Noname manuscript No.
(will be inserted by the editor)

Lossless Compaction of Model Execution Traces

Fazilat Hojaji · Bahman Zamani · Abdelwahab Hamou-Lhadj · Tanja Mayerhofer · Erwan Bousse

Received: date / Accepted: date

Abstract Dynamic Verification and Validation (V&V) techniques are used to verify and validate the behavior of software systems early in the development process. In the context of model-driven engineering, such behaviors are usually defined using Executable Domain-Specific Modeling Languages (xDSML). Many V&V techniques rely on *execution traces* to represent and analyze the behavior of executable models. Traces, however, tend to be overwhelmingly large, hindering effective and efficient analysis of their content. While there exist several *trace metamodels* to represent execution traces, most of them suffer from scalability problems. In this paper, we present a generic compact trace representation format called generic Compact Trace Metamodel (CTM) that enables the construction and manipulation of compact execution traces of executable models. CTM is generic in the sense that it supports a wide range of xDSMLs. We evaluate CTM on traces obtained from real-world fUML models. Compared to existing trace metamodels, the results show a significant reduction in memory and disk consumption. Moreover, CTM offers a common structure with the aim to facilitate interoperability between existing trace analysis tools.

Fazilat Hojaji
E-mail: f.hojaji@eng.ui.ac.ir

Bahman Zamani
E-mail: zamani@eng.ui.ac.ir

Abdelwahab Hamou-Lhadj
E-mail: wahab.hamou-lhadj@concordia.ca

Tanja Mayerhofer
E-mail: mayerhofer@big.tuwien.ac.at
https://big.tuwien.ac.at/people/tmayerhofer/

Erwan Bousse
E-mail: bousse@big.tuwien.ac.at
https://big.tuwien.ac.at/people/ebousse/

Keywords Execution Trace · Compaction · Model Execution · Trace Metamodel

1 Introduction

Model Driven Engineering (MDE) is a software development paradigm that aims to decrease the complexity of software development by raising the level of abstraction through the use of models and well-defined modeling languages [1]. For this purpose, two main types of modeling languages are used: General Purpose Modeling Languages (GPMLs), such as UML, for modeling systems regardless of the domain, and Domain Specific Modeling Languages (DSMLs) that are designed for specific tasks in a given domain [2]. Also, it can be distinguished between structural models (e.g., UML class diagrams) to model a system's structure, and behavioral models (e.g., UML activity diagrams) to model the behavior of a system.

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, such as methods to ease the development of executable DSMLs (xDSMLs) [3, 4, 5, 6, 7], or to support the execution of UML models [8]. In addition, many V&V techniques require the capability to capture and manipulate information about an execution in the form of a trace. For instance, model checking techniques [4, 9, 10, 11] check whether execution traces satisfy predefined temporal properties, and rely on execution traces also for representing counter examples. Omniscient debugging [12, 13, 14] utilizes execution traces to go back in the execution and revisit

previous states. Semantic differencing [15, 16] identifies the semantic variations between two models by comparing their execution traces.

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 [17, 18, 19].

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 to implement their semantics. 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. [20] and Langer et al. [15]. 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. [15] 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. [21], 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 inputs and outputs values, hindering expressiveness.

In this paper, we provide a generic, scalable trace metamodel that can be used for any xDSML and supports the representation of traces in a compact form. This is achieved through the following contributions:

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

We provided an EMF-based implementation of CTM that can be installed in the Eclipse GEMOC Studio[1], a language and modeling workbench based on Eclipse. To evaluate the genericity of CTM, we successfully applied it to capture execution traces for models of five different xDSMLs. We also evaluated the scalability of CTM with regard to memory consumption and disk space by comparing CTM traces to those modeled using the metamodel proposed by Bousse et al. [21, 22]. The experiments show that our approach has a small overhead for constructing traces during model execution, while reducing memory consumption and disk space of traces. Using CTM, we can reach an average compaction rate of 59% in memory usage and 95% in disk space. Besides, we provide a mechanism to transform compacted traces into their original format, demonstrating the fact that CTM preserves the information contained in traces.

Our research methodology relies on the *Design Science Research Methodology (DSRM)* presented by Peffers et al. [23] aligned with the guidelines for design science defined by Hevner et al. [24]. The DSRM approach consists of the following steps: 1) Problem identification and motivation, 2) Definition of objectives for designing a trace structure, 3) Design and development of the trace structure, 4) Demonstration, 5) Evaluation and 6) Communication of the trace structure. At least one iteration was performed in each step of the process, which we present in each section of this paper in detail. From a top-level methodological perspective, we resorted to different research techniques in each step and performed activities to appropriately support our overall objectives. The practical relevance and importance of the research problem has been well demonstrated as being critical for design science research [24]. To ensure that our design objectives are consistent with prior research, we conducted a systematic mapping study in the field of model tracing, which has already been published in [25]. We aim to further disseminate the contributions of this effort in peer review scholarly publications.

The remainder of the paper is structured as follows. In Section 2, we provide a background around

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

model tracing, and make an overview of trace compaction techniques and data serialization formats as well. In Section 3, we motivate the problem domain and describe our ideas for overcoming existing limitations. Section 4 gives an overview of our approach. In Section 5, we discuss the design of CTM with respect to the defined requirements for the development of a compact trace metamodel. Section 6 presents a detailed implementation of CTM-enabling tools within the language and modeling workbench GEMOC. Section 7 shows the evaluation of CTM. Section 8 discusses related work. Section 9 summarizes the contributions of this paper, and Section 10 provides an outlook on future work.

2 Background

In this section, we first define all concepts important in this work, such as *executable model* and *xDSML*, *execution state*, *execution step*, and *execution trace*. We then give an overview of the most common techniques used for trace compaction. Finally, we discuss popular formats used for data serialization.

2.1 Model Execution

In the following, we first define the terms *executable model* and *xDSML*, then give an example of an xDSML. Some of these definitions are based on the one's proposed by Bousse et al. [21].

Definition 1 An *executable model* is a model conforming to an executable modeling language and defines an aspect of the behavior of a system in sufficient detail such that the model can be executed.

Definition 2 An xDSML is defined by:

- An *abstract syntax*, i.e., a metamodel.
- An *execution metamodel*, which is an extension of the abstract syntax with additional classes and properties defining the dynamic state of a model.
- An *operational semantics*, which includes an execution transformation that modifies a model that conforms to the execution metamodel by changing the values of dynamic fields, and by creating/destroying instances of classes of the execution metamodel.
- An *initialization function*, which is an in-place model transformation that transforms a model conforming to the abstract syntax into a model conforming to the execution metamodel.

In the MDE community, a wide range of different xDSMLs have been developed, and are used to express the behavior of systems. Examples of rather well-known xDSML include Petri nets [26], fUML [27], BPMN [28], live sequence charts [29], or story diagrams [30].

Fig. 1 shows an example of a Petri nets xDSML. The abstract syntax contains three classes: **Net**, **Place**, and **Transition**. The top right of the figure shows the execution metamodel, which extends the class **Place** with a new property using *package merge*. The new property `tokens` defines the number of tokens of a **Place** object during execution. The initialization function (not shown) creates executable objects (i.e., a **Place** object with a `tokens` field) as defined in the execution metamodel, and initializes each `tokens` field with the value of `initialTokens`. 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**.

2.2 Model Execution Traces

In this subsection, we first define the terms *execution state*, *execution step*, and *execution trace*, then provide an example of an execution trace obtained by executing a Petri net model. Once again, note that part of these definitions is based on the ones previously proposed by Bousse et al. [21].

Definition 3 An *execution state* refers to the set of values of all dynamic properties of an executed model at a given point of an execution.

Definition 4 An *execution step* is the set of changes applied to the execution state of a model that is obtained by the application of a model transformation rule. An execution step may be composed of several execution steps, organized in a hierarchical structure.

As an example, for the Petri nets xDSML shown in Fig. 1, each application of the execution rules *run* and *fire* on a Petri net results in an execution step. Since the rule *run* repeatedly calls the rule *fire* for each enabled transition, the execution step created for the application of the *run* rule will be composed of the execution steps created for the applications of the *fire* rule. The execution state of the model changes each time *fire* is executed. In particular, the `tokens` field of several places will change when the *fire* rule is applied, hence changing the executions of the model.

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

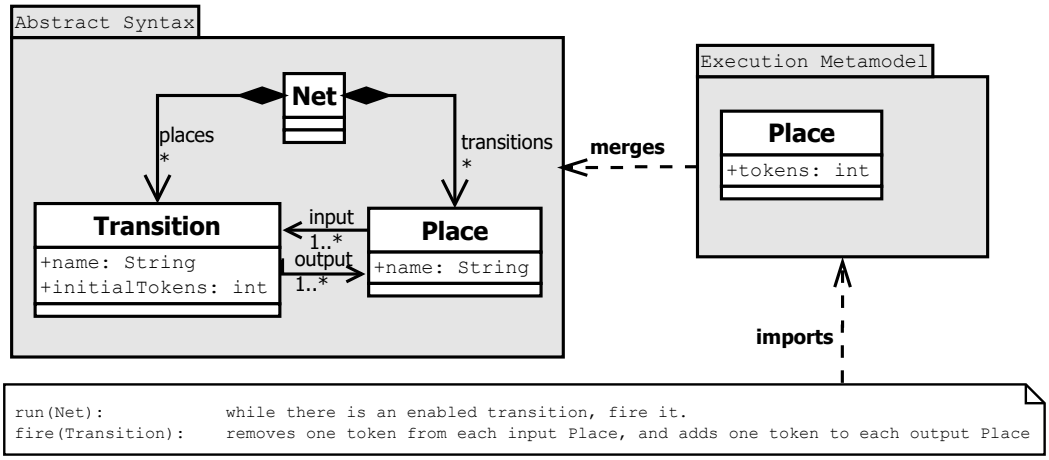


Fig. 1: Petri nets xDSML [21]

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 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.

Definition 5 A *model execution trace* captures information about the execution of an executable model. This information may include a sequence of execution states, execution steps, the state of objects during execution, the processed input parameters, and the produced output parameters.

Fig. 2 illustrates an execution trace of a sample Petri net model that is executed using the operational semantics of the xDSML shown in Fig. 1. We use the concrete syntax of Petri nets to show the execution. In this example, Transition *t1* is fired three times, producing three steps. Thus, these steps are recorded for the application of *fire* on the Transition *t1*. Furthermore, for actually starting the execution of the Petri net, the rule *run* is applied once on the **Net** object representing the complete Petri net. The execution step created for this application of *run* thus is composed of the three execution steps created for the applications of *fire*. For each **Place**, we recorded its execution state, i.e., the value of the `tokens` field after each execution step. Moreover, the whole Petri net model is considered as the input parameter of the *run* step, and Transition *t1* as the input parameter for the *fire* steps.

2.3 Trace Compaction Techniques

Compaction techniques are required to reduce the size of execution traces. Many compaction approaches have been proposed for tackling the large volume of traces (e.g., [32, 33, 34, 35, 36]). However, most of these techniques have only been applied to code-centric approaches. Their effectiveness, when applied to executable models, has yet to be shown. We categorize the commonly used trace compaction techniques into different groups and summarize them in the following.

