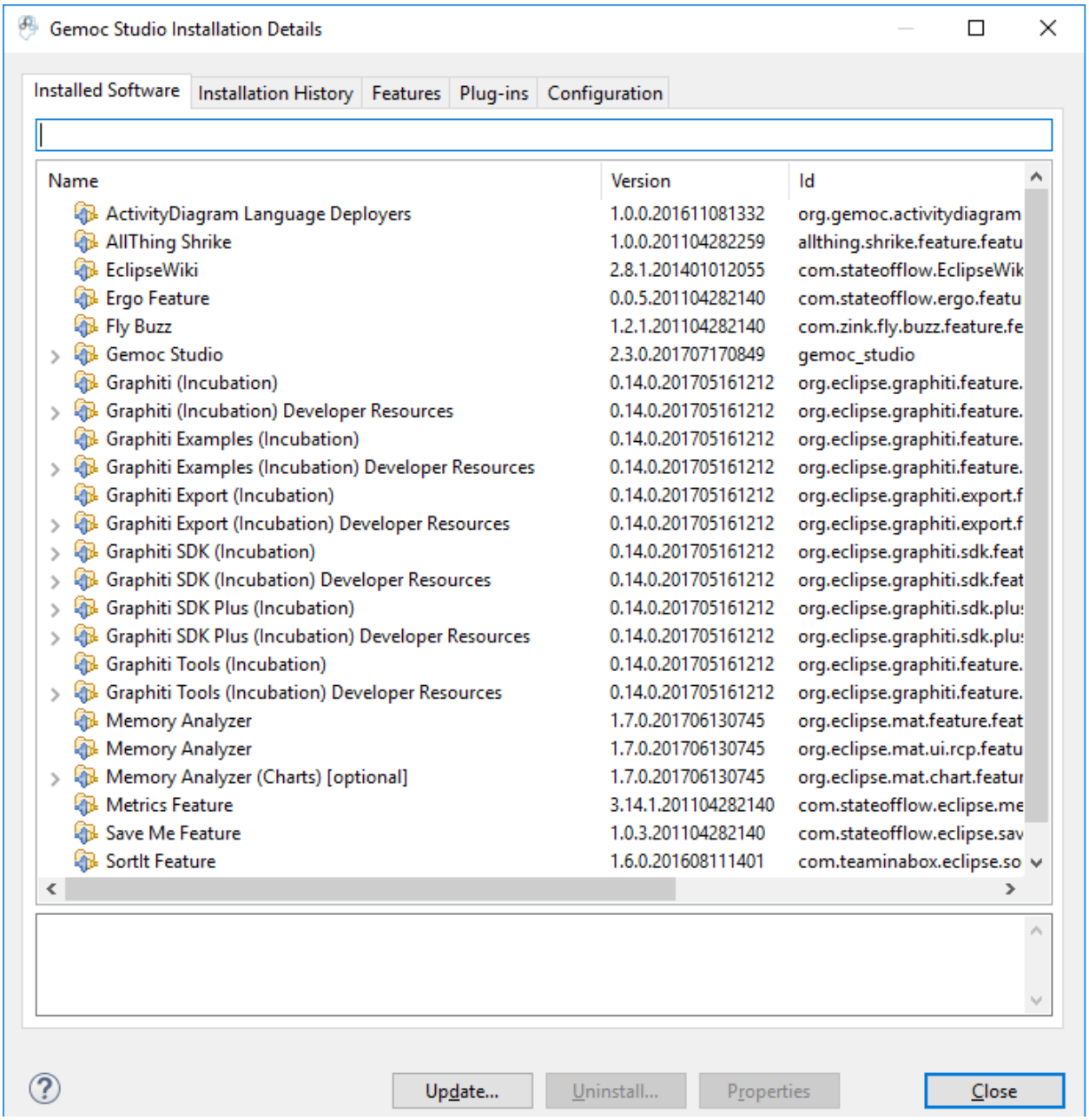

Figure A.1: Gemoc Studio Installation details

A.2.1 Features in Gemoc Studio 2.3.0

This section[2] lists the main features of the current version of the Gemoc Studio, which are related to our work.

A.2.1.1 Ecore Tools

EcoreTools provides a complete graphical modeler to create, edit and analyze Ecore models.

Website : http://www.eclipse.org/ecoretools/

Documentation : http://www.eclipse.org/ecoretools/doc/

A.2.1.2 Gemoc Execution Engine

This feature allows to run a given model according to an xDSML definition. It provides an interface to define, and run a new simulation. It also supports different execution engines associated to their specific metalanguages, and the behavioral coordination of DSLs.

