Towards Scalable Multidimensional Execution Traces for xDSMLs

Erwan Bousse[1], Benoit Combemale[2], and Benoit Baudry[2]

[1] University of Rennes 1, France
erwan.bousse@irisa.fr
[2] Inria, France,
{benoit.combemale, benoit.baudry}@inria.fr

Abstract. Executable Domain Specific Modeling Languages (xDSML) opens many possibilities in terms of early verification and validation (V&V) of systems, including the use of *dynamic* V&V approaches. Such approaches rely on the notion of *execution trace*, *i.e.* the evolution of a system during a run. To benefit from dynamic V&V approaches, it is therefore necessary to characterize what is the structure of the executions traces of a given xDSML. Our goal is to provide an approach to design trace metamodels for xDSMLs. We identify seven problems that must be considered when modeling execution traces, including concurrency, modularity, and scalability. Then we present our envisioned approach to design scalable multidimensional trace metamodels for xDSMLs. Our work in progress relies on the *dimensions* of a trace (*i.e.* subsets of mutable elements of the traced model) to provide an original structure that faces the identified problems, along with a trace API to manipulate them.

1 Introduction

In the recent years, a lot of effort have been made to provide tools and methods to design executable Domain Specific Languages (xDSMLs) [4,5]. Executability of models opens many possibilities in terms of early verification and validation (V&V) of systems, including the possiblity to rely on *dynamic* V&V approaches such as debugging, runtime verification [6] or model checking [2].

A central concept in dynamic V&V approaches is the *execution trace*, which represents the evolution of a system during a run. A trace is the alternate sequence of *states* of the system and *events* that triggered the state changes. All previously cited approaches rely on traces: model checking consists in verifying a property of a system by analyzing all its possible traces, and a counter-example in the form of a trace is provided if it is not satisfied; runtime verification consists in checking whether or not a trace satisfies a property; debuggers require traces to be able to replay faulty scenarios in order to investigate for bugs.

Henceforth, a significant prerequisite for the V&V of executation models is the definition of the structure of the execution traces of a considered xDSML. Since an xDSML should define what is the state (also called *runtime data*) of

a model during its execution [4], a *trace metamodel* could potentially be defined based on this information. But many questions remain regarding what data should contain a trace, or how it should be structured. In this paper, we identify a number of problems to face when modeling execution traces, and we present an approach to face these problems. We first introduce in Section 2 a motivating example, which is an xDSML called RoboML and a simple scenario involving two robots following one another. Then in Section 3 we list and illustrate a series of seven problems linked to trace modeling, including concurrency, modularity and scalability. In Section 4 we present our approach to generate scalable and multidimensional trace metamodels. Finally Section 5 concludes with perspectives.

2 Motivating example

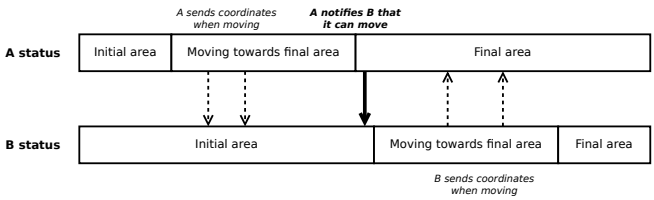

Fig. 1. Scenario of two robots following one another

Our motivating example consists in an xDSML called RoboML, which extends timed automata with hierarchy, events, and domain specific actions. A RoboML model consists in a set of communicating timed automata. When executed, the current state of the automata changes depending on the current state, events, clocks and conditions. Additional domain-specific runtime data is available, such as the GPS coordinates of the robot. Actions triggered on transitions allow a robot to move, interact and communicate with other robots. The very simple scenario we consider is based on two robots A and B, each configured with a RoboML model. Robots are configured to regularly send their coordinates to the other robot when they change their position. Figure 1 illustrates the scenario, which can be summed up in the following way:

1. Both robots are in an initial area.
2. Robot A moves towards a final area.
3. When arrived, A waits 5 seconds and then sends a message to B to move.
4. Then B moves to the final area.

We want to check the following two properties:

(a) B starts moving 5 seconds after A, not before nor after.
(b) Each time a robot covers 1 meter, it sends a message to the other one with its new coordinates.

