# Navigating the trace of executable domain specific languages through a trace domain query language

**Hiba AJABRI[†], Jean-Marie MOTTU[†‡], Christian ATTIOGBE[†], and Pascal BERRUET[§]**
[†]Nantes Université, École Centrale Nantes, IMT Atlantique(‡), CNRS, LS2N, UMR 6004, F-44000 Nantes, France
[§]Lab-STICC/University of Bretagne Sud, Lorient, France

**ABSTRACT** Executable Domain Specific Languages (xDSL) enable domain experts to design and simulate the systems they develop. Various strategies have been explored for capturing system execution traces, allowing domain experts to perform advanced analysis. However, extracting meaningful data from these traces presents significant challenges for many domain experts due to: (1) a limited understanding of the trace structure and (2) a lack of software development skills needed to implement programs that navigate and extract data from execution traces. This paper presents an approach that allows domain experts to write trace queries using domain terminology, without relying on developers or requiring an in-depth understanding of the trace structure. This approach is at the heart of our main contribution: Trace Domain Query Language (*TraceDQL*), a query language that enables domain experts to write queries that explore the execution trace and return the requested data. This language is designed to be applicable across different application domains. The operational semantics of *TraceDQL* abstracts away the complexity of trace navigation, enabling domain experts to extract data without needing to understand the underlying trace structure. We illustrate our proposal with two case studies: the Simple Manufacturing System (SMS) xDSL and the Arduino xDSL.

**KEYWORDS** Domain-Specific Language, Model Execution, Execution Trace, Trace Query Language, Object Constraint Language (OCL).

## 1. Introduction

In Model-Driven Engineering, there is a shift from Domain-Specific Languages to *executable* Domain-Specific Languages (xDSLs) which are widely used not only to describe models but also to simulate their behavior (i.e., models are executable) for various purposes; for instance, simulating industrial systems (Kaiser et al. 2022), verifying and testing purposes (Lübke & Van Lessen 2017), or enabling further computations (An et al. 2011). During simulations, the modifications of the system state are recorded in an *execution trace* that provides an efficient means of storing execution data for advanced analysis without always running the system again. These traces can be analyzed to extract valuable data for domain experts (Alawneh & Hamou-Lhadj 2009; Aljamaan et al. 2014; Bousse et al. 2014).

For example, in industrial engineering, xDSLs are designed to model manufacturing systems and to simulate their behavior. In a previous study (Ajabri et al. 2024), we introduced the Simple Manufacturing System xDSL (SMS xDSL) to model and simulate simple manufacturing systems. Therefore, an expert in the industrial domain can define and run SMS models, such as the **Hammer SMS** model illustrated in Fig. 1(a). The system includes three machines: *GenHandle* (producing handles), *GenHead* (producing heads), and *Assembler* (combining them into hammers). *Conveyors* transport the products, and a tray holds them. An execution of the Hammer SMS model simulates the Hammer production system and generates an execution trace that captures the different states of the system. Fig. 1 shows two states from the Hammer SMS simulation: (a) a hammer head on *conveyor A*, and (b) the same head placed in the *tray*.

While the generation of the execution trace allows domain experts to dynamically store a large amount of information,

**JOT reference format:**
Hiba AJABRI, Jean-Marie MOTTU, Christian ATTIOGBE, and Pascal BERRUET. *Navigating the trace of executable domain specific languages through a trace domain query language*. Journal of Object Technology. Vol. 24, No. 2, 2025. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2025.24.2.a8

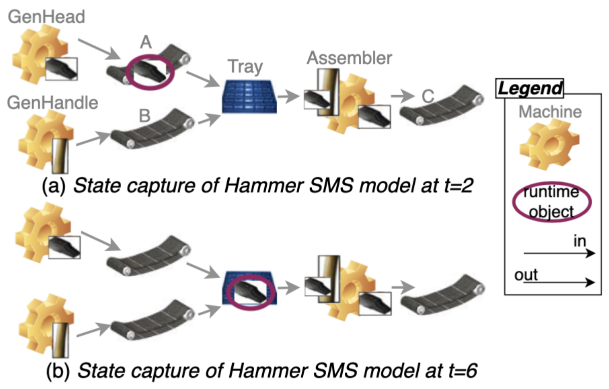

GenHead
A
Tray
Assembler
C
GenHandle
B
Legend
Machine
runtime object
in
out
(a) State capture of Hammer SMS model at t=2
(b) State capture of Hammer SMS model at t=6

**Figure 1** Two states of the Hammer production SMS model at different times, captured during the simulation.

extracting specific data from it remains a challenging task. Implementing a software component (e.g., EMF Java code, OCL query) to navigate the trace and retrieve the required data is challenging because: (1) navigating the trace requires a thorough understanding of the trace structure; (2) developing a dedicated software component requires software development skills that not all domain experts have. Hence, the assistance of software experts may help, but domain experts would prefer to write and manage their queries, using domain concepts and terms, to gather data from the trace, allowing them to focus on what is stored and requested in the trace instead of how it is stored. Moreover, this trace analysis should be general enough to be applied to other models, described with different xDSL.

Regarding these difficulties, two research questions may be formulated:

- **RQ#1:** can we design a query language to exploit execution traces regardless of the application domain of the model being traced ?
- **RQ#2:** can we make such a trace query language reusable and easy to use by domain experts ?

To address these research questions, we present our contribution in the form of Trace Domain Query Language (*TraceDQL*); it is a generic, executable language that enables the domain experts to write queries using their domain terminology, to explore execution traces of a given model, and to retrieve needed data from traces. *TraceDQL* provides operators that simplify the task of writing queries. For instance, considering an execution trace from the Hammer SMS model simulation, the domain expert may want to retrieve the quantity of hammer heads produced during this simulation. Retrieving these specific data requires navigating through the model states recorded in the execution trace and counting the hammer heads. However, the same hammer head is recorded several times (e.g., in Fig. 1 the head is first recorded on *conveyor A* and a second time in the *tray*). Implementing a software to count hammer heads and to remove duplicates may be complex for the domain expert. Thus, *TraceDQL* offers for instance a *REMOVE REDUNDANCY BY* instruction. In addition, *TraceDQL* supports query reusability. A parametric query can be reused with new parameter values, a query can be reused in another query through a call, and the output of a query can be reused as input for another query, depending on its type. Moreover, the operational semantics of the *TraceDQL* makes navigating the execution trace transparent to the domain expert, and avoids having to understand the structure of the trace language.

To illustrate the contribution, we consider two case studies: an updated version of the Simple Manufacturing System (SMS) xDSL and the Arduino xDSL. The implementation is available on a public GitLab repository[1] and it is permanently available via this link (Ajabri et al. 2025).

In the remainder, Section 2 discusses the research background. Section 3 outlines the approach and explains how the domain expert may use domain-specific queries to extract data from the execution trace. Thereafter, Section 4 presents the *TraceDQL* language. Sections 5 discloses the application of *TraceDQL* with the SMS xDSL and the Arduino xDSL considering different examples and in Section 6 we conduct the discussion. Finally, the related work is outlined in Section 7, and the conclusion and future work are drawn in Section 8.

## 2. Background

The background section presents the main concepts and underlying approaches used in our contributions, illustrated through the Simple Manufacturing System xDSL case study.