Table 1 provides an overview of the techniques that are used for trace compaction. The first column of the table presents the name of the technique and the second column refers to the application domain. The third column shows the technique that we used in our approach.

Trace filtering [19, 32] refers to a set of techniques that consider a partial trace instead of the whole trace by sampling its content, removing specific components, grouping similar parts of trace (pattern matching), etc.

Graph reduction treats traces as graphs and applies graph theory to transform them into more compact forms. For example, Hamou-Lhadj et al. [19] proposed a graph transformation technique in which a trace of routine calls, represented as a tree structure, is converted into an ordered directed acyclic graph (DAG) by representing similar sub-trees only once. The idea is removing repetitions by collapsing them into one node, resulting in a significant size reduction.

Dynamic slicing [33, 37] refers to a set of techniques in which a trace is divided into different parts using slicing algorithms by identifying several types of dependencies in traces. All unnecessary parts are removed from the trace, which can further reduce its size.

Recording the modifications of the dynamic model [38] is an approach in which, instead of stor-

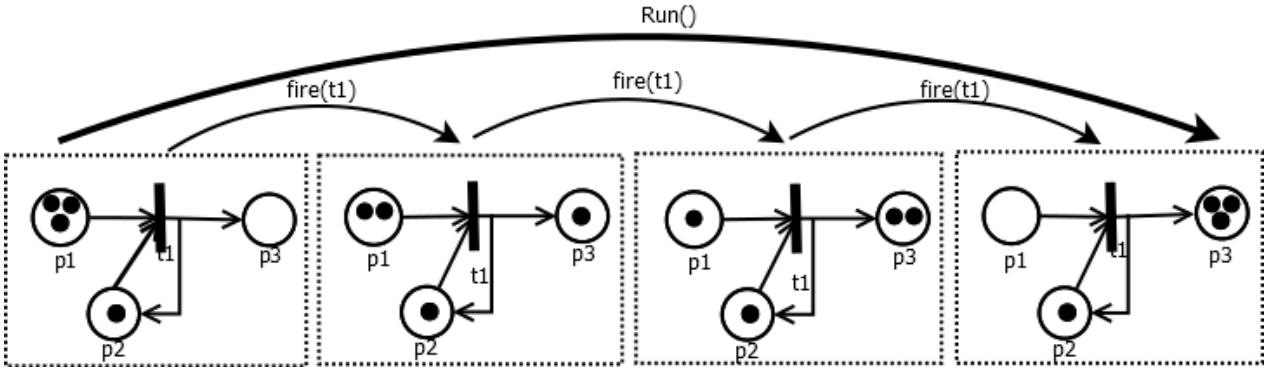


Fig. 2: Example of a Petri net execution trace shown using concrete syntax

Table 1: Common techniques used for trace compaction

Name	Technical space	Applied in our approach
Trace filtering [19, 32]	code-centric	
Graph reduction [19]	code-centric	*
Dynamic slicing [33, 37]	code-centric	
Recording the modifications of the dynamic model [38]	MDD	*
Sharing immutable objects [21]	MDD	*
C-store [39]	data management	
RainStor [40]	data management	*

ing all the dynamic information of the new state of the model, only the modification (delta) between two subsequent states is represented.

Sharing immutable objects [21] is a mechanism to avoid duplicating immutable runtime objects, i.e., objects that cannot be changed during execution. These objects can be shared between the original model and the trace representation.

C-Store [39] is a column-oriented database management system that stores data column wise-instead of row-wise. Each column in C-Store is compressed, and for each column, a different compression method may be used.

RainStor [40] is a column storage technique for storing data. Every unique value in the dataset is stored only once, and every record is represented as a binary tree that allows reconstructing the original record using a breadth-first traversal of the tree. RainStor provides a compression algorithm by creating a network, or graph of values, and storing every value in a database only once. The algorithm yields a data reduction rate of 40:1, i.e., it requires 40 times less storage.

2.4 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) [41] 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 [42] 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.

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

Efficient XML Interchange (EXI) [43] 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) [44] 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) [45] 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.

3 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.

3.1 Requirements for an execution trace metamodel

The results of our systematic survey [25] on model execution tracing approaches show that there has been an increasing interest in this domain in recent years. However, there exist several 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. In addition, our survey shows that existing model tracing techniques use large amount of information about the execution of the models. In fact, nearly half of the surveyed techniques keep detailed execution state information. Also, the generated traces depend on the execution scenarios used to exercise the systems. Complex execution scenarios are expected to result in very large traces. 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 to implement their semantics.

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.

3.2 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 [46], a text format as proposed by Maoz et al. [47, 48], or metamodels such as the one's proposed by Hegedus et al. [38]. However, the results of our survey on trace representation formats [25] 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., [20], [15]) 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 [25], 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. [10], 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 [38] to reduce traced state information by only capturing state modifications and events related to state modifications. Similarly, Bousse et al. [21] 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 [46] 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 paper 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 compaction techniques.

4 Approach Overview

To overcome the limitations observed in existing trace formats, and to better comply with the requirements mentioned in section 3.1, 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.

Fig. 3 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. 5.2. 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 decompaction 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 Generic Compact Trace Metamodel (CTM)

This section explains a two-step process for designing CTM with the aim of supporting the genericity and scalability criteria described in Sect. 3. 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

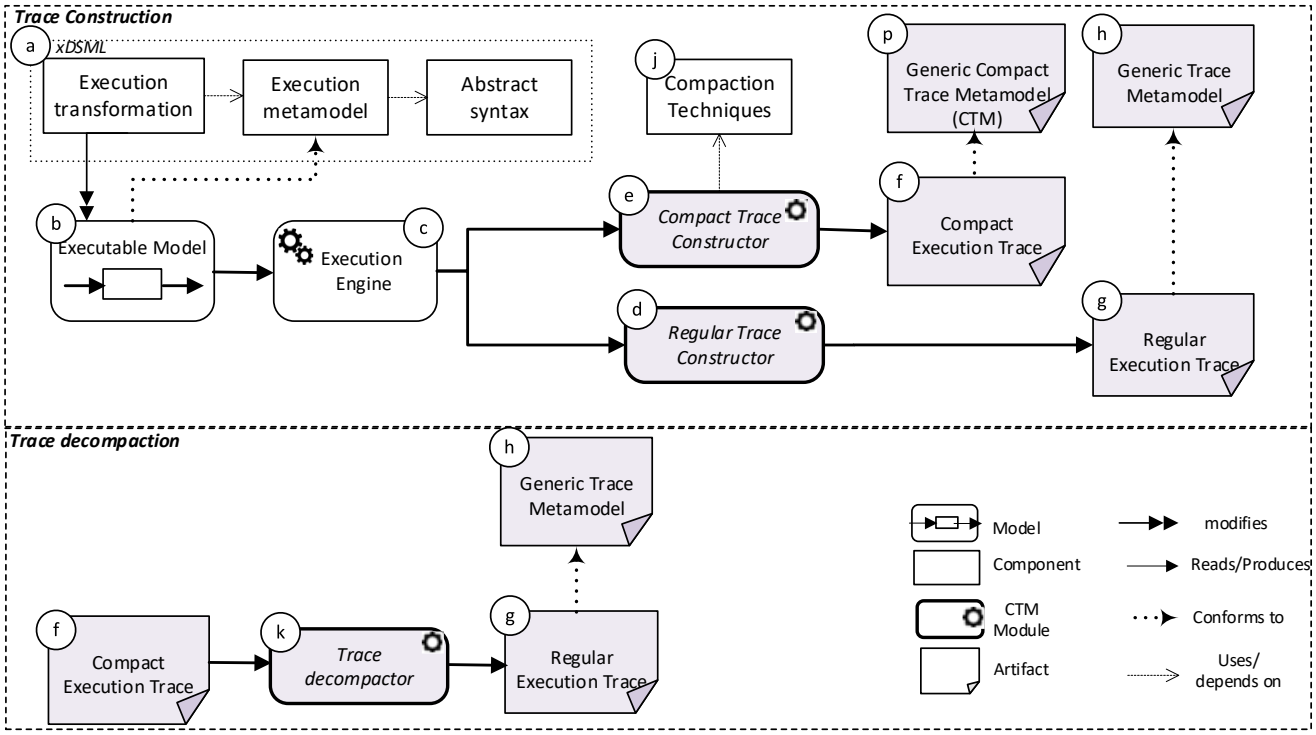


Fig. 3: Approach overview, with our contributions highlighted in gray

is explained in Sect. 5.1. 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 Sect. 5.2.

5.1 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.

Fig. 4 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 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. There is the reference `estructuralfeatures` from **TransientObject** to Ecore's metaclass **EStructuralFeature** to represent the name of the fields corresponding to each object. Also, **EClass** owns a set of **EStructuralFeature**, each being either **EReference** or **EAttribute**.

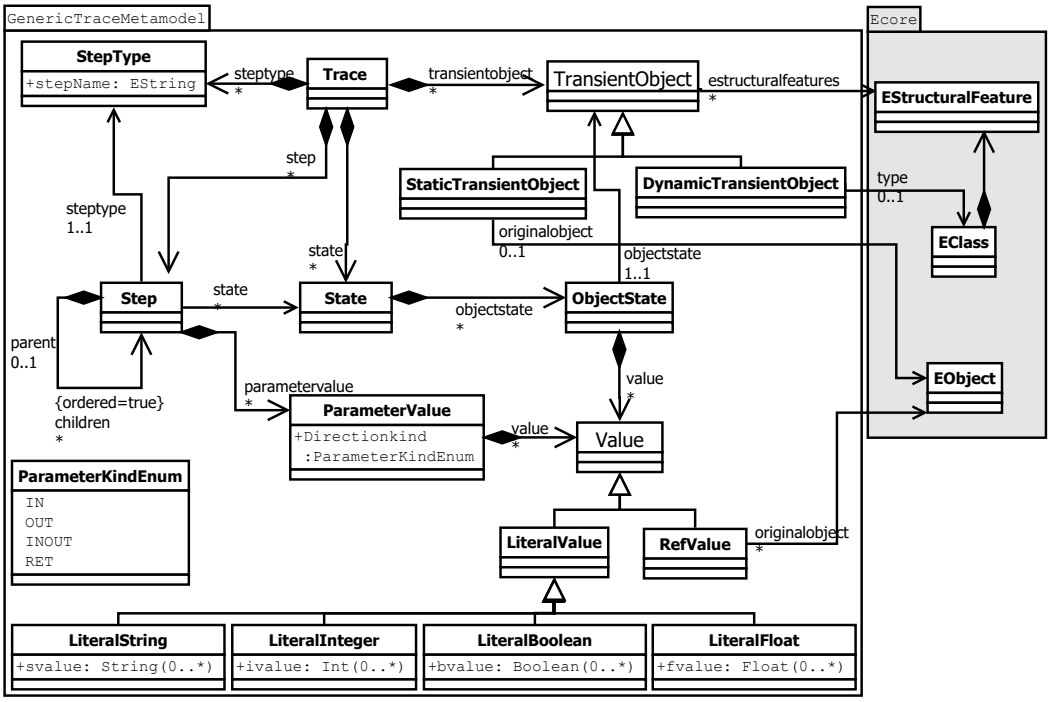


Fig. 4: The proposed generic trace metamodel

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.