A.2.1.3 Gemoc Language Designer

This feature provides facilities to create new executable languages. The language designer supports the following services.

- Language definition tools
- Editor definition (textual, graphical) tools

A.2.1.4 Gemoc Modeling Workbench

The modeling workbench allows creating and executing models conformant to executable DSMLs. It provides the following services.

- Gemoc Execution Engine

[2]The content of this subsection has been taken from the Gemoc Initiative web site. http: gemoc.org

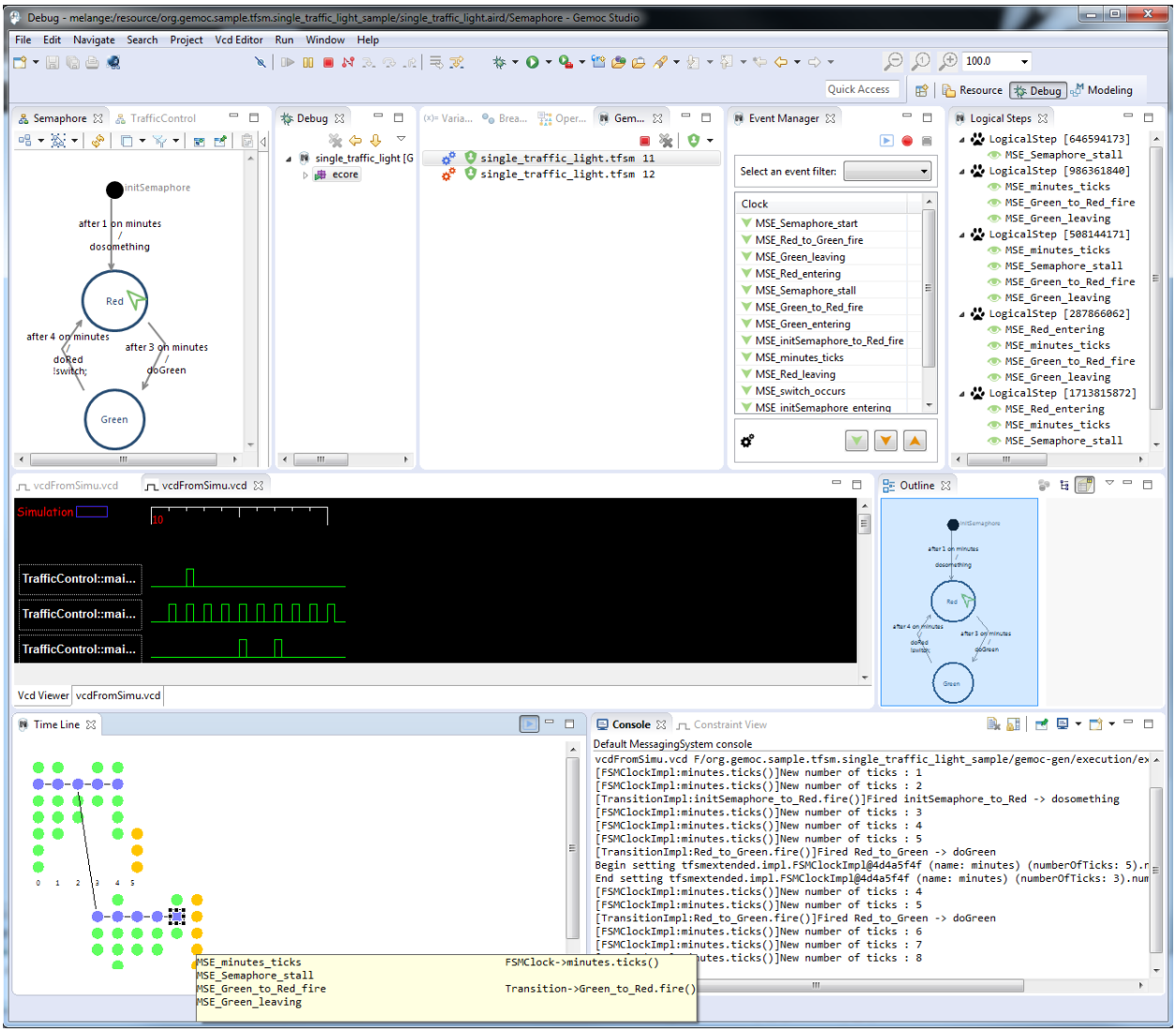

Figure A.2: Screenshot of Gemoc Studio Modeling Workbench on the TFSM example

- Gemoc Animation
- Model edition

Figure A.2 shows the screenshot of Gemoc Studio Modeling Workbench for the execution and animation of a TFSM example.