### 2.1. Executable domain-specific languages

A Domain-Specific Language (DSL) is a language tailored to a specific domain. Executable Domain-Specific Languages (xDSLs) extend DSLs beyond system modeling by enabling execution (Combemale et al. 2012). Three essential elements constitute an xDSL: abstract syntax, execution semantics, and concrete syntax. The abstract syntax defines the concepts of the language and their relationships. It is typically represented as an object-oriented model, called a metamodel. The execution semantics determine how a model conforming to the abstract syntax is executed. There are two main types: translational semantics (i.e., compilation) (Cleenewerck & Kurtev 2007), and operational semantics (i.e., interpretation) (Plotkin 2004). In this paper, we adopt the operational semantics. The operational semantics of an xDSL consists of two parts: A runtime state definition, which describes the possible runtime states of a model under execution. A set of execution rules, which dictate how the runtime state evolves over time. Finally, the concrete syntax is the notation used to represent abstract syntax concepts. It can be either textual or graphical.

### 2.2. Simple manufacturing system xDSL case study

In previous work, we developed the Simple Manufacturing System xDSL (SMS xDSL) (Ajabri et al. 2024). Fig. 2 presents an excerpt from the SMS xDSL metamodel, while the complete version is available in a public GitLab repository[2], and it is permanently available at this link (Ajabri et al. 2025). This metamodel contains 16 metaclasses and consists of two main

[1] https://gitlab.univ-nantes.fr/rodic/evaluationKPIxDSL
[2] https://gitlab.univ-nantes.fr/rodic/evaluationKPIxDSL/-/tree/master/Language_Workbench/xSMS

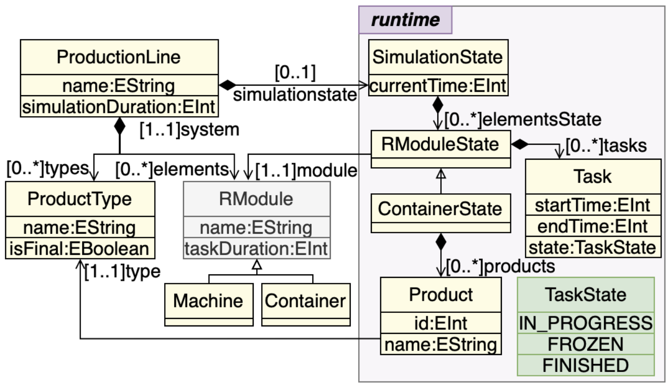


**Figure 2** Excerpt from the SMS metamodel: abstract syntax (left part) and runtime state definition (right part).

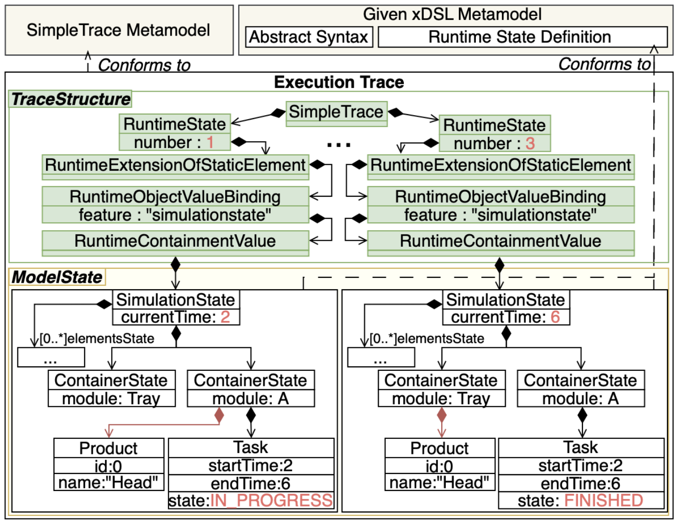


**Figure 3** An excerpt from the execution trace model of a Hammer SMS model execution, that includes the states (a) and (b) depicted in Fig. 1, illustrating that the *tray* has received the hammer head (red associations).

parts: the abstract syntax (left side of Fig. 2) and the runtime state definition (right side). The abstract syntax defines the root metaclass *ProductionLine*, containing a set of *RModules* (representing machines or containers such as conveyors and trays), and *ProductTypes* (defining different product types in the system). The runtime state definition defines the system's state during execution. It introduces additional dynamic metaclasses and features to represent the runtime state of static language concepts. The *runtime* package contains a *SimulationState* metaclass, which tracks simulated time via the *currentTime* attribute. *SimulationState* has a set of *RModuleState*, which references the *RModule* and contains a set of *Task* elements defined by *startTime*, *endTime* and *state* attributes. A *ContainerState* is a subtype of *RModuleState*, which can contain a set of *Product* during runtime. A *Product* has an *id*, a *name* and a *type* reference to the *ProductType* metaclass.

## 2.3. Modeling and execution framework

Designing an xDSL such as SMS xDSL can be accomplished using a Modeling and Execution Framework. GEMOC Studio[3] is an Eclipse extension that provides a comprehensive set of tools for developing xDSLs. The abstract syntax of an xDSL can be defined in GEMOC Studio using the Eclipse Modeling Framework (EMF). The GEMOC Execution Framework (Bousse et al. 2016) and various meta-programming approaches allow the definition of execution semantics. The framework also supports the development of the concrete syntax of an xDSL. Finally, the framework allows models conforming to the xDSL to be executed. In-depth analysis of such executions often requires tracing the execution process.

The SMS xDSL, designed with this framework, allows the modeling of a hammer production line (Hammer SMS model in Fig. 1), and simulating its behavior. This simulation can answer questions such as: *what was the duration of the simulation?* or *how many hammers were produced in a given period?* Such questions can be answered by looking at the final state of the model under simulation in the execution trace. However, more complex analyses may require examining multiple states, for instance: *how many hammer heads were produced in a given period, after three hours? At which state was the first hammer produced?* Such insights require tracing the execution for further analysis.

## 2.4. Trace language and execution traces

A trace language is used to define execution traces, which are models conforming to a trace language metamodel. To capture the state of the model during execution, the trace language must allow to select which elements of the given xDSL will be traced. It is a feature of genericity. Formally, a language $L$ described by a metamodel ($MM$), is the set of the models generated by $MM$ or equivalently that conform to $L$. For any model $m$ conforming to the metamodel $MM$ (that describes the language $L$), we have $m \in L$, meaning $m$ conforms to $MM$. Let $L$ be a given xDSL and $T_{[L]}$ be the trace language, considering $m \in L$ and $t \in T_{[L]}$, $t_m$ is the execution trace of $m$ meaning $t$ embeds a portion of $m$, which is the model state that conforms to the runtime state definition of $L$. Therefore, the execution trace $t_m$ is a parametric trace which contains two parts: a part that pertains to the trace language structure *TraceStructure*, and a part that comprises the model states *ModelState*. Fig. 3 presents an excerpt from an execution trace of the Hammer SMS model and illustrates these two parts.

***Trace framework*** In our work, we use the *SimpleTrace*[4] framework to generate execution traces, as illustrated in Fig. 4. It includes a *Trace Generator and is implemented using Gemoc Studio.* Given an xDSL, the language engineer annotates the runtime state elements that should be traced, using the "dynamic" keyword. A domain expert defines and executes a model conforming to the given xDSL. At the end of the execution, the captured runtime states are collected and structured into the execution trace model.