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.

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 Fig. 2. Fig. 5 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*.

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 Fig. 5, there exist many repetitions in the trace, particularly in the **ObjectState**, **State**,

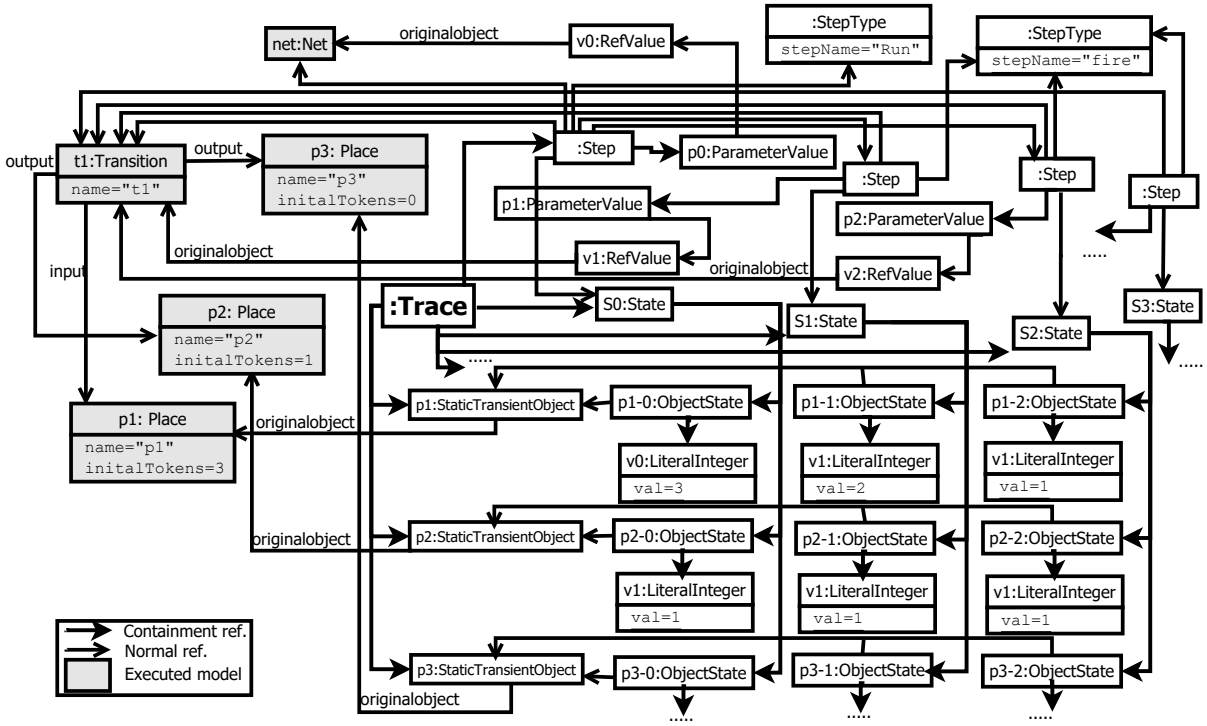


Fig. 5: Excerpt of execution trace of the Petri net example that conforms to the proposed generic trace metamodel

ParameterValue and **Value**. Additionally, there are repetitions of **Steps** due to the existence of a loop in the model, causing the Transition t1 to be fired three times. In the next subsection, we explain the techniques applied to the generic trace metamodel, with the aim of eliminating the repetitions. Although, applying the compaction techniques results in adding new concepts to CTM, making execution traces a bit more complex, our evaluation results in Sec. 7 show that CTM meets the scalability requirement.

5.2 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.

*5.2.1 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 (Fig. 5), 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 Fig. 5, this means storing only the **ObjectState** for *p1* and *p3* at each execution step.

Fig. 6 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 Fig. 4) 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 **TransientOb-**

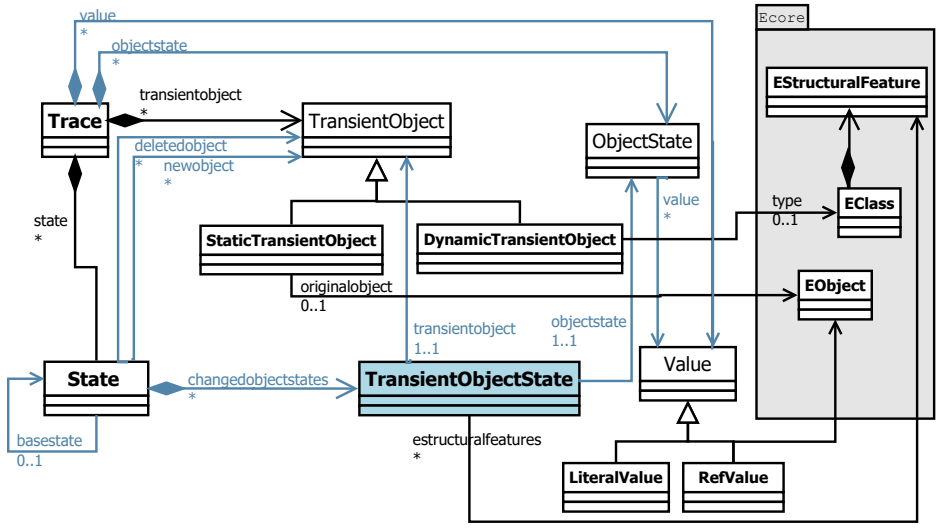


Fig. 6: Excerpt of CTM modeling concepts related to **State** with the changes highlighted in blue

ject. More precisely, instead of having a reference from the class **ObjectState** to the class **TransientObject**, the class **TransientObjectState** defines the relationship between these two classes. Such a structure allows using an **ObjectState** object for different **TransientObject** objects. The **TransientObjectState** objects are only created for the **TransientObject** objects that have changed in the current state. 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).

Following a structure similar to Fig. 4, **StaticTransientObject** and **DynamicTransientObject** are derived from **TransientObject**, and linked to **EObject** and **EClass**, respectively. Likewise, the reference `estructuralfeatures` from **TransientObjectState** to Ecore's metaclass **EStructuralFeature** are used to represent the name of the fields corresponding to each 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.

For the compaction of **States**, we used a notification framework implemented in GEMOC Studio 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, compares the **ObjectState**s and **Value**s contained in the chosen **State** with those of the current **State**, and find the closet **State** object. In addition to considering state modifications after executing steps, our trace constructor also supports recording of state modifications that occurs before starting a step. This is relevant when a step makes a change before calling another (sub-)step. In such a case, the *trace constructor* creates an instance of **Step** object and its **StepType** is assigned to *"Implicit step"*.

Fig. 7 shows an excerpt of the trace of the Petri net example model, conforming to the part of CTM shown in Fig. 6. The trace illustrated in Fig. 7 is a compact

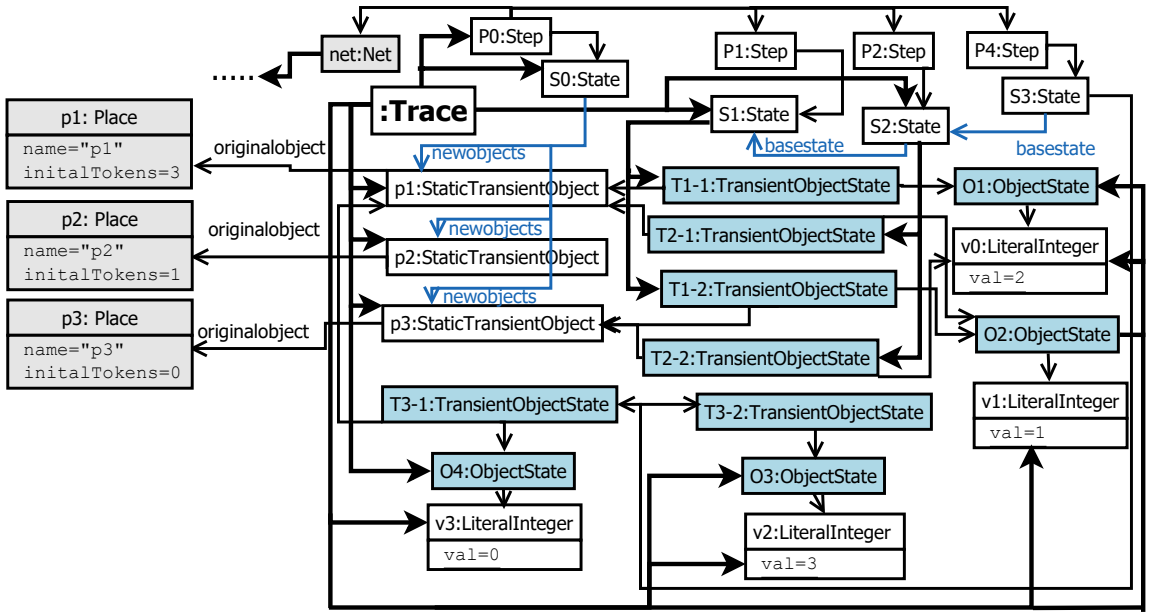


Fig. 7: Excerpt of execution trace of the Petri net example (**State** with compaction)

version of the trace that was presented in Fig. 5. 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 Fig. 5, the first state is represented by using four `newobjects` references, and no **ObjectState** objects are created. 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.

5.2.2 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 Fig. 5, there are three repetitions, caused when the transition *t1* is fired.

To achieve this, we adopted the *Flyweight design pattern* [49] and the *Composite design pattern* [49] 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:

StepPattern: 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 Fig. 4) 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.