A.2.1.5 Kermeta 3

Kermeta 3 is a metaprogramming environment based on Xtend language that provides aspect oriented and model typing facilities. It is used to define the execution data and execution functions through aspects weaved onto the metaclasses of the Domain Model.

A.2.1.6 Sirius

This fetaure allows to easily create the graphical modeling workbench by leveraging the Eclipse Modeling technologies, including EMF and GMF.

A.2.1.7 Xtend

Xtend is a programming language which compiles to Java source code. it is syntactically and semantically based on the Java programming language.

A.3 Download and setup CTM

This section provides the required steps to download, and setup CTM tool. First we give an overview of the CTM tool, then we describe the required configuration for executing a model.

A.3.1 CTM Tool Overview

Download the CTM's resources from the project web page[3]. Unzip the archive file. There are two main folders in the root folder:

Traceconstruction: includes the following plugins; *Tracemanagement* that is the core of CTM tool containing trace constructor add-on, trace decompactor and several plugins responsible to create semantics for the trace metamodel. *Tracemetamodel* that consists of the ecore of generic metamodel and CTM, and the genmodel files. Beside these plugins, there are several folders, each dealing with a particular xDSML, which contains a set of plugins for executing a model conforming

[3]https://github.com/MDSEGroup/TraceCompaction

to the respective xDSML. The xDSML are IML, TFSM, Petrinet, Petrinetcomplex, fUML, xMOF Gemoc Engine, xMOF virtual machine.

runtime-modelingworkbench: contains several example models conforming to all tested xDSML, and a plugins with a lunch configuration file.

By using these plugins, you can execute any model conforming to a specific xDSML, and create its trace in both regular and compact form. Also, using these plugins, a uncompacted trace can be reconstructed from a compact trace. While this tool allows executing and constructing compact traces, it allows applying required plugins (abstract syntax and execution semantics) for a new executable DSML, and executing models conforming to the language. Figure A.3 shows a screenshot of the CTM workspace in Gemoc Studio. In the left of the figure, there exist different working sets, each containing a set of plugins related to the working set. For instance, *trace management* consists of five plugins, dealing with trace construction and trace decompaction. TFSM working set contains a set of plugins that define the abstract syntax, operation semantics and the melange language for the TFSM. These plugins allow to debug, and run a TFSM model. This structure is the same for the other languages.

A.3.2 Launch Configuration

You can use the Run and Debug actions (in Run Menu) to start running and debugging a model, respectively. These two actions are configured using the launch configuration. Figure A.4 shows the debug configuration for a TFSM model. You can see the description of each field on the figure.

A.3.3 Trace Generation

Using the launch configuration described in the previous section, you can execute a sample model and generate trace. We have defined four Boolean variables for specifying the compaction of different parts of the trace. Listing A.3.1 presents the definition of these variables defined in the *GenericTraceConstructor* class.