3 Problems when Modeling Traces

In this section, we present a series of problem that must be dealt when modeling execution traces, and we illustrate them with the example introduced in the previous section.

3.1 Trace Contents

The first category of problems concerns the contents of an execution trace, apart from the state of the executed model and the events that triggered state changes.

(Pb. 1) Concurrency modeling The execution of one or multiple models may imply concurrent variables within runtime data. In such case, the different states of these variables may or may not be independent from one another. In our example, the status of a robot is most of the time completely independent from the status of the other. However, communications between the robots imply some kind of ordering between the states of the robots. For instance, we know that B didn't start moving before receiving a message from A, thus allowing us theoretically to verify property (a). A challenge is thus to take into account these concurrency relationships between variables into the trace structure.

(Pb. 2) User-defined additional information Analyzing the behavior of a system requires information about how it changes over time. Yet, in some cases, properties may concern data that we do not directly have. In our example, property (b) concerns the meters covered by a robot, which is not directly available in the state of the executed model. Such variable could easily be derived from the evolution of the coordinates of the robot and would belong in the trace to verify such properties.

(Pb. 3) Scalability in space Execution traces can be arbitrarily large, as some complex systems are monitored continuously in case a failure occurs. Thus a challenge is to manage scalability in space when manipulating traces, whether it is offline (file or database storage) or online (in memory). In our example, a robot changes its internal state all the time to update its coordinates or listen to communications, leading quickly to a large amount of states.

3.2 Trace Manipulations

The second category of problems concerns the eventual manipulations of an execution trace, which may constrain the structure of the trace.

(Pb. 4) Modularity A trace must be *constructed* during the run of a program, in order to be *manipulated* either during or after the run. Modularity of the trace is a problem for both the construction and the manipulation phases. First, when tracing a system, one can be interested in observing only a subset of the variables, which require a non-monolithic and modular trace format. Second,

when manipulating a trace, one might want to extract a subset of the information (e.g. a subtrace with only a selection of variables), or conversely to add new information within the trace (e.g. derived variables). In our example, property (a) only concerns the movement states of the robots and not their coordinates, thus extracting a trace with only the former information would be relevant to prove this property.

(Pb. 5) Manipulation safety Generic trace metamodels exist to model all kinds of traces for any executable language [1]. However, constructing traces with such metamodels may lead to inconsistent trace models, since their genericity does not forbid one to create a trace whose states are not relevant to the concerned xDSML. In our example, for instance, we may want to ensure that coordinates are stored as a pair if integers instead of a string.

(Pb. 6) Reuse of trace manipulations To verify properties on traces, or to be able to write interesting queries to explore them, we must define operations that manipulate traces. An important need is to be able to reuse such operations from a trace to another, from a system to another, or even from an xDSML to another. In our case, verifying property (a) requires an operator that checks all states that are found 5 seconds after a specific situation, which can be generalized in a *within* operator.

(Pb. 7) Scalability in time Traces are eventually analyzed, which requires iterating over the steps of the trace. The potentially large size of a trace compromises the capacity to make queries in a reasonable time. Moreover, if some variable referenced in a property only changes lately in a trace, we would still have to iterate through all steps before noticing that change. This is the case with property (a) of our example, where B starts moving very lately.

4 Envisioned Approach

In this section, we present the multiple choices we made for our approach to design scalable multidimensional trace metamodels, and how these choices participate in solving the problems stated in the previous section.

4.1 Trace Structure

Our approaches relies on an original way to structure execution traces. We illustrate it informally in Figure 2, and we highlight important choices in the following paragraphs.