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

[4] https://github.com/gemoc/simpler-traces

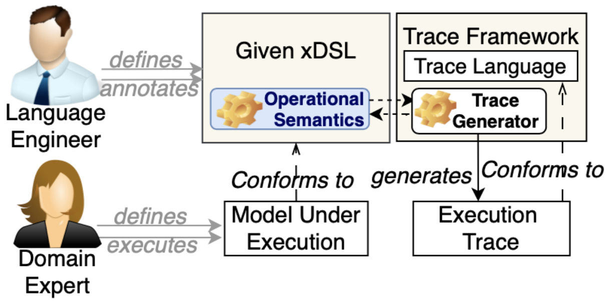


**Figure 4** Execution trace generation process capturing model states over execution.

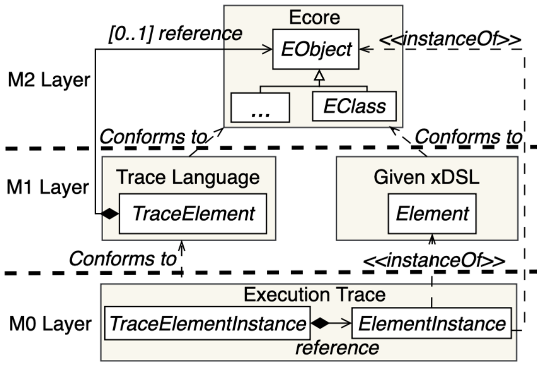


**Figure 5** Layered structure involving the trace language.

To integrate an xDSL's runtime states into the trace, the SimpleTrace language employs a layered structure, as depicted in Fig. 5. We consider a metaclass *TraceElement* of the trace language metamodel, which has a containment *reference* to *EObject*, the root for all elements in the Ecore[5] metamodel (i.e. M2-layer in the MOF[6] standard). The *reference* defined at the level of M2- M1-layer allows an instance *TraceElementInstance* of *TraceElement* to contain instances of *EObject*. An instance *ElementInstance* of a metaclass *Element* of a given xDSL is an instance of *EObject* that may be contained in *TraceElementInstance*, at the model level (i.e., M0- layer). Therefore, the execution trace $t_m$ can embed instances of metaobjects in a given xDSL since the trace language metamodel refers to Ecore objects, which reveals the parametric aspect of $t_m$. For SimpleTrace, only instances of metaobjects of a given xDSL annotated as dynamic or contained in metaobjects annotated as dynamic, are included in the trace because they represent the model state during runtime.

***Trace hypotheses*** We formulate two key hypotheses to support our proposal. First, among two main types of execution traces in the literature: event-based traces and state-based traces (Ezzati-Jivan & Dagenais 2012; Montplaisir et al. 2013), we consider the second one, which captures the states of the system under execution. The captured states must represent the *ModelState* part. Second, the given xDSL must be annotated appropriately: the *Trace Generator* (see Fig. 4) requires annotating runtime state elements that should be traced, using the "dynamic" keyword. It is essential to identify, based on the annotation of the trace framework, which elements are relevant for the domain expert when analysing a trace. This allows us to relieve the domain experts from specifying such structural information in the queries, enabling them to focus solely on the analysis task, without having to manage the internal structure of the trace.

## 2.5. Putting into practice and main issues

The given xDSL and SimpleTrace language are both implemented using GEMOC Studio. It enables the definition of the SMS xDSL, the instantiation of models such as the Hammer SMS model, and their simulation, ultimately producing an execution trace. Fig. 3 presents an excerpt from the execution trace of the Hammer SMS model, illustrating the two parts of this parametric trace: *TraceStructure* and *ModelState*.

It is challenging for domain experts to extract data from such parametric traces $t_m$. It requires: (1) navigating the *TraceStructure* and (2) exploring each *ModelState* and moreover (3) comparing and analyzing different ModelStates over time.

For example, to count the amount of hammer heads generated during a simulation, a domain expert must implement the algorithm shown in Fig. 6. The algorithm achieves three main tasks: task (1) defines the *getCaptures(traceMM,executionTrace)*, which navigates the *TraceStructure*, task (2) navigates each *ModelState* of the SMS xDSL to identify the *hammer heads*, and task (3) prevents counting several times the same hammer through the different ModelStates. Therefore, domain experts must implement the first task of the algorithm; this requires a good understanding of the trace structure (1). They must implement the second part to navigate the model state which conforms to the runtime state definition of the SMS xDSL metamodel (2), and then count the hammer heads products without duplicates (3). For instance, in Fig. 3 the same product Head is duplicated.

This highlights the need for dedicated tools and abstractions to assist domain experts in querying and analyzing traces with-

```
Inputs:
smsMM: the SMS xDSL metamodel
traceMM: the simple trace metamodel
executionTrace: the trace of the execution of the model
begin
    countHead ← 0;
    ids ← [];
(1) foreach capture ∈ getCaptures(traceMM, executionTrace) do
        simulation ← capture.castTo(SimulationState);
(2)     foreach moduleState ∈ simulation.getElementsState() do
            if moduleState instanceof ContainerState then
                container ← moduleState.castTo(ContainerState);
                foreach p ∈ container.getProducts() do
                    if p.type = Head and !ids.contains(p.id) then
(3)                     countHead ← countHead + 1;
                        ids.add(p.id);
end end end end end end
```

**Figure 6** An algorithm to navigate in the execution trace of the *Hammer* production line SMS model and to compute the quantity of hammer heads generated during the simulation.

[5] https://eclipse.dev/modeling/
[6] https://www.omg.org/mof/

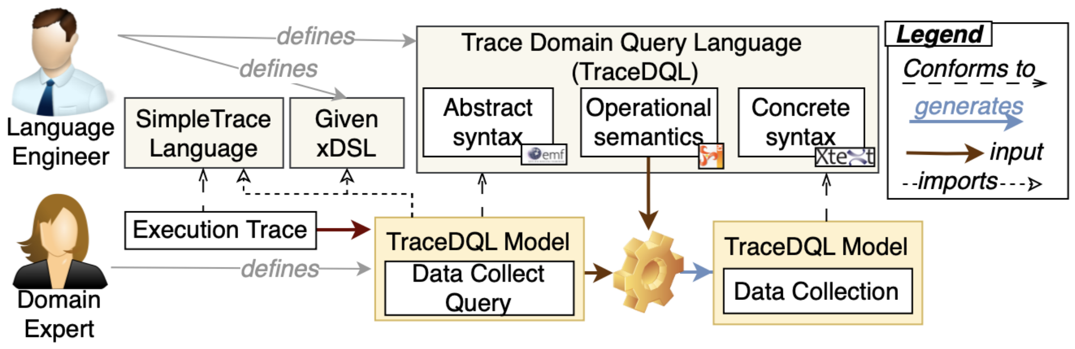


**Figure 7** Overview of the approach that provides query facilities for the domain expert to extract data from the execution trace.

out deep familiarity with the underlying trace infrastructure. Notably, the algorithm in Fig. 6 is tailored to answer a single specific query, and is tightly coupled to the SMS xDSL when navigating the ModelState. Moreover, the logic for traversing multiple ModelStates is interwoven and hard to isolate, making reuse and generalization difficult. This entanglement further increases the complexity for domain experts who must adapt or rewrite similar logic for each new query.

## 3. Overview of the Proposed Approach

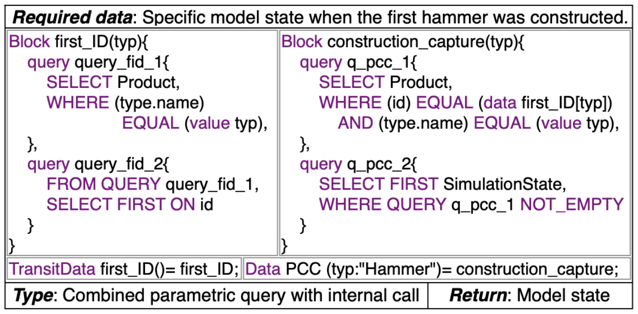
We propose an approach to support domain experts in analyzing execution traces without requiring them to manage the internal structure of the trace metamodel. The goal is to enable queries to be written in the expert's domain terminology, while abstracting away the complexity of the trace structure, particularly the navigation across multiple *ModelStates*, which is often tangled and query-specific, as previously illustrated.