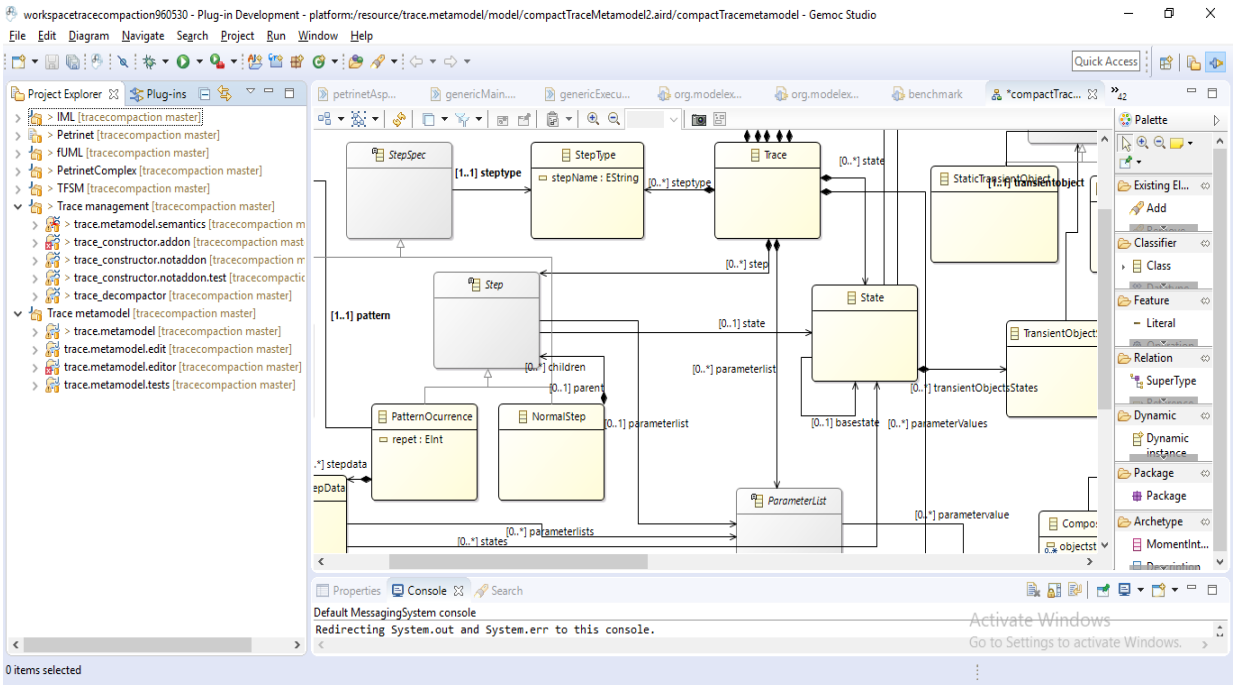

Figure A.3: Screenshot of the CTM workspace

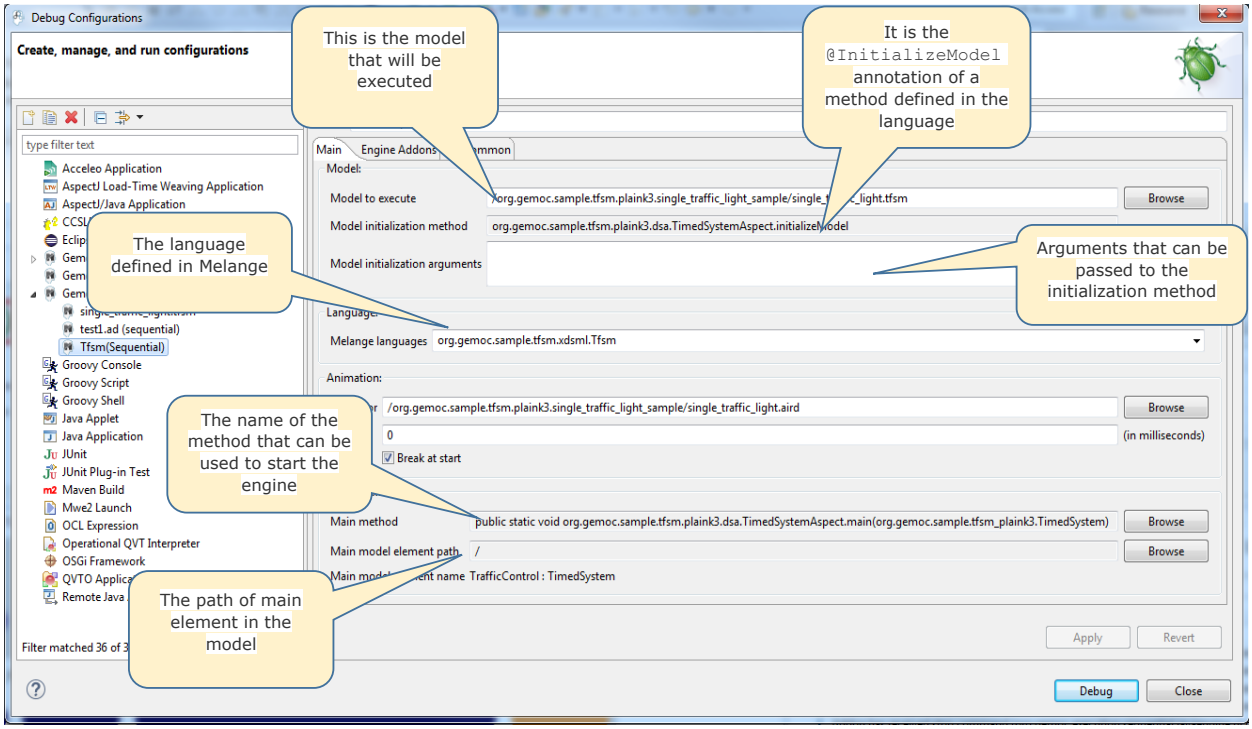

Figure A.4: An example of debug configuration for a TFSM model

```
1public class GenericTraceConstructor {
2// false: State without compaction , True: State with compaction
3  val boolean statecompaction=false
4//  false: Parameter without compaction , True: Parameter with
    compaction
5  val boolean parametercompaction=false
6// false: ObjectState without compaction , True: ObjectState with
    compaction
7  val boolean objectstatecompaction=false
8//  false: Step without compaction , True: Step with compaction
9  val boolean stepcompaction=false
10  ....
11 }
```