Multiple dimensions A trace can be defined as a single alternate sequence of *states* and *events*. Yet, it is very likely that only a subset of the variables really change from a state to another. Our idea is thus to consider multiple *dimensions* in a trace, each being a set of mutable elements of the executed model. We then

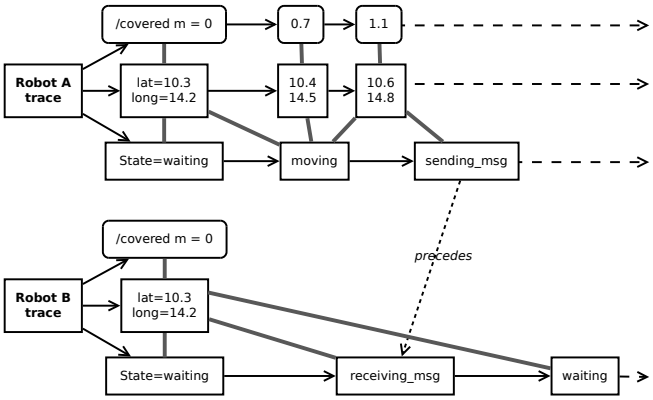

Fig. 2. Intuitive and partial representation of a multidimensional trace, matching the very beginning of the robots scenario (until the first coordinates message is sent).

define a trace as a set of *subtraces*, each being the evolution of a specific dimension within the run. Such structure allow us to solve many problems. First, by manipulating dimensions separately, we can define concurrency relationships between them (Pb. 1). More precisely, we consider that states can be linked by *observations* (*i.e.* states that were simultaneous at some point) or *synchronizations* (*i.e.* states that changed simultaneously), while events can be linked by ordering relationship such as *precedence* or *coincidence* (we refer to [7] for more relationships between events). Second, we gain modularity (Pb. 4), as we can add or remove dimensions depending on the needs. Third, such modularity helps enriching the trace with additional dimensions (Pb. 2). Finally, this structure allow us to iterate separately through different dimensions, which should improve scalability in time (Pb. 7). Figure 2 shows an example: our robot trace consists in six dimensions (three per robot: covered meters, coordinates, state). We have a precedence relationship between the sending and the receiving of the message. Covered meters being not in the runtime data, it is added as a new dimension derived from coordinates.

Data sharing For scalability in space (Pb. 3), our idea is to maximize data sharing among steps of the trace, *i.e.* reduce redundancy from a step to another. Our approach relies on ideas of our previous work on scalable model cloning [3]. More precisely, each dimension state is stored in a dedicated storage structure in order to be referenced by the steps of the same dimension.

Domain specific trace metamodel To provide manipulation safety to trace models (Pb. 5), our solution is to design *domain specific* trace metamodels, *i.e.* metamodels each of which defines precisely what are the traces of a single xDSML. This choice ensures that all traces are consistent with regard to the traced language. In addition, we can define the state of a model of a given

xDSML without relying on unsafe generic types (e.g. *EObject* when using the Eclipse Modeling Framework (EMF)). The drawback is that is becomes necessary to provide one trace metamodel per xDSML, but we plan to overcome this by providing a generative approach.

4.2 Trace API

Generating trace metamodels has multiple advantages as stated in the previous section. However, the main drawback is that operations defined for a given trace metamodel are not compatible with a different trace metamodel (Pb. 6). Following the same trend as recent first-class traces approaches [8], our solution is to generate, along with a trace metamodel, a complete API with trace specific operations in order to refine, query, explore, or transform a trace. Operations such as *filter*, *merge*, *slice*, *during*, etc. are part of this API.

5 Conclusion

Verification and validation of executable models is a challenge that requires the modeling of execution traces. We identified seven problems that must be considered when modeling traces, and we presented our idea of multidimensional traces coupled with a trace manipulation API. Such traces take into account concurrency, scalability, modularity among other aspects. Further work will be the implementation of the approach and its application to RoboML.

Acknowledgement. This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

1. Luay Alawneh and A Hamou-Lhadj. Execution traces: A new domain that requires the creation of a standard metamodel. *Advances in Software Engineering*, 2009.
2. Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*.
3. Erwan Bousse, Benoit Combemale, and Benoit Baudry. Scalable Armies of Model Clones through Data Sharing. *MODELS 2014, Valencia, Spain*, 2014.
4. Benoît Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. *The 19th Asia-Pacific Software Engineering Conference*, 2012.
5. Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In *International Conference on Software Language Engineering*, 2013.
6. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2009.
7. Frédéric Mallet. Clock constraint specification language: Specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 4, 2008.
8. Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Expositor: Scriptable time-travel debugging with first-class traces. *ICSE 2013, San Francisco, CA*, 2013.