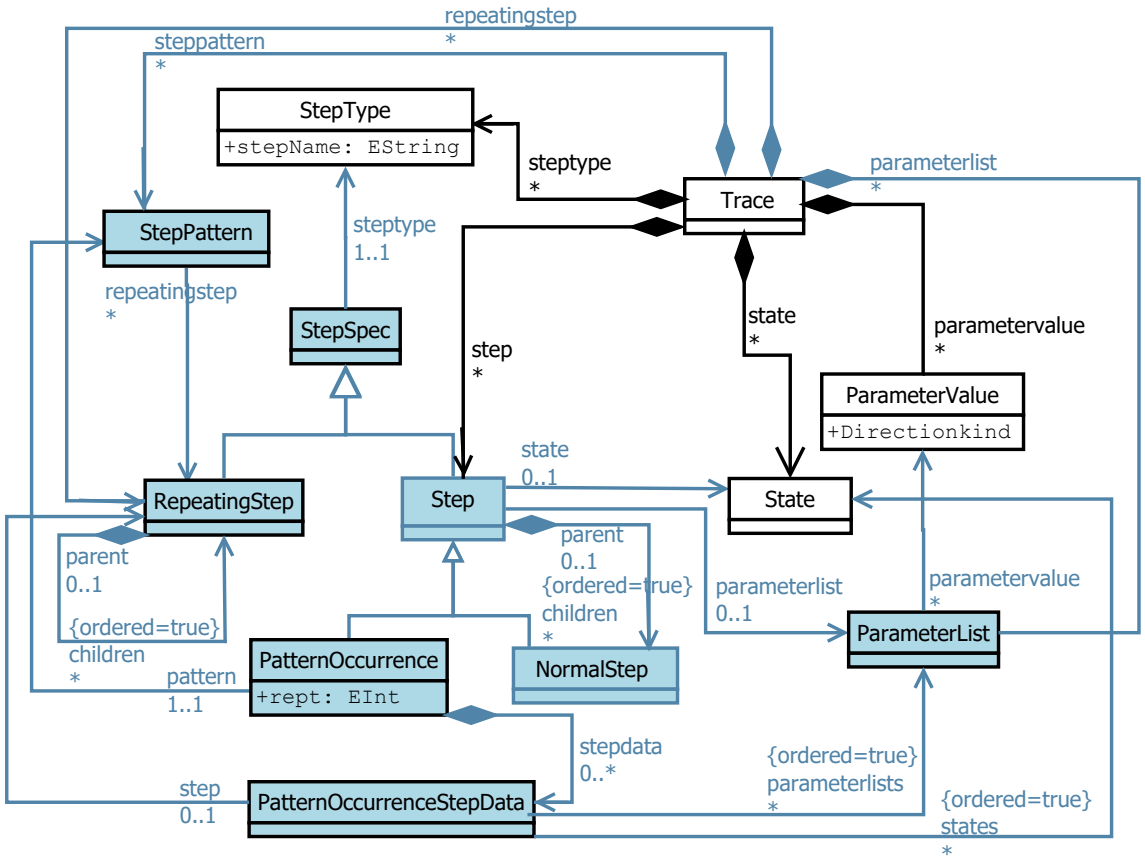


Fig. 8: Excerpt of CTM with modeling concepts related to **Step**, with the changes highlighted in blue

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 Fig. 8, 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 the occurrence of the patterns and contains an attribute `rept` that is used to specify the number of repetitions of a pattern.

Each **PatternOccurrence** object is related to a **StepPattern** object using the reference `pattern`. 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

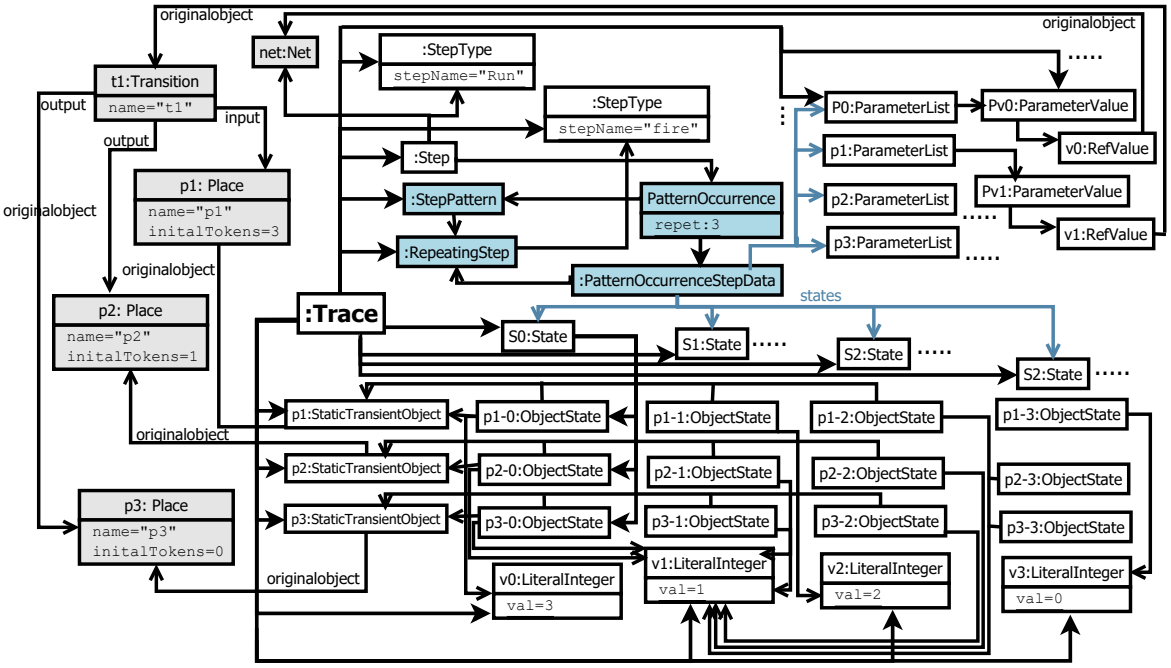


Fig. 9: Excerpt of an execution trace of the Petri net example model including a loop (**Step** with compaction)

ParameterList objects of **Step** objects included in a loop. Similar to the technique used by Taniguchi et al. [50] 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 can 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.

From a technical point of view, we used an extension of Valiente's algorithm [51] proposed by Hamou-Lhadj and Lethbridge [19] to detect redundant patterns within execution steps. The idea behind this algorithm is converting a tree structure into a more compact ordered directed acyclic graph. In our case, the execution trace would be a tree, containing the nodes corresponding to the **Step**s. The aim is to detect patterns involving consecutive repetitions of **Step**s existing due to loops or recursion. The algorithm simply takes a complete execution trace and produces a *certificate* and *signature* for each node by traversing the tree of **Step**s in a bottom-up fashion (from the leaves to the root). The *certificates* (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 **StepType** of the respective **Step** and the *certificates* of its direct children. To carry out this work, we extended the **Step** class by adding two new properties (*signature*, *certificate*) in the trace metamodel. Given a *signature*, we can recognize repetitions that might be included in the corresponding **Step**, and thus construct the trace, which conforms to the trace metamodel shown in Fig. 8. Note that the technique of step compaction is done *offline* (i.e., after the execution of the model), due to the complexity of applying graph reduction technique during execution. We defer doing the step compaction *on the fly* to future work.

Fig. 9 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 the **StepPattern** object, and stores the value three 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 (Fig. 5), the resulting compact trace requires only five objects and six references to represent the step part of the trace as opposed to four objects and 12 references when compaction is not used.

*5.2.3 Dealing with repetitive values 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.

As an example, we use a trace of a colored Petri net, shown in Fig. 10. 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 2 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 fourth 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.

Table 2: Excerpt of **ObjectState** data for the **Place** objects captured during the execution of the colored Petri net model shown in Fig. 10

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

At the bottom of Fig. 10, 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 Fig. 10, e.g., the **IntegerValue** objects with "zero" value and the links between **State** and **StaticTransientObject**. Moreover, the figure does not show **TransientObjectState** objects and their links to **LeafObjectState**, **CompositeObjectState** and **StaticTransientObject** either. 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.

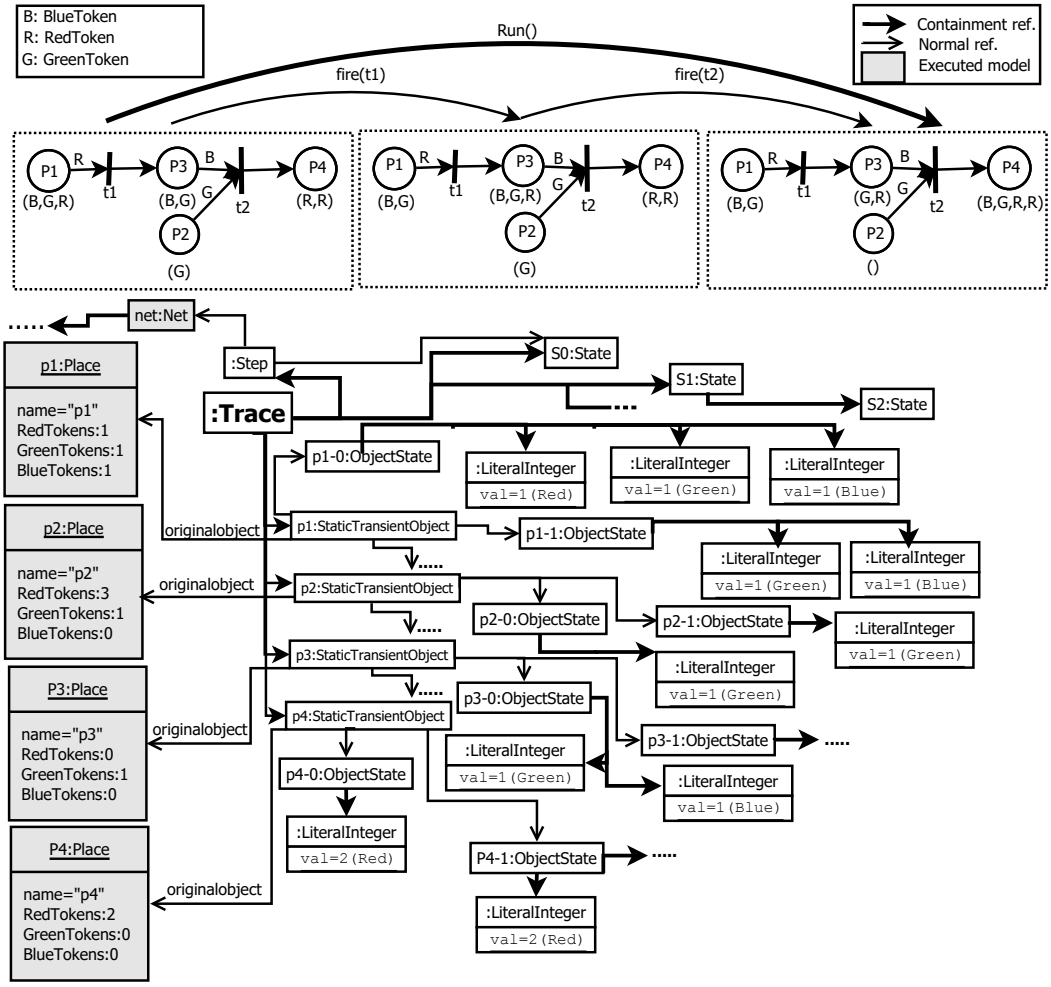


Fig. 10: Excerpt of execution trace of the colored Petri net example model (**ObjectState** without compaction)

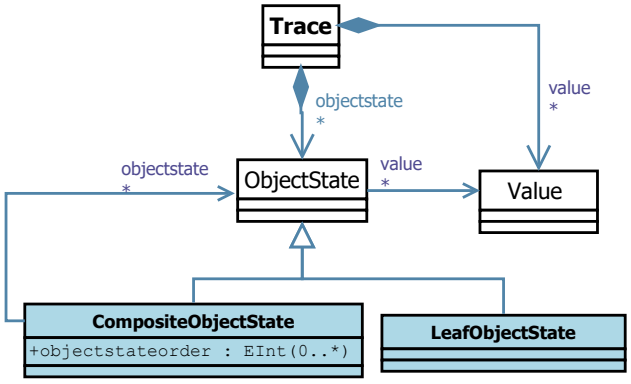


Fig. 11: Excerpt of CTM with modeling concepts related to **ObjectState**, with the changes highlighted in blue

Fig. 11 shows our solution for improving CTM generic metamodel by compacting repetitive values of **ObjectState** objects. Our solution was inspired by the Rainstore approach [40], introduced in Sect. 2.3. As explained in Sect. 2.3, 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* [49]. 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.