Listing A.3.1: Excerpt of the *GenericTraceConstructor* class, which defines Boolean variables for compaction, written in Xtend

```
public static void createSerializedTrace(Trace trace,String tracename)
      {
  // Register the XMI resource factory for the .trace extension
  Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
  Map<String, java.lang.Object> m = reg.getExtensionToFactoryMap();
  m.put("xmi", new XMIResourceFactoryImpl());
  // Obtain a new resource set
  ResourceSet resSet = new ResourceSetImpl();
  // create a resource
  Resource resource =
  resSet.createResource(URI.createURI("trace/" + tracename+filename    )
   );
  // Get the first model element and cast it to the right type, in my
  // example everything is hierarchical included in this first node
    resource.getContents().add(trace);
    try {
        resource.save(Collections.EMPTY_MAP);
         System.out.print("saving Xmi successfull");
      }
    catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
      }
  }
```

Listing A.3.2: Serialization of a trace in an xml file

The trace is generated, and serialized in both XML and EXI formats. Listing A.3.2 shows how the trace is serialized. Figure A.5 and figure A.6 show an excerpt of the trace of an fUML model serialized in XML and EXI formats respectively.

```
<?xml version="1.0" encoding="ASCII"?>
<trace:Trace xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:trace="trace">
<state newobjects="//@transientobject.0 //@transientobject.1 //@transientobject.2">
  <transientObjectsStates transientobject="//@transientobject.0" objectstate="//@objectstate.4"/>
  <transientObjectsStates transientobject="//@transientobject.1" objectstate="//@objectstate.5"/>
  <transientObjectsStates transientobject="//@transientobject.2" objectstate="//@objectstate.6"/>
</state>
<state basestate="//@state.0">
  <transientObjectsStates transientobject="//@transientobject.0" objectstate="//@objectstate.7"/>
  <transientObjectsStates transientobject="//@transientobject.2" objectstate="//@objectstate.8"/>
</state>
<transientobject xsi:type="trace:StaticTransientObject">
  <originalObject href="platform:/resource/exampleproj/src/models/TFSM_simpleTrafficLight.xmi#//@tfsms.0"/>
</transientobject>
<transientobject xsi:type="trace:StaticTransientObject">
  <originalObject href="platform:/resource/exampleproj/src/models/TFSM_simpleTrafficLight.xmi#//@tfsms.0/@localEvents.0"/>
</transientobject>
<transientobject xsi:type="trace:StaticTransientObject">
  <originalObject href="platform:/resource/exampleproj/src/models/TFSM_simpleTrafficLight.xmi#//@tfsms.0/@localClock"/>
</transientobject>
<steptype stepName="tfsm.TFSM"/>
<steptype stepName="tfsm.FSMClock"/>
<steptype stepName="tfsm.State"/>
<steptype stepName="tfsm.Transition"/>
<steptype stepName="tfsm.FSMEvent"/>
<steppattern repeatingstep="//@repeatingstep.0 //@repeatingstep.1"/>
<value xsi:type="trace:RefValue"/>
<value xsi:type="trace:LiteralInteger">
  <intvalue>0</intvalue>
</value>
```

Figure A.5: An excerpt of the trace of an fUML model serialized in XML

Figure A.6: An excerpt of the trace of an fUML model serialized in EXI

List of Acronyms

CTM Compact Trace Metamodel

DSML Domail Specific Modeling Language

EMF Eclipse Modeling Framework

EXI Efficient XML Interchange

xDSML Executable Domail Specific Modeling Language

XML Extensible Markup Language

fUML Foundational UML

GPML General Propose Modeling Language

JSON JavaScript Object Notation

MDD Model Driven Development

MDE Model Driven Engineering

OMG Object Management Group

UML Unified Modeling Language

V&V Verification and Validation

XMI XML Metadata Interchange

xMOF eXecutable MOF