At the heart of our approach lies a Trace Domain Query Language (*TraceDQL*), designed to express queries over execution traces in a high-level, declarative manner. This language provides specialized operators tailored to trace analysis needs, such as querying across temporal states or filtering elements based on dynamic model behavior.

While each execution trace conforms to a specific trace language, *TraceDQL* is designed as a generic and reusable query language. The domain expert does not have to write in the queries the parts navigating the trace structure. This task is managed by *TraceDQL* by analyzing the trace language structure and extracting the relevant parts automatically.

Fig. 7 illustrates the overall approach. On the bottom left, an execution trace is generated via the trace generation process (as previously depicted in Fig. 4). The domain expert then imports both the SimpleTrace language and the given xDSL into a *TraceDQL* model, referred to as a *Data Collect Query*. She also provides the path to the corresponding execution trace. Using the abstract and concrete syntaxes of *TraceDQL*, the expert defines queries that reflect their information needs.

Finally, these queries are executed through the operational semantics of *TraceDQL*. The result is a new model, named *Data Collection*, which contains the extracted and structured data. This output is ready for interpretation (for instance, numerical data can be used to compute Key Performance Indicators (KPIs), or for further analysis, such as applying *OCL* queries if the data are *subtraces*).



***Required data***: Specific model state when the first hammer was constructed.

```
Block first_ID(typ){                       Block construction_capture(typ){
  query query_fid_1{                         query q_pcc_1{
    SELECT Product,                            SELECT Product,
    WHERE (type.name)                          WHERE (id) EQUAL (data first_ID[typ])
            EQUAL (value typ),                   AND (type.name) EQUAL (value typ),
  },                                         },
  query query_fid_2{                         query q_pcc_2{
    FROM QUERY query_fid_1,                    SELECT FIRST SimulationState,
    SELECT FIRST ON id                         WHERE QUERY q_pcc_1 NOT_EMPTY
  }                                          }
}                                          }
TransitData first_ID()= first_ID;          Data PCC (typ:"Hammer")= construction_capture;
```

***Type***: Combined parametric query with internal call | ***Return***: Model state

**Figure 8** An overview of *TraceDQL* concrete syntax

## 4. TraceDQL: Trace Domain Query Language

This section describes the Trace Domain Query Language (*TraceDQL*) by going through its two parts: abstract syntax and operational semantics. Fig. 8 provides a glimpse of the *TraceDQL* concrete syntax. It is a textual syntax implemented with the Xtext[7] framework. The complete textual syntax is permanently accessible via this link (Ajabri et al. 2025).

### 4.1. *TraceDQL* abstract syntax

Fig. 9 illustrates an excerpt from the abstract syntax of *TraceDQL*. We define the metaclass *Package* as a root class of the *TraceDQL* abstract syntax. A *Package* can only be contained in one super-package, but it can contain a set of sub-packages and a set of *ImportedResources* (i.e., *TraceDQL* models that may be imported using their packages qualified names). The *SimpleTrace* metaclass is a subtype of the *ImportedResource* metaclass, indicating that the imported resource is the SimpleTrace language. The *Package* defines a *tracePath* attribute to specify the path to the execution trace on which the queries will be executed and it has a containment reference to the metaclass *Block*. A *Block* has a number of parameters defined by the metaclass *Parameter*, and a set of queries represented by the metaclass *Query*. *Parameter* has a value attribute, along with the *Block*, the *Query* and the *Package* metaclasses, it inherits a name attribute from *NamedElement*. Fig. 8 presents an example

[7] https://eclipse.dev/Xtext/index.html

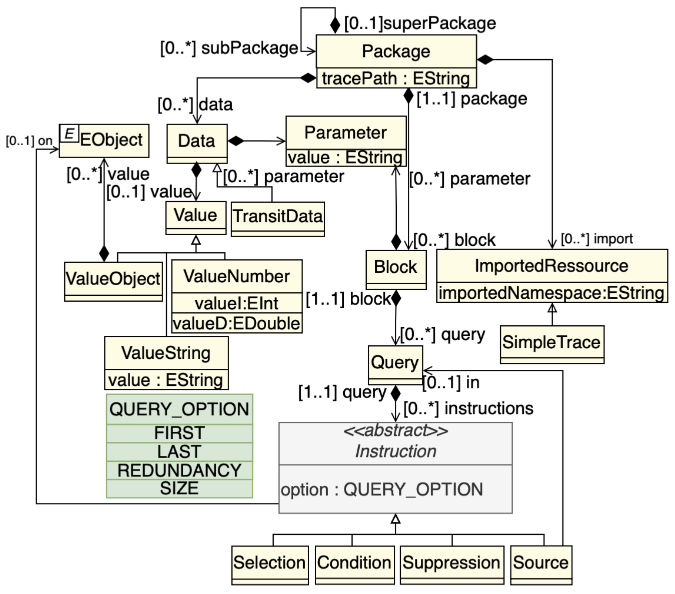


**Figure 9** *TraceDQL* abstract syntax excerpt, this part depicts the structure of a *TraceDQL* query and the structure that holds the values of the evaluated queries.

of a *Block* named *construction_capture*. This *Block* contains two queries and it includes a *Parameter* called *typ*. The value of the *typ* parameter is assigned when the *Block* is invoked by a *Data* element. In this example, the parameter is assigned the value *Hammer*.

A *Query* consists of a number of instructions defined by the metaclass *Instruction*. An instruction may be an instance of *Source*, *Selection*, *Condition* or *Suppression* metaclasses. An instruction may be applied to an *EClass* and/or an *EStructuralFeature* using the *on* reference. The *EClass* and the *EStructuralFeature* are metaclasses of the Ecore metamodel. More precisely, the *Selection* instruction may be applied to an *EClass* or an *EStructuralFeature*, whereas the *Source* is applied only to an *EClass*, and *Suppression* and *Condition* are applied solely to an *EStructuralFeature*. The instructions within the query are ordered in a way that the *Source* always comes first, if it exists, following by the *Selection*, whereas the *Condition* and *Suppression* do not follow a specific order. Each instruction may be used with an operator given by the *Query_Option* enumeration. More precisely, *FIRST* and *LAST* options may be used with the *Source* and the *Selection*, the *SIZE* option may be applied with the *Selection*, whereas *REDUNDANCY* is left for the *Suppression*. The *Source* may be applied to another query, within the same block or reuse another query declared in another block, through the *in* reference. With this reference, the queries are linked within the *Block*, i.e., a query references its previous query. The *Source* may also accept the output of another parametric computed block (i.e., expressed later with *Data*). This allows one to select instances of an *EClass* or an *EStructuralFeature* from the output of another query. Fig. 8 shows two examples of *TraceDQL Query*. The first *Query* (1), named *query_fid_1*, includes a *Selection* instruction on the *Product EClass* from the SMS xDSL metamodel, along with a *Condition* instruction. The second *Query* (2), named *query_fid_2*, selects the first value of a specific *EStructuralFeature*, i.e. *id*, from the first query using the *Source* instruction.