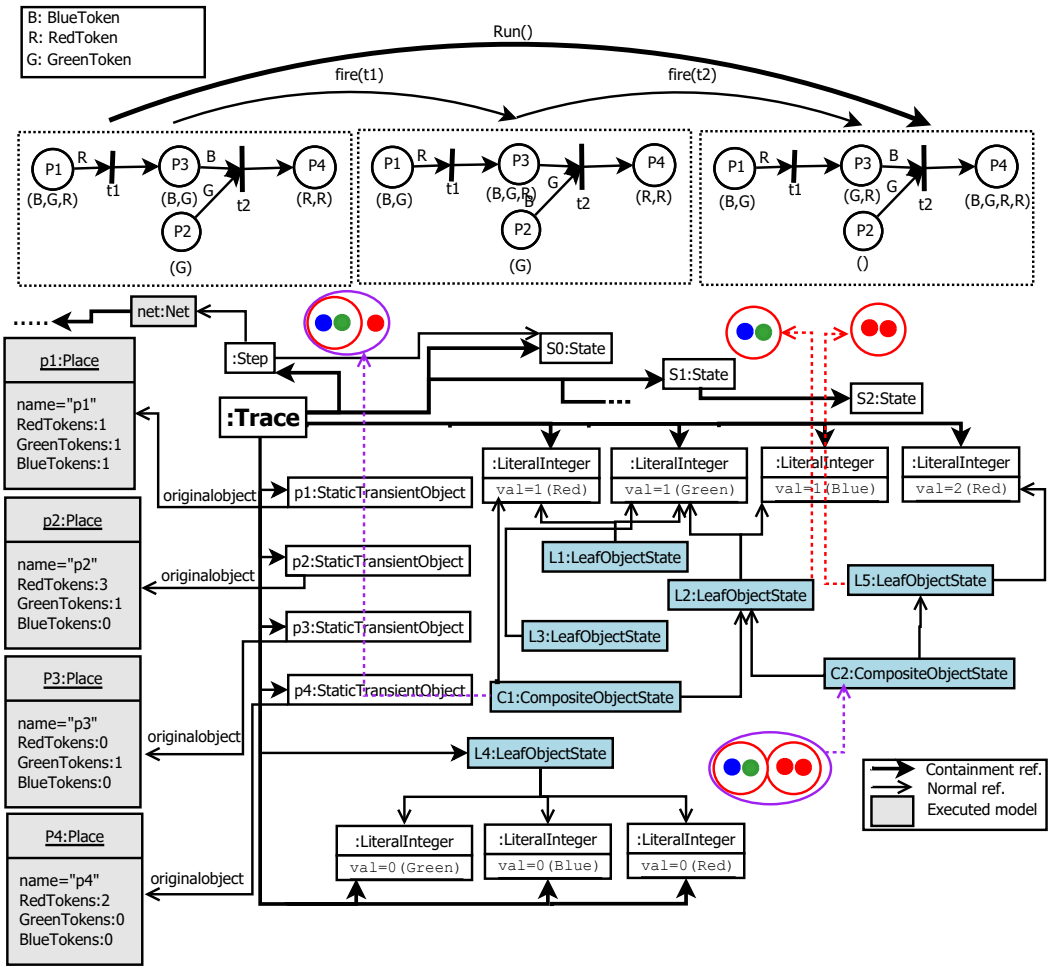


Fig. 12: Excerpt of an execution trace of the colored Petri net example model (**ObjectState** with compaction)

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 **CompositeObjectState**. 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.

For the implementation of the **ObjectState** compaction, the main challenge was how to efficiently identify redundant values of **ObjectState**s that can be shared among different **ObjectState**s. In this work, we used LCM (Linear time Closed item set Miner) [52], a powerful algorithm for enumerating frequent closed item sets, which creates a set of **ObjectState**s includ-

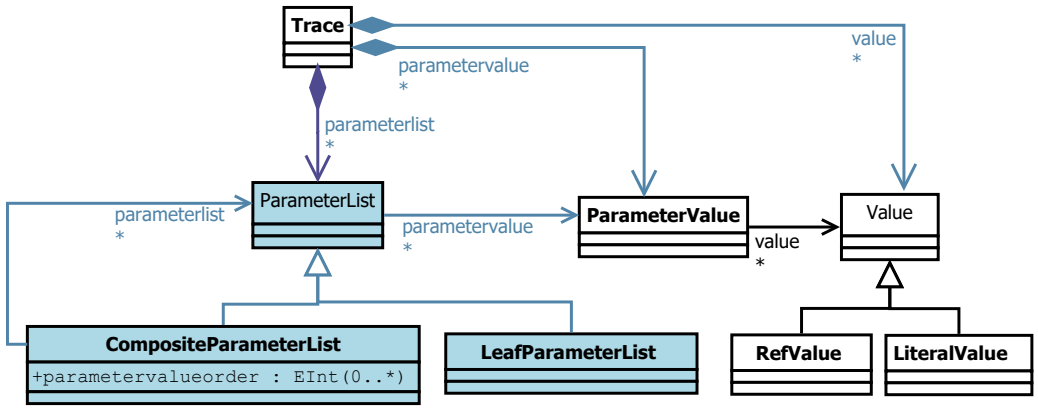


Fig. 13: Excerpt of CTM with modeling concepts related to **ParameterList**, with the changes highlighted in blue

ing 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 by executing several fUML models multiple times and selected the value that leads to the least memory consumption during the execution. The compaction can be done both *on the fly* and *offline*.

Fig. 12 shows the compact version of the trace of the colored Petri net example model from Fig. 10. We can see that in the last execution state, *p4* holds one Blue token, one Green token, and two Red tokens. Therefore, the pattern of one Blue token and one Green token can be observed multiple times in the Petri nets execution (see Table 2). 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 **ObjectStates** of **Place** objects where the **Place** objects hold one Green and one Blue token. To illustrate this, Fig. 12 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. Also, it refers to a **LiteralInteger** that defines one Red token.

In comparison to the original trace, shown in Fig. 10, 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.

*5.2.4 Dealing with repetitive values in **ParameterList***

The last part of our compaction strategy deals with redundancies among input and output parameters of **Step** objects concerning 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 Sect. 5.2.3 for **ObjectState**. Due to space restrictions, we only present the relevant part of CTM in this section.

As shown in Fig. 13, 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.

From an implementation point of view, we used the same algorithm as ObjectState compaction for finding frequent **ParameterValue** objects within **ParameterList** objects, and consequently creating **CompositeParameterList** and **LeafParameterList**. Note that,

the compaction can be done both *on the fly* and *offline*.

6 Implementation

This section presents the implementation of our work within the language and modeling workbench Eclipse GEMOC Studio. We first give an overview of Eclipse GEMOC Studio and its execution framework. Then, we present the implementation of CTM in GEMOC.

6.1 Eclipse GEMOC Studio

Eclipse GEMOC Studio is a framework for designing, integrating EMF-based modeling languages. The framework offers two workbenches: a language workbench and a modeling workbench. The language workbench is used by language designers and language integrators to build and compose new xDSMLs. The designer defines the abstract syntax, the concrete syntax, and the operational semantics of the xDSML. The operational semantics is defined by using different meta-programming approaches, such as Java-based languages (Kermeta [53], Xtend, pure Java) and xMOF [5]. The modeling workbench is used by domain designers to create, execute, and coordinate models conforming to xDSMLs developed with the language workbench.

Eclipse GEMOC Studio offers an execution framework [7] that provides a generic interface allowing to integrate different execution engines. Using this interface, execution engines for different meta-programming languages have been added to Eclipse GEMOC Studio. One functionality offered by the interface is to have so-called *engine add-ons*, that are add-ons registered at execution engines, which will get notified about the progress of carrying out model executions (e.g., the beginning of the execution and the completion of execution steps).

6.2 Implementing the *CTM add-on* for Eclipse GEMOC Studio

We provided an EMF-based implementation of our work consisting of a set of Eclipse plugins. We chose EMF since GEMOC Studio is based on EMF, and also we had to extend EMF libraries—including implementations of EObject and EClass—to get the runtime objects in the model. Despite this limitation, the data structure and the proposed compaction techniques can be implemented with other technologies. Concepts of data classes, attributes and references exist anywhere.

The plug-ins contain both generic and compact trace metamodels and the *trace constructor* that creates traces conforming to CTM during model execution, as well as the *trace de-compactor* that translates compacted traces into their original uncompacted traces. We explain and discuss the different parts of the implementation in the following sections.

1) Implementation of the trace metamodels. We implemented CTM using the Eclipse Modeling Framework. In particular, we employed the metamodeling language Ecore for defining the abstract syntax of the metamodels (the generic trace metamodel and the generic compact trace metamodel), and Kermeta [53] for implementing their operational semantics.

The output are plugins that are generated within the language workbench, and are automatically deployed into the modeling workbench.

2) Implementation of the trace constructor. The main element of our implementation is the *trace constructor* (shown by *d* and *e* elements of Fig. 3), which is used for constructing traces during a model execution. It contains two components: 1) the *regular trace constructor*, which constructs traces in a regular form conforming to the generic trace metamodel introduced in Sec. 5.1, 2) the *compact trace constructor* which constructs traces in a compact form conforming to CTM. 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*.

As shown in Fig. 3, the input of both trace constructors is an xDSML, which is defined using Ecore for the abstract syntax, and using either Kermeta [53] or xMOF [5] for the operational semantics, and an input model for the execution as well. The output of the trace constructors is an execution trace either in an original or compact form.

In the case of generating the compact trace, we have applied several compaction techniques, each corresponding to one of the compactions presented in Sec. 5.

Note that the generated trace in both cases, regular or compact form, is serialized in two different serialization formats (XMI and EXI), and is stored to disk. We chose EXI because it greatly outperforms other schemes [54], and allows the sharing of data in a convenient and flexible way.

3)Implementation of the trace de-compactor. Another plugin is the *trace de-compactor*, which takes a serialized compact trace as input, and produces its uncompacted version. We used the trace de-compactor to show that the compact trace constructor compacts traces without the loss of information. For this, we compared the traces produced with the regular trace constructor with original traces that were reproduced by the *trace de-compactor*. 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.

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 code, respectively. The source code (EPL 1.0 licensed) is available at our project web page[4].

7 Evaluation

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 meta-programming 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.

7.1 fUML

Foundational UML (fUML) [27] 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 explaining the behavior of these classes using UML activities. The fUML execution model is a model that describes 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 produces 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.

7.2 Experiments

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

7.2.1 Genericity

To evaluate the genericity of 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 meta-programming approaches?

To answer these questions, we tested CTM with a selection of different xDSMLs that had previously been developed using Eclipse GEMOC Studio. Table 3 presents all the xDSMLs considered in this study, with links to their source material. The *trace constructor* and the *trace de-compactor* were successfully tested for these languages. For each xDSML, we executed several

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

Table 3: xDSMLs applied to test our prototype

xDSML	Link	Description	Semantics
Petri nets	link[a]	Simple Petri nets (see Fig. 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

example models with different parameters and generated execution traces using the *trace constructor*. As we explained in Sec. 6, 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. [21], for the number of **Step**, **State** and **Value** objects, and observed the same results. We also reconstructed the uncompacted version of the compact traces using the *trace de-compactor*, compared them with the uncompacted ones produced by the regular trace constructor and observed similar results.

7.2.2 Scalability

To evaluate the 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. [21].

To proceed, first, we chose the set of fUML models, which have been selected by Maoz et al. [16] from different industrial sources (e.g., IBM, Nokia), and have been already used for similar case studies (e.g., [14], [21]). We chose these fUML models because they have also been used by Bousse et al. [21] for the generation of traces. Note that the experiments contain a total of 38 model executions of the considered fUML models, with the number of execution states ranging between 180 and 340, and different parameter settings. 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 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 and EXI 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:

$$A = Nref_{CTM} + Nobj_{CTM} \tag{1}$$

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'.

$$B = Nref_{DS} + Nobj_{DS} \tag{2}$$

We measure the memory compaction rate as follows:

$$CR_{memory} = (1 - A/B) * 100\% \tag{3}$$

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

$$CR_{disk} = (1 - A/B) * 100\% \tag{4}$$

where A and B refer 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, ObjectState, 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.

7.2.3 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 de-compactor* introduced in Sec. 6, 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.

7.2.4 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

All artifacts about this work including the set of fUML models, the spreadsheet containing of the evaluation results, and the CTM traces have been collected and made publicly available.[8]

7.3 Results

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 regardless of the meta-programming 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 meta-programming approaches. However, this experiment was performed only on EMF-based languages. As CTM relies on EMF cross-references, e.g., the `originalobject` reference to **EObject** (as shown in Fig. 4), it requires any target language resolves cross-reference model elements, which may cause some limitations. To solve this tooling issue, we plan to implement indirect references from trace models to EMF models using query-driven soft traceability links as proposed by Hegedüs et al. [55]. We defer this work as the future work.

RQ #3: Scalability. Fig. 14 shows the number of objects used to represent the CTM traces with the traces generated from the domain-specific trace metamodels [21]. 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, Fig. 15

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

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. Besides, we observe that fewer references are created using CTM compared to domain-specific traces. As shown in Fig. 15, 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, Fig. 16 shows the disk space usage of the execution traces with CTM and the domain-specific traces serialized in both XML and EXI format. The Y-axis shows the amount of disk used by the trace in kilobytes (kB). Compared to XML-based domain-specific traces, we observe a significant reduction of the disk usage ranging from 92% to 96% for the CTM traces serialized in EXI, and 65% to 73% for the CTM traces 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 and disk usage than traces obtained by domain-specific trace metamodels as defined in [21]. 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 a selection of fUML models [9]. Indeed, these experiments were carried on 10 of 38 model executions of the fUML models considered for our experiments. We chose models with different number of execution states, and different parameter settings.

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 17 shows a pie 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 state modifications 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.

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 18 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 op-

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

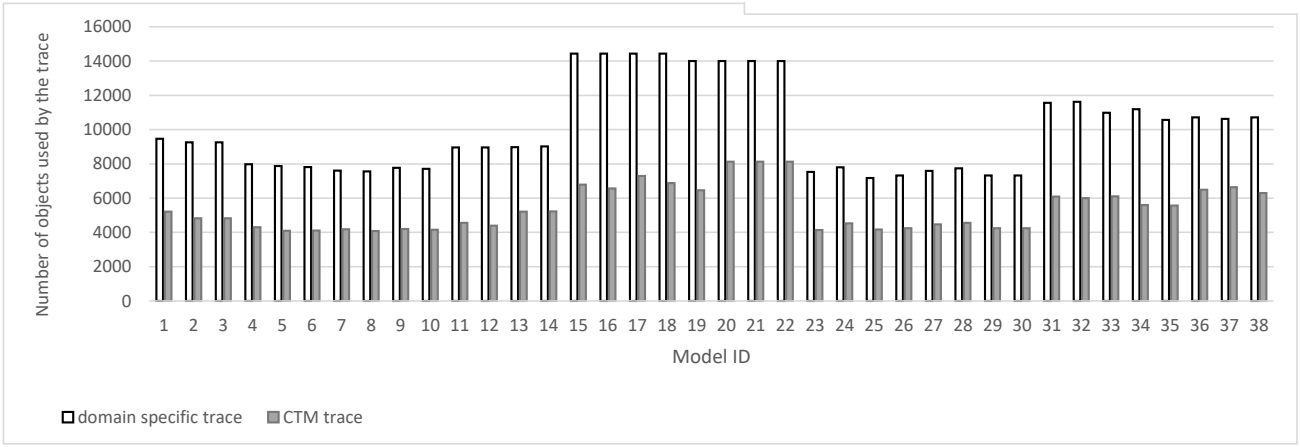


Fig. 14: Number of objects used by both CTM traces and domain-specific traces

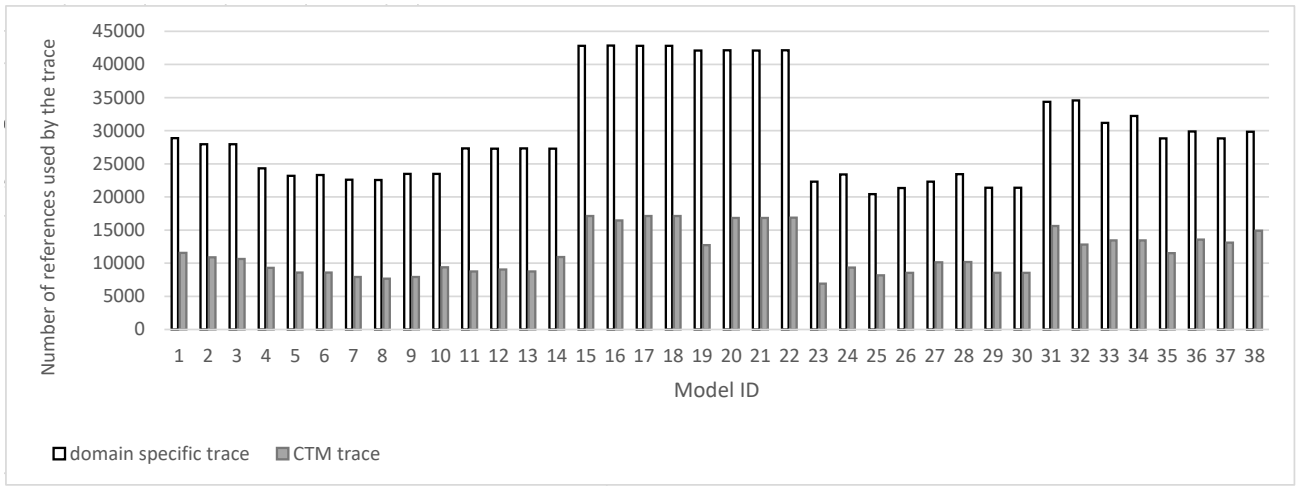


Fig. 15: Number of references used by both CTM traces and domain-specific traces

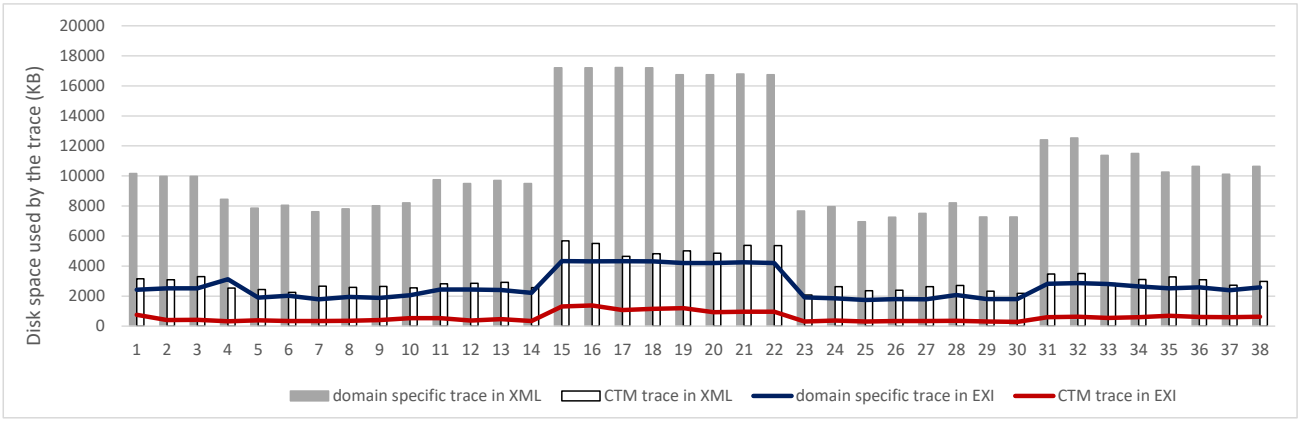


Fig. 16: Disk space used by both CTM traces and domain-specific traces

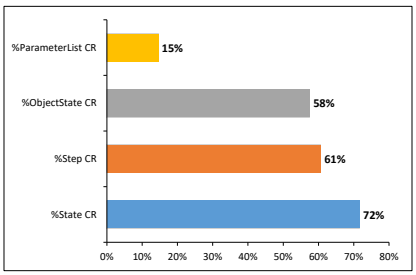


Fig. 17: Compaction rate of CTM trace elements

eration for executing it, as well as the respective memory consumption. These experiments were carried out on the same 10 fUML models[10]. 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 19 and Table 4 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 4 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)[11] 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 instances, 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 20 and Table 5 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.

8 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.

8.1 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 [34, 35, 56, 57]. A considerably less emphasis was placed on managing traces generated from executable models. We surveyed the approaches conducted in tracing executable models [25] 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.

In the TopCased project, Combemale et al. [3, 58, 59] proposed an approach to define an execution trace metamodel for discrete-event system modeling. They manually provided a metamodel specific to an xDSML. 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

[10] https://github.com/MDSEGroup/TraceCompaction/tree/master/runtime-modelingworkbench/examples.fuml.models
[11] http://www.eclipse.org/mat/

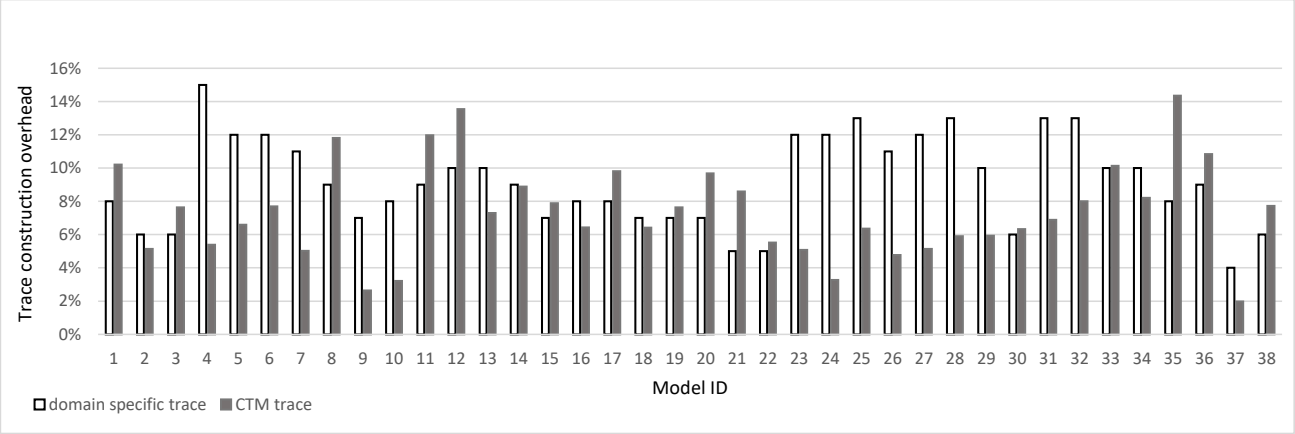


Fig. 18: Runtime overhead of the CTM and domain-specific trace construction, for each executed model