For the sake of conciseness, the corresponding part in the abstract syntax describing the structure of the *Condition* is not shown in a figure, but it is available in the implementation (Ajabri et al. 2025). The *Condition* is considered to be a logic expression. This expression may be a composition of other expressions with logic operators, or it may be a comparison between the value of an *EStructuralFeature* with a number, or a value of a parameter or the output value of an arithmetic expression between a set of *Block(s)* (i.e., expressed later with *Data*). The expression may also use primitive operators (such as *EMPTY* and *NOT_EMPTY*) on an *EStructuralFeature*.

To store and expose the results of the computed queries, the *TraceDQL* abstract syntax, in Fig. 9, defines the *Data* metaclass, for which the *Package* has a containment reference. The *Data* metaclass is a super type of *TransitData* metaclass. A *TransitData* is specifically designed to be invoked within a *TraceDQL Block*. If the *TransitData* defines parameters, their values are provided by the corresponding parameter values of the calling block. In Fig. 8, the *TransitData* named *first_ID* is invoked within the *construction_capture Block*, illustrating this behavior. The *Data* inherits a name attribute from *NamedElement*, and it has two containment references, one to *Value* metaclass and the other to *Parameter* metaclass. In the present paper, we limit a *Value* to three types, a string value represented by the *ValueString* metaclass, a number value appearing in instances of *ValueNumber* metaclass and finally the *ValueObject* metaclass that has a value containment reference to an external object from the Ecore. As explained in Subsection 2.4, this kind of references allows instances of metaobjects of a given xDSL to be included. The *Block* metaclass and the *Data*, are connected in a way that the *Data* may be connected to one *Block* to store its return value, or it may be associated to an arithmetic expression between a set of *Block*, each has a number as the return value. More details are provided at this link (Ajabri et al. 2025).

Before defining their operational semantics, we defined a few rules to ensure that queries are well-formed. The well-formedness conditions are different for *Source*, *Selection*, *Condition* and *Suppression*, so we defined four different rules for them.

***Instruction validation rule: Source Instruction example (isValidSrc).*** A *Source* instruction is valid if it is applied to an *EClass* that is not null.

```
If     src isInstanceOf Source
       src.on isInstanceOf EClass
       src.on ≠ null
then
       isValidSrc(src) = true
```

We have defined three other similar rules for the *Selection*, *Condition* and *Suppression* instructions, which are respectively *isValidSel*, *isValidCond* and *isValidSuppr*.

***Query validation rule (isValidQ).*** A query made of several instructions is valid if it is made of one selection instruction ($i_j$), at most one source instruction ($i_k$), and all the other instructions

(Condition/Suppression) are applied to the *EStructuralFeature* of the selected *EClass*. Particularly for the *Condition*, if this feature is an *EReference*, the instruction can also be applied transitively to the *EStructuralFeature* of the referenced *EClass*, and this process can continue recursively for each subsequent *EReference*. The function GETTRANSITIVECOMPONENTS gathers all these *EStructuralFeatures*.

```
If      $i_{i \in 1 \cdots n}$ isInstanceOf Instruction
        query = $[i_1;i_2 \ldots i_n]$
        $\exists! \ j \in 1 \cdots n \ . \ i_j$ isInstanceOf Selection
        sources = [ $i_k$ isInstanceOf Source | $k \in 1 \cdots n$ ]
        sources.size $\leq 1$
        comp = GETTRANSITIVECOMPONENTS($i_j$.on)
        $\forall k \in 1 \cdots n \mid k \neq j \wedge i_k \notin$ sources
           $\Longrightarrow$ EStructuralFeature of $i_k \in$ comp
then
        isValidQ(query) = true
```

***Block validation rule (isValidB).*** A block made of several queries is valid only if the queries do not have the same name, and they do not have the same previous query.

```
If      $q_{i \in 1 \cdots n}$ isInstanceOf Query
        block = $[q_1;q_2 \ldots q_n]$
        $\forall i, j \in 1 \cdots n . i \neq j \implies q_i.name \neq q_j.name$
        $\forall i, j \in 1 \cdots n . i \neq j \implies q_i.previous \neq q_j.previous$
then
        isValidB(block) = true
```

## 4.2. *TraceDQL* operational semantics

The *TraceDQL* operational semantics allows the computation of *TraceDQL* queries over the execution trace. We use Kermeta3[8] to implement the *TraceDQL* operational semantics. We exploit the OCL as a query language, and we transform the *TraceDQL* queries into OCL queries. In MDE, OCL is a standard query language widely utilized in both academic and industrial projects. OCL is well supported and considered a basis for a family of languages (EOL[9], QVT[10] and ATL)(Akehurst et al. 2005) and GEMOC Studio environment.

***Instruction OCL building rule (buildOCL).*** An OCL expression associated with a given *Instruction* is built only if this instruction is valid. The function TRANSLATEOCL performs this building, and it takes the *Instruction* as an argument. Accordingly, the resulting expression is an OCL expression that, in the case of the *Source*/*Selection* instruction, will return the instances of the *Source*/*Selection EClass* or the value of the *Selection EStructuralFeature* when applied to an execution trace. In other cases, the resulting OCL expression will filter/remove instances of the *Selection EClass* based on the value of its *EStructuralFeature*.

```
If      ins isInstanceOf Instruction
        isValidI(ins)
        exprOCL = TRANSLATEOCL(ins)
then
        buildOCL(ins) = exprOCL
```

[8] https://diverse-project.github.io/k3/
[9] https://eclipse.dev/epsilon/doc/eol/
[10] https://www.omg.org/spec/QVT/1.3/About-QVT

***Query OCL building rule (buildOCL).*** An OCL query associated with a given *TraceDQL Query* is built only if this *Query* is valid. The buildOCL function is applied to the *TraceDQL Query* instructions to return their corresponding OCL expressions. Then, the resulting OCL expressions are concatenated to obtain the OCL query.

```
If      $i_{i \in 1 \cdots n}$ isInstanceOf Instruction
        query = $[i_1;i_2 \ldots i_n]$
        isValidQ(query)
        exprOCL$_{i \in 1 \cdots n}$= buildOCL($i_i$)
        queryOCL= [exprOCL$_1$;exprOCL$_2$ ... exprOCL$_n$]
then
        buildOCL(query) = queryOCL
```

***Block OCL building rule (buildOCL).*** In order to build the OCL query associated with a given *TraceDQL Block*, the buildOCL function is applied to the *TraceDQL Querys* within this block to return their corresponding OCL queries. Then, considering the order in which the *TraceDQL Querys* are linked, the resulting OCL queries are concatenated to obtain the OCL query.

```
If      $q_{i \in 1 \cdots n}$ isInstanceOf Query
        block = $[q_1;q_2 \ldots q_n]$
        exprOCL$_{i \in 1 \cdots n}$= buildOCL($q_i$)
        queryOCL= [exprOCL$_1$;exprOCL$_2$ ... exprOCL$_n$]
then
        buildOCL(block) = queryOCL
```

***Block evaluation rule (evaluateB).*** A block of queries is evaluated when it is a valid block, and then it results in the evaluation of the OCL query obtained after the block OCL building. We specify the execution trace, and we use a predefined function EVALUATEOCL to evaluate the OCL query over this trace.

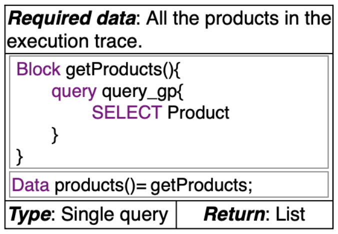
```
If      $q_{i \in 1 \cdots n}$ isInstanceOf Query
        block = $[q_1;q_2 \ldots q_n]$
        isValidB(block)
        executionTrace isConformsTo SimpleTrace
        queryOCL = buildOCL(block)
        obj = EVALUATEOCL(executionTrace, queryOCL)
then
        evaluateB(block) = obj
```

***Data evaluation rule (evaluateD).*** The evaluation of the data in a collection of blocks results in the evaluation of the arithmetic expression with the return values of the blocks, using COMPUTEARITHMETICEXPRESSION function. In case the return value of one block is not a number, this value is returned if it was the only block, otherwise the function returns *null*.

```
If      $b_{i \in 1 \cdots n}$ isInstanceOf Block
        data = $[b_1;b_2 \ldots b_n]$
        $v_{i \in 1 \cdots n}$ = evaluateB($b_i$)
        obj = COMPUTEARITHMETICEXPRESSION($v_{i \in 1 \cdots n}$)
then
        evaluateD(data) = obj
```

## 4.3. OCL query generation

Any query integrates a trace structure that itself contains the model state elements. Therefore, the translation of a query should preserve this global schema. Our translation into OCL then consists of (1) a first step to extract the *TraceStructure*, (2) a second step to navigate through the *ModelState*.



**Required data**: All the products in the execution trace.

```
Block getProducts(){
      query query_gp{
            SELECT Product
      }
}
Data products()= getProducts;
```

**Type**: Single query | **Return**: List

**Figure 10** A *TraceDQL* query to return all the instances of *Product* existing in the Hammer SMS execution trace (see Fig. 3).

For example, given the Hammer SMS execution trace illustrated in Fig. 3, a *TraceDQL* query that retrieves *Product* instances from the trace (see Fig. 10), is translated into a two-part OCL query. The first part, which navigates over the *TraceStructure*, is shown in Listing 1. The second part, which navigates over the *ModelState*, is shown in Listing 2. The first part is concatenated with the second part to form the final OCL query.

```
#OCL query to navigate over the TraceStructure.
self.states
->selectByKind(simple::RuntimeState)
.runtimeExtensions
->selectByKind(simple::
    RuntimeExtensionOfStaticElement)
.runtimeBindings
->selectByKind(simple::RuntimeObjectValueBinding)
.runtimeValue
->selectByKind(simple::RuntimeContainmentValue)
.runtimeObject
```

**Listing 1** OCL query for navigating the TraceStructure.

```
#OCL query to navigate over the ModelState.
->selectByKind(runtime::SimulationState)
.moduleState
->selectByKind(runtime::ContainerState)
.products
->selectByKind(runtime::Product)
```

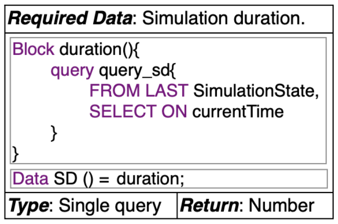
**Listing 2** OCL query for navigating the ModelState.

## 5. Application

This section illustrates the feasibility of our proposal on an advanced version of the SMS xDSL and on an Arduino xDSL. *TraceDQL* is put to the test in terms of ease of use, expressiveness and efficiency. For this purpose, we target some usual questions that domain experts raised when examining an execution trace. Among these questions are:

– the duration of a simulation,
– the quantity produced for a given item,
– the absence, occurrences and frequency of some specific event in their systems.

We manually validate the results of the *TraceDQL* queries, either by directly checking the value of the output in the execution trace or by verifying the generated OCL query.

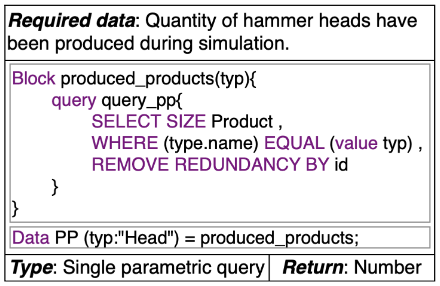


**Required Data**: Simulation duration.

```
Block duration(){
      query query_sd{
            FROM LAST SimulationState,
            SELECT ON currentTime
      }
}
Data SD () = duration;
```

**Type**: Single query | **Return**: Number

**Figure 11** A *TraceDQL* query to extract the simulation duration from the Hammer SMS model.



**Required data**: Quantity of hammer heads have been produced during simulation.

```
Block produced_products(typ){
      query query_pp{
            SELECT SIZE Product ,
            WHERE (type.name) EQUAL (value typ) ,
            REMOVE REDUNDANCY BY id
      }
}
Data PP (typ:"Head") = produced_products;
```

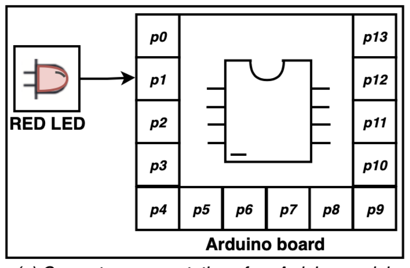
**Type**: Single parametric query | **Return**: Number

**Figure 12** A *TraceDQL* query to count the quantity of produced hammer heads in the execution trace of the Hammer SMS model.

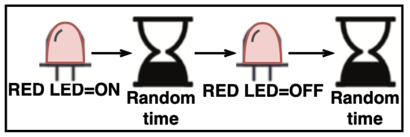
### 5.1. Case study 1: *TraceDQL* to query SMS xDSL models execution traces

The main scenario is as follows: a domain expert, in order to study her system, has traces generated during a simulation; she then recovers and analyses them. Prior to the simulation of the system model, a language engineer provides the SMS xDSL to design the model, with the required annotation; the *TraceDQL* is available and the related components to manipulate models and traces are available. She is able then to apply the approach depicted in Fig. 7. The domain expert defines a new *TraceDQL* model, i.e. the *Data Library Getter*, she imports the SimpleTrace language and the SMS xDSL, and provides the path to the execution trace. She may then write different queries to extract the data from the execution trace, using the terminology of the SMS xDSL domain (e.g. *SimulationState*, *Product*, *type*, etc.). The following are examples of *TraceDQL* queries that answer the questions previously invoked in Subsection 2.3. Note that we selected among the most common questions asked by experts when simulating production systems. For example, the queries in Figures 11 and 12 may assist in computing the throughput KPI for the production system, whereas the query in Fig. 8 returns a complete state capture of the model, which may be useful for diagnostic purposes.

– The *TraceDQL* query of Fig. 11: this query allows extracting the simulation duration, which corresponds to the last capture of the time, since the simulation starts at time 0. Thus, the simulation duration is the *currentTime* attribute value of the last *SimulationState*. This query is a *Single TraceDQL* query, where the *TraceDQL Block* contains one *TraceDQL Query*. The return type is a number.



(a) Concrete representation of an Arduino model.



(b) Behavioral representation of an Arduino model.

**Figure 13** An example illustrating both the concrete and behavioral representations of an Arduino model with a red *LED* plugged on an Arduino *Board*.

– The *TraceDQL* query of Fig. 12: this query allows counting the quantity of hammer heads produced during the simulation. It selects the *SIZE* option on *Product* to count the quantity of product instances, after filtering them considering their type, and eliminating duplicates relying on their *id*. This query is a *Single TraceDQL* query, parametrized with the product type; the requested type in this case is the *Head*. The return type is a number.

– The *TraceDQL* query of Fig. 8: this query (*construction_capture* Block) allows retrieving the capture of the model state when the first hammer was performed. To do so, another *TraceDQL* query, named *first_ID*, is defined to retrieve the first *id* of the *Products* by type. Then, a first *TraceDQL Query* (*q_pcc_1*) filters the *Products* considering their *id* and type. The *id* must be equal to the return value of the *first_ID* of the considered type. The output is the first product of the specific type. A second *TraceDQL Query* (*q_pcc_2*) selects the first *SimulationState* that contains the first product retrieved; this *SimulationState* corresponds to the capture when the first product was performed. The query combines two *TraceDQL queries*. It is parametrized with the type of product and contains an internal call to another query (*first_ID*); the requested type is the *Hammer*. The return type is a model state.

## 5.2. Case study 2: *TraceDQL* to query Arduino models execution traces

***5.2.1. Arduino xDSL and Arduino model*** We consider an Arduino-based application made of an Arduino board, on which a single red *LED* is plugged. This *LED* is plugged to a *Pin* named *p1*. Fig. 13 (a) shows a representation of this model. The *LED* exhibits a blinking behavior, depicted in Fig. 13 (b), where it alternates between two states (i.e., ON and OFF) at random time intervals. When the value of *p1* is set to "0", the red LED is switched off. Conversely, when *p1* is set to "1", the *LED* lights up.

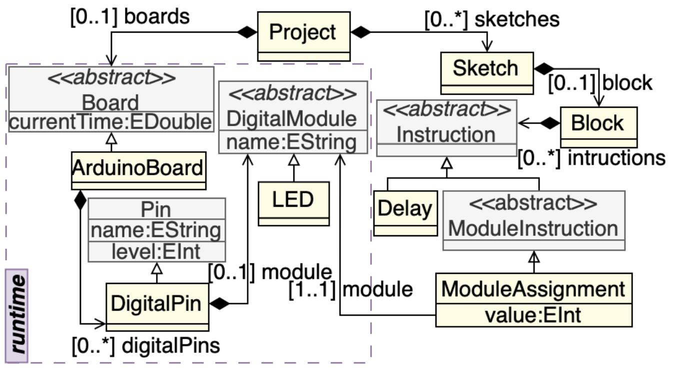


**Figure 14** Excerpt from the Arduino xDSL metamodel

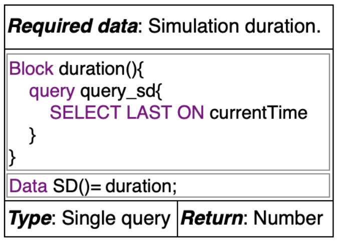
The domain expert would like to build a model of this Arduino application and simulate it to analyse the desired behavior. One of the questions he may ask about the behavior of the model, among others, is the percentage of time the *LED* was on or off during the simulation. For this reason, a specific language is required for building the model of the application. Therefore, for this second case study, we use the Arduino Designer[11] that was developed and provided as an Arduino xDSL. We apply *TraceDQL* querying facilities on this second (Arduino xDSL) language.

An excerpt from the Arduino xDSL metamodel is shown in Fig. 14. The full metamodel comprises 59 metaclasses. The *Project* metaclass contains a *Board* metaclass and a set of *Sketch* metaclass. The *Board* is the physical part of the Arduino system, it may contain a set of *DigitalPin* which has a *name* and a *level* attribute representing the signal level; the *DigitalPin* may contain a *DigitalModule* such as a *LED*. The *Sketch* is the logical part of the system, it contains a *Block* of *Instructions*. An instruction may be a *ModuleAssignment* or a *Delay*. A *ModuleAssignment* has a *module* reference to the *DigitalModule* and a *value* attribute specifying the value to assign to the *Pin* of the *Module*. A *Delay* specifies a random amount of time to wait.

Before generating the execution trace, we must consider annotating the runtime state definition of the Arduino xDSL metamodel. We consider the runtime state definition of the Arduino xDSL to include the *Board* that contains the *Pins* and the *LEDs* plugged on them. At each runtime state, the level of the pin is captured to indicate the state of the *LED*, the level value equals "0" meaning the *LED* is off, the level value equals "1" meaning the *LED* is on. We add a *currentTime* attribute at the *Board* to capture the real time in which the *LED* was on/off. This updated version of the Arduino xDSL is available in GitLab repository[12], and permanently available via this link (Ajabri et al. 2025).

***5.2.2. TraceDQL and Arduino xDSL*** Before the domain expert may apply the overall approach (see Fig. 7), the necessary prerequisites must be in place: The given xDSL, i.e. the Arduino language, is available, the Arduino model has been designed

[11] https://github.com/mbats/arduino
[12] https://gitlab.univ-nantes.fr/rodic/evaluationKPIxDSL/-/tree/master/Language_Workbench/xArduino



***Required data***: Simulation duration.

```
Block duration(){
   query query_sd{
      SELECT LAST ON currentTime
   }
}
```

```
Data SD()= duration;
```

***Type***: Single query ***Return***: Number

**Figure 15** A *TraceDQL* query to extract the simulation duration from an execution trace of an Arduino xDSL model.

and simulated, and the corresponding execution trace has been generated. In a new *Data Library Getter*, the domain expert imports both the SimpleTrace language and the Arduino xDSL, then she specifies the path to the generated execution trace. The following are examples of *TraceDQL* queries that the domain expert may write to extract data from the execution trace of the Arduino model (Fig. 13), using the terminology of the Arduino xDSL domain (e.g. *Board*, *DigitalPin*, *level*, etc.). These two query examples (Figures 15 and 16) may assist in calculating the percentage of time the *LED* was on/off, thereby answering the previous question (see Subsection 5.2.1).

– The *TraceDQL* query of Fig. 15: This query is similar to the one in Fig.11, which retrieves the simulation duration by selecting the final time capture. The key difference between them lies in their structure: the query in Fig.11 includes a *Source* instruction, while this one does not. *Both* ways are possible.
– The *TraceDQL* query of Fig. 16: this query allows extracting a list of captured timestamp when a particular *LED* was on/off. This query is a nested parametric *TraceDQL* query, where the *TraceDQL Block* consists of three *TraceDQL Query*: (1) The first applies a filter on the *LEDs*, which considers the name and level of the related *DigitalPin*. (2) The second considers the *Board* captures that satisfy the filter of the first *TraceDQL Query*, i.e., the *Board* captures which contain the particular *LED* on/off. (3) The third query selects the *currentTimes* from the output of the second. The return type is a List. This helps to compute the period in which the *LED* was on/off during the execution.

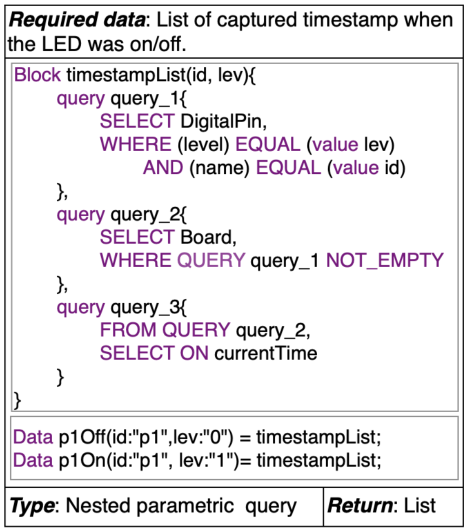
## 5.3. *TraceDQL* and reusability

The reusability aspect of our proposal appears in different ways:

***The use of parameters:*** A parametric query can be reused for new values of parameters. For example, considering the SMS xDSL case study, if the domain expert asks for the produced products of other types, e.g., the Hammer; we can compute these data by passing a new value of the *typ* parameter to the data of the *TraceDQL* query of Fig. 12:

```
Data PP (typ:"Hammer") = produced_products;
```

***The reuse of queries:*** A *TraceDQL Query* may be called from outside the *TraceDQL Block* in another *TraceDQL Block*. For example, considering the Arduino xDSL case study, we may define a new *TraceDQL* query that reuses the query *query_2* of the *TraceDQL* query in Fig. 16, as follows:



***Required data***: List of captured timestamp when the LED was on/off.

```
Block timestampList(id, lev){
      query query_1{
            SELECT DigitalPin,
            WHERE (level) EQUAL (value lev)
                  AND (name) EQUAL (value id)
      },
      query query_2{
            SELECT Board,
            WHERE QUERY query_1 NOT_EMPTY
      },
      query query_3{
            FROM QUERY query_2,
            SELECT ON currentTime
      }
}
```

```
Data p1Off(id:"p1",lev:"0") = timestampList;
Data p1On(id:"p1", lev:"1")= timestampList;
```

***Type***: Nested parametric query ***Return***: List

**Figure 16** A *TraceDQL* query to get the list of captured timestamp when the *LED* was on/off from an execution trace of an Arduino xDSL model.

```
Block timestampList_version_2(id,lev){
   query query_times{
        FROM QUERY timestampList.query_2,
        SELECT ON currentTime
   }
}
```

***The reuse of the outputs:*** The output of a query can be reused in another query as an input, depending on its type. For example, we consider a data that retrieves the quantity of hammer produced before a specific timestamp (e.g., 13 units of time). To compute this data, we first retrieve a subtrace corresponding to the model states recorded before that particular timestamp. This is achieved by using the following *TraceDQL* query:

```
Block subtrace(timestamp){
    query query_subtrace{
        SELECT SimulationState,
        WHERE (currentTime) LESS_THAN (value timestamp)
    }
}
```

Thereafter, we may update the *TraceDQL* query of Fig. 12 to compute the quantity of hammer produced products, relying on the output of the obtained subtrace.

```
Block produced_products(timestamp, typ){
   query query_pp{
        FROM DATA subtrace[timestamp],
        SELECT SIZE Product,
        WHERE (type.name) EQUAL (value typ),
        REMOVE REDUNDANCY BY id
   }
}
TransitData subtrace () = subtrace;
Data PP(timestamp:"13",typ:"Hammer")=produced_products;
```

## 6. Discussion and Assessment

In this section, we give an initial assessment of the results, and we examine the advantages and disadvantages of our proposal.

### 6.1. Assessment

Our proposal aimed to answer two research questions. **RQ#1** is about the genericity of a trace query language. We designed *TraceDQL* as a language that uses a domain xDSL as a parameter. Therefore, domain experts can use *TraceDQL* with various trace models. However, in this paper, we illustrate its applicability only with two xDSLs, the SMS xDSL and the Arduino xDSL; the experiments were conducted with the EMF framework and the Ecore metamodel; *TraceDQL* was succesfully used to handle traces from the model xDSL used as a parameter; which showcases the potential of our generic approach. Nevertheless, further experiments have to be done with a broader range of xDSLs and possibly with other implementation frameworks.

The second research question (**RQ#2**) targets *TraceDQL* querying facilities, which aims to evaluate the query structure and the reusability feature. We illustrated the use of *TraceDQL* through the SMS xDSL and Arduino xDSL case studies, which showcase the use of different application domains. We have also highlighted the reusability aspect of the *TraceDQL* queries in different ways. However, right now we have examined a limited set of data that may be extracted from the execution trace. We have considered covering different data types in the trace model, but this could lead to an extension of *TraceDQL*. We are postponing the compromise solution because we want to keep the syntax of *TraceDQL* easy to use.

***Expressivity*** The first case studies show that we are able to express with *TraceDQL*, the usual queries of domain experts without difficulties. The use of *TraceDQL* to navigate through the model execution traces was also easily tractable. The *TraceDQL* is translated into OCL to be executed over the execution trace, but it does not replace OCL, it rather complements it. *TraceDQL* is inspired by OCL query language and Structured Query Language (SQL), offering an intuitive syntax that incorporates OCL and SQL keywords, along with other self-explanatory terms, a syntax that uses domain terminology and close to natural language. The *TraceDQL* focuses on the navigation in traces (mainly by abstracting the trace structure, and providing operators to analyse and navigate through several runtime states over time). Once *TraceDQL* has been used, it is still possible to apply OCL to the returned model for further analysis.

***Description of TraceDQL execution times*** We have conducted first experiments on the evaluation of *TraceDQL* queries, focusing on their execution times considering various scalability metrics:

– Trace Size (in MB): it is the size of the execution trace in megabytes (MB).
– Model State Capture: it represents the number of model states recorded in the execution trace.
– Number of Object: it shows the number of objects gathered in the execution trace.

Table 1 presents preliminary results. These experiments were conducted using the SMS xDSL as the given xDSL. The execution traces were produced from an SMS xDSL model containing 30 objects. In each model state capture, 12 objects are consistently present, while other objects, i.e. instances of *Product* and *Task*, may dynamically appear or disappear between captures. We consider two execution traces:

– An execution trace, with a trace size of 3.316MB, captures 500 model states and contains 20,350 objects.
– An execution trace, with a trace size of 33.820MB, captures 5,000 model states and contains 133,975 objects.

To compute the number of objects that are dynamically present or absent in each execution trace, i.e., instances of *Product* and *Task*, we have used two *TraceDQL* queries illustrated in Fig. 17.

Different execution times are evaluated:

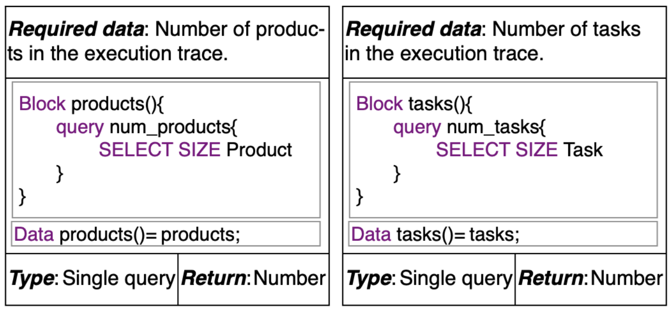
– $t_{load}$ represents the time required to load the trace;
– $t_{transform}$ refers to the time needed to transform the *TraceDQL* query into an OCL query;
– $t_{exec}$ is the time taken to execute the OCL query on the trace.
– Finally, $t_{total}$ denotes the overall time, it is the sum of the three previous execution times.

These time measurements are valuable for explaining the overall execution time of a *TraceDQL* query. They help to identify which part of the process is most time-consuming, and therefore where optimization efforts should be focused.

Three main types of *TraceDQL* queries are observed:

– *Single*, which refers to a *TraceDQL Block* containing a single *TraceDQL Query* ;
– *Nested*, where a *TraceDQL Block* contains multiple *TraceDQL Querys* (in this study, we evaluate a *TraceDQL Block* with four *TraceDQL Querys*); and
– *Combined*, where a *TraceDQL Block* (referred to as Q2) includes an internal call to another *TraceDQL Block* (referred to as Q1).

For the *Single* type, we evaluate three variations of *TraceDQL* queries: one containing only a *Selection* instruction (referred to as *S*); another containing both *Selection* and *Condition* instructions (referred to as *SC*); and a third containing *Selection* and *Suppression* instructions (referred to as *SR*). For the remaining types, the *TraceDQL* queries include *Selection* and/or *Condition* instructions.



***Required data***: Number of products in the execution trace.

```
Block products(){
    query num_products{
        SELECT SIZE Product
    }
}
Data products()= products;
```

***Type***: Single query ***Return***: Number

***Required data***: Number of tasks in the execution trace.

```
Block tasks(){
    query num_tasks{