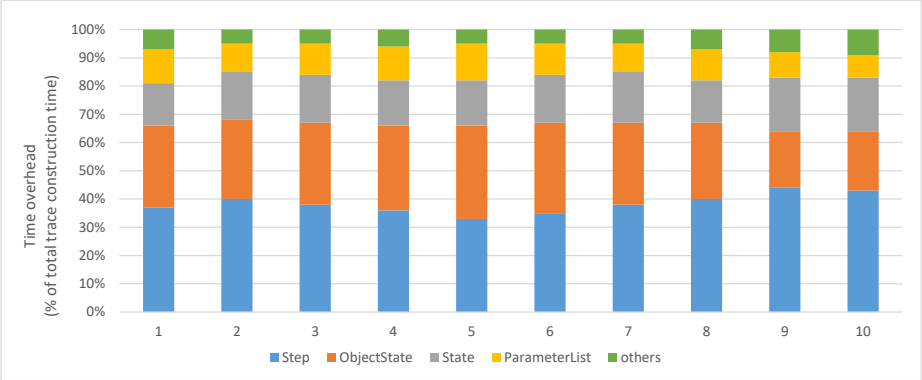


Fig. 19: Time measurements for CTM trace compaction techniques

Table 4: 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

any compaction scheme. Hence, it is not scalable to support large traces.

Hegedüs et al. [38] 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 [46] 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.

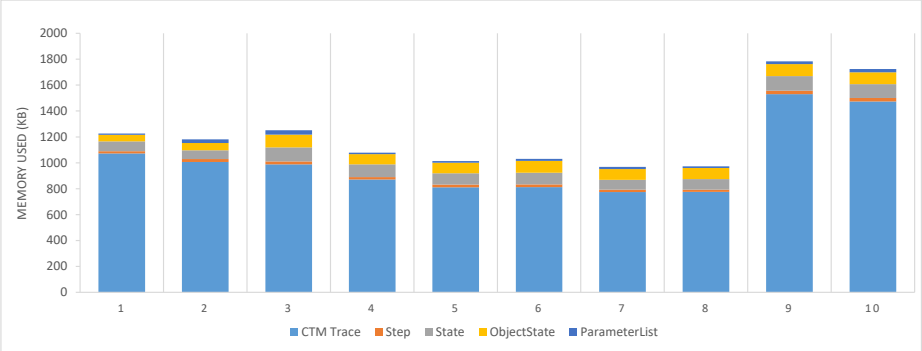


Fig. 20: Memory consumption measurements for CTM trace elements

Table 5: 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)

Meyers et al. [10] applied a generative approach as part of their ProMoBox framework, which generates domain-specific trace metamodels for xDSMLs. The obtained trace metamodel 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 approaches as mentioned above, it does not consider any technique to compact execution traces.

Similarly, Gogolla et al. [60] generated 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 huge, hindering scalability.

Aljamaan and Lethbridge [61, 62, 63] applied a different approach to enable model execution tracing. They proposed Umple[12]—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. [64, 65] 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. [66] 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 generic and supports any xDSML.

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

Furthermore, the authors' approach does not consider the compaction of traced data.

Hendriks et al. [67, 68] 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 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. [69] proposed UPPAAL as a back-end analysis tool for real-time systems. The approach provides a systematic way to define metamodels for Uppaal's 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. [21, 22] 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 [70] 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.

8.2 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 [71], event log transformation [72], trace clustering [73] and discovery techniques such as fuzzy mining [74] and pattern discovery [72, 75, 76]. 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.[73] 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 [72] 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. [77] discover patterns of local behavior from event logs. Such patterns do not capture the behavior of the complete traces. For instance, Tax et al. [77] 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.

8.3 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.

8.3.1 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. [5], Combemale et al. [3], and Schivo et al. [69, 78] 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.

8.3.2 Relational-based approaches

Another idea for persisting huge models is storing models in a relational database. An example is Connected Data Objects (CDO)[13] project in which an Ecore metamodel derives a relational schema and allows developers interacting with models. This 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. Because models tend to be interconnected with numerous references between them, querying of models require multiple expensive table joins to be executed. Hence, such approaches do not scale well for large models [79]. As an example of a relational-based approach, Dominguez et al. [80] 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.

8.3.3 Graph-based NoSQL databases

NoSQL databases provide better scalability and performance compared to relational databases [79]. MORSA [81] 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. As MORSA stores one model element per document, each model is mapped to an array of references to the documents that represent its root objects. Indeed, the EReferences of the models are serialized as document references, hindering insertion and query speed [81]. Hence, 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. [82] 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 [20], 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 rather than at the model level. Thus, for loading and saving model versions, it first needs to determine which version of a related model element must be retrieved. The effectiveness of this approach is influenced by the navigation process and the resolution mechanisms of modeling elements. Similar to this approach, we only store incremental changes rather than snapshots of a complete model. While this approach offers a considerable reduction (around 99.5%) in memory usage for small modifications, it induces a serious overhead for full model change storage. Such overhead is related to several features, i.e., detecting changes in a model, navigating in versions, comparing and merging models, and inserting new elements. Hence, for 100% of modifications of a model, this approach creates a significant overhead even more than that of the classic full model sampling strategy.

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

9 Conclusion

Dynamic V&V of models requires the ability to capture execution traces for the execution of models. We identified three main requirements for designing a trace metamodel: *genericity*, *scalability in space*, and *information preservation*. Furthermore, using such a trace metamodel for manipulating traces must induce an acceptable overhead. Our reviews of the literature show that there is a lack of approaches that address these requirements.

In this paper, we presented CTM, a metamodel for representing traces generated from executable models that is built with genericity and scalability in mind. 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 different compaction techniques that remove redundancies in model states, object states, values, steps, and parameters.

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. [21]. We also evaluated the overhead caused by CTM regarding execution time and memory consumption. 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. Moreover, the performance overhead required to the CTM trace construction is small enough that makes it practically applicable. In addition, CTM is lossless, meaning that the original trace can be fully reconstructed from the corresponding compact version.

10 Perspectives

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

Applying soft traceability links for interconnecting models. As explained in Sec. 7, we provided an EMF-based implementation for CTM. It relies on generic EMF cross-references, meaning that any target language needs storage-specific identifiers for uniquely identifying instances of model elements. This may be a challenging and error prone task for non EMF-based tools due to the identification strategies of model elements in the EMF infrastructure. One solution is to rely on indirect references from trace models to EMF-based models. A practical example is the approach proposed by Hegedüs et al. [55], which uses a soft linking technique for interconnecting EMF models by combining derived references and incremental model queries. In such approach, derived references and model elements are dynamically identified based upon query results instead of static unique identifiers. This proposal could be incorporated and implemented in a future version of CTM.

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

Besides, as mentioned in Sec. 5.2.2, the step compaction including the pattern detection is performed *offline* (i.e., after the execution of the model). In the future, we intend to design techniques that work *on the fly*, i.e., during trace generation. This way, a trace will be represented in CTM as it is generated.

Further evaluation. We intend to test CTM with more and larger traces, generated from real world models to further evaluate its efficiency. By efficiency, we mean the ability of constructing execution traces as compact as possible with minimal runtime overhead. This will require the availability of large xDSML models.

Combining compaction with compression techniques. CTM uses several compaction techniques that reduce redundancies in traces. This is different from compression techniques in information theory, where compressed data has to be uncompressed before it is used. A compact trace does not need to be "uncompacted". A compact trace can be further compressed to gain disk space and communication channels if transmitted remotely.

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 con-

tent through the analysis of a smaller more compact trace.

Applying process mining abstraction techniques. As mentioned in Sec. 8.2, 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.

A Tool Suite. The techniques presented in this paper need to be integrated with trace analysis tools. For this, we need to investigate how existing V&V techniques, especially dynamic analysis, can be used with our tracing technique.

Acknowledgement

The authors would like to acknowledge the financial sponsorship provided by co-founding of Kharazmi University and Ministry of Science, Research, and Technology (MSRT) of Islamic Republic of Iran under IMPULS Program. Lastly, the authors would like to thank Austria (OeAd) for supporting this research facilities through Contract No: 4/11937.

References

1. Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2): 25–31, 2006. doi: 10.1109/MC.2006.58.
2. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Engineering in practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, second edition, 2017.
3. Benoît Combemale, Xavier Crégut, and Marc 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)*, volume 1, pages 282–287. IEEE, 2012. doi: 10.1109/APSEC.2012.79.
4. Ábel Hegedüs, István Ráth, and Dániel Varró. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica Electrical Engineering and Computer Science*, 56(3):71–82, 2013. ISSN 2064-5279.
5. Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *Proceedings of the International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2013.
6. Jérémie Tatibouet, Arnaud Cuccuru, Sébastien Gérard, and François 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)*, volume 8767 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2014. doi: 10.1007/978-3-319-11653-2_9.
7. Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien DeAntoni, and Benoît Combemale. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 84–89. ACM, 2016. URL http://dl.acm.org/citation.cfm?id=2997384.
8. Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, 2018. ISSN 1619-1374. doi: 10.1007/s10270-018-0675-4.
9. Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4): 21:1–21:54, 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592438.
10. Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A Framework for Generating Domain-specific Property Languages. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.
11. Frank Hilken and Martin Gogolla. Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams Using Filmstripping. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 708–713. IEEE, 2016. doi: 10.1109/DSD.2016.42.
12. Earl T. Barr and Mark Marron. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, pages 67–82. ACM, 2014. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660209.
13. Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xDSMLs. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering*, pages 137–148. ACM, 2015. doi: 10.1145/2814251.2814262.

14. Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for Executable DSLs. *Journal of Systems and Software*, 137:261–288, 2018.
15. Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic model differencing utilizing behavioral semantics specifications. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2014.
16. Shahar Maoz, Jan Oliver Ringert, and Bernhard 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*, pages 179–189. ACM, 2011. doi: 10.1145/2025113.2025140.
17. Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4EMF, a scalable persistence layer for EMF models. In *Proceeding of the European Conference on Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2014.
18. Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A Metamodel for Dynamic Information Generated from Object-Oriented Systems. *Electronic Notes Theoretical Computer Science*, 94:59–69, 2004. doi: 10.1016/j.entcs.2004.01.004.
19. Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A Metamodel for the Compact but lossless Exchange of Execution Traces. *Software and Systems Modeling*, 11(1):77–98, 2012. doi: 10.1007/s10270-010-0180-x.
20. Thomas Hartmann, Francois Fouquet, Gregory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves 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*, volume 8767 of *Lecture Notes in Computer Science*, pages 252–268. Springer, 2014.
21. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling*, 18(1):385–421, 2019.
22. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *European Conference on Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2015.
23. Ken. Peffers, Tuure. Tuunanen, Marcus A. Rothenberger, and Samir. Chatterjee. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, pages 45–77., 2007. doi: 10.2753/MIS0742-1222240302.
24. Alan R. Hevner, Salvatore T. March, Jinsoo. Park, and Sudha. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.
25. Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamou-Lhadj, and Erwan Bousse. Model Execution Tracing: A Systematic Mapping Study. *Software and Systems Modeling, in press*, 2019. doi: 10.1007/s10270-019-00724-1.
26. Carl Adam Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress*, pages 386–390. North Holland, 1962.
27. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.3, July 2017.
28. Object Management Group. Business Process Model and Notation (BPMN), Version 2.0, January 2011.
29. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001. doi: 10.1023/A:1011227529550.
30. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on the Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998. doi: 10.1007/978-3-540-46464-8_21.
31. Luay Alawneh and Abdelwahab Hamou-Lhadj. *Execution traces: A new domain that requires the creation of a standard metamodel*, volume 59 of *Lecture Notes in Communications in Computer and Information Science book series*, pages 253–263. Springer, 2009.
32. Wim De Pauw, David H Lorenz, John M Vlissides, and Mark N Wegman. Execution Patterns in Object-Oriented Visualization. In *USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, volume 98, pages 1–17, 1998.
33. Dhananjay M Dhamdhere, K Gururaja, and Prajakta G Ganu. A Compact Execution History for Dynamic Slicing. *Information Processing Letters*, 85(3):145–152, 2003. ISSN 0020-0190.

34. Abdelwahab Hamou-Lhadj. Techniques to simplify the analysis of execution traces for program comprehension. In *PhD Dissertation, University of Ottawa*, 2005.
35. Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the 14th International Conference on Program Comprehension*, pages 181–190. IEEE, 2006.
36. Heidar Pirzadeh and Abdelwahab Hamou-Lhadj. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 221–230. IEEE, 2011.
37. Raymond Smith and Bogdan Korel. Slicing Event Traces of Large Software Systems. *arXiv preprint cs/0101005*, 2001.
38. Abel Hegedus, Gábor Bergmann, István Ráth, and Dániel 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)*, pages 145–155. IEEE, 2010. doi: 10.1109/SEFM.2010.28.
39. Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.
40. Daniel Abadi. Teradata rainstor's compression and performance technology. 2015. URL http://blogs.teradata.com/data-points/teradata-rainstors-compression-performance-technology/.
41. Object Management Group (OMG). XML Metadata Interchange specification, version 2.5.1, 2011.
42. Isocpp.org. Serialization and unserialization. URL https://isocpp.org/wiki/faq/serialization#serialize-text-format.
43. W3C. Efficient Extensible Markup Language (XML) Interchange (EXI), Format 1.0. Standard, IJIS Institute Technical Advisory Committee, 2014.
44. Douglas Crockford. The application/JSON media type for javascript object notation (JSON). RFC 4627, 2006.
45. Kenton Varda. Google Protocol Buffers: Google's Data Interchange Format. Technical report, 2008. URL http://code.google.com/p/protobuf/.
46. Peter Kemper and Carsten Tepper. Automated trace analysis of discrete-event system models. *IEEE Transactions on Software Engineering*, 35(2): 195–208, 2009. doi: 10.1109/TSE.2008.75.
47. Shahar Maoz and David Harel. On tracing reactive systems. *Software and Systems Modeling*, 10(4): 447–468, 2011.
48. Shahar Maoz. Using model-based traces as runtime models. *IEEE Computer Society*, 42:28–36, 2009. doi: 10.1109/MC.2009.336.
49. Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional,, 1994.
50. Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Extracting Sequence Diagram from Execution Trace of Java program. In *Proceeding of the 8th International Workshop on Principles of Software Evolution*, pages 148–151. IEEE, 2005.
51. Gabriel Valiente. Simple and Efficient Tree Pattern Matching. Report, Technical University of Catalonia, 2000.
52. Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets. In *Proceedings of Workshop on Frequent itemset Mining Implementations (FIMI'03)*, volume 90, 2003.
53. Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software and Systems Modeling*, 14(2):905–920, 2015.
54. Sebastian Bittl, Arturo A Gonzalez, M Spähn, and W Heidrich. Performance Comparison of Data Serialization Schemes for ETSIITS Car-to-X Communication Systems. *International Journal on Advances in Telecommunications*, 8(1-2):48–58, 2015.
55. Ábel Hegedüs, Ákos Horváth, István Ráth, Rodrigo Rizzi Starr, and Dániel Varró. Query-driven soft traceability links for models. *Software & Systems Modeling*, 15(3):733–756, 2016.
56. Fazilat Hojaji, Bahman Zamani, and Abdelwahab Hamou-Lhadj. Towards a tracing framework for model-driven software systems. In *Proceedings of the 6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 298–303. IEEE, 2016.
57. Heidar Pirzadeh, Sara Shanian, Abdelwahab Hamou-Lhadj, Luay Alawneh, and Arya Sharifee. Stratified sampling of execution traces: Execution phases serving as strata. *Elsevier Journal of Science of Computer Programming, Special Issue*

on Software Evolution, Adaptability and Maintenancey, 78(8):1099–1118, 2013.

58. Benoit Combemale, Xavier Crégut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing simulation and model animation in the MDE Topcased toolkit. In *Proceedings of the 4th European Congress Embedded Real Time Software (ERTS)*, 2008.
59. Xavier Crégut, Benoit Combemale, Marc Pantel, Raphaël Faudoux, and Jonatas Pavei. Generative Technologies for Model Animation in the TopCased Platform. *ECMFA*, 6138:90–103, 2010.
60. Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In *Modellierung*, volume 225, pages 273–288, 2014.
61. Hamoud Aljamaan and Timothy C Lethbridge. Towards Tracing at the Model Level. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pages 495–498. IEEE, 2012. doi: 10.1109/WCRE.2012.59.
62. Hamoud Aljamaan, Timothy C Lethbridge, Omar Badreddin, Geoffrey Guest, and Andrew 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)*, pages 79–86. IEEE, 2014.
63. Hamoud I Aljamaan, Timothy Lethbridge, Miguel Garzón, and Andrew Forward. UmpleRun: a dynamic analysis tool for textually modeled state machines using Umple. In *Proceedings of the First International Workshop on Executable Modeling colocated with MODELS 2015*, pages 16–20, 2015.
64. Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. Execution and simulation of (profiled) UML models using Populo. In *Proceedings of the international workshop on Models in software engineering*, pages 75–81. ACM, 2008. doi: 10.1145/1370731.1370749.
65. Lidia Fuentes and Pablo Sánchez. Dynamic Weaving of Aspect-Oriented Executable UML Models. *Transactions on Aspect-Oriented Software Development*, 5560:1–38, 2009.
66. Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fUML. In *Proceedings of the 7th Workshop on Models@ run. time*, pages 53–58. ACM, 2012. doi: 10.1145/2422518.2422527.
67. M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers. Analyzing execution traces: critical-path analysis and distance analysis. *Proceedings of the International Journal on Software Tools for Technology Transfer*, pages 1–24, 2016. doi: 10.1007/s10009-016-0436-z. Export Date: 22 December 2016 Article in Press.
68. Martijn Hendriks, Jacques Verriet, Twan Basten, Bart Theelen, Marco Brassé, and Lou Somers. Analyzing Execution Traces: Critical-path Analysis and Distance Analysis. *International Journal on Software Tools for Technology Transfer*, 19(4):487–512, 2016. doi: 10.1007/s10009-016-0436-z.
69. Stefano Schivo, Buğra M Yildiz, Enno Ruijters, Christopher Gerking, Rajesh Kumar, Stefan Dziwok, Arend Rensink, and Mariëlle" 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*, volume 10606 of *Lecture Notes in Computer Science*, pages 319–336. Springer, 2017.
70. Luay Alawneh and Abdelwahab Hamou-Lhadj. An exchange format for representing dynamic information generated from high performance computing applications. *Elsevier Journal of Future Generation Computer Systems*, 27(4):381–394, 2011.
71. Niek Tax, Natalia Sidorova, and Wil M. P. van der Aalst. Discovering more precise process models from event logs by filtering out chaotic activities. *Journal of Intelligent Information Systems*, 52(1):107–139, Feb 2019. ISSN 1573-7675. doi: 10.1007/s10844-018-0507-6. URL https://doi.org/10.1007/s10844-018-0507-6.
72. R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. Abstractions in Process Mining: A Taxonomy of Patterns. In Umeshwar Dayal, Johann Eder, Jana Koehler, and Hajo A. Reijers, editors, *Business Process Management*, volume 5701 of *Lecture Notes in Computer Science*, pages 159–175, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
73. Minseok Song, Christian W Günther, and Wil MP Van der Aalst. Trace clustering in process mining. In *International Conference on Business Process Management*, volume 17 of *Lecture Notes in Business Information Processing*, pages 109–120. Springer, 2008.
74. Christian W Günther and Wil MP Van Der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *International conference on business process management*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2007.
75. Claudia Diamantini, Laura Genga, and Domenico Potena. Behavioral process mining for unstructured processes. *Journal of Intelligent Information Systems*, 47(1):5–32, Aug 2016. ISSN 1573-7675. doi:

10.1007/s10844-016-0394-7. URL https://doi.org/10.1007/s10844-016-0394-7.

76. Veronica Liesaputra, Sira Yongchareon, and Sivadon Chaisiri. Efficient Process Model Discovery Using Maximal Pattern Mining. In Hamid Reza Motahari-Nezhad, Jan Recker, and Matthias Weidlich, editors, *Business Process Management*, volume 9253 of *Lecture Notes in Computer Science*, pages 441–456, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23063-4.
77. Niek Tax, Natalia Sidorova, Reinder Haakma, and Wil M. P. van der Aalst. Event abstraction for process mining using supervised learning techniques. *Lecture Notes in Networks and Systems*, page 251–269, Aug 2017. ISSN 2367-3389. doi: 10.1007/978-3-319-56994-9_18. URL http://dx.doi.org/10.1007/978-3-319-56994-9_18.
78. Stefano Schivo, Jetse Scholma, Brend Wanders, Ricardo A Urquidi Camacho, Paul E van der Vet, Marcel Karperien, Rom Langerak, Jaco van de Pol, and Janine N Post. Modeling biological pathway dynamics with timed automata. *IEEE journal of biomedical and health informatics*, 18(3):832–839, 2014. doi: 10.1109/BIBE.2012.6399719.
79. Konstantinos Barmpis and Dimitrios S Kolovos. Comparative analysis of data persistence technologies for large-scale models. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 33–38. ACM, 2012.
80. Eladio Domínguez, Beatriz Pérez, and María A Zapata. A UML profile for dynamic execution persistence with monitoring purposes. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pages 55–61. IEEE, 2013. doi: 10.1109/MiSE.2013.6595297.
81. Javier Espinazo Pagán, Jesús Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *International Conference on Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2011.
82. Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. Analyzing complex data in motion at scale with temporal graphs. In *The 29th International Conference on Software Engineering and Knowledge Engineering (SEKE'17)*, page 6. KSI Research, 2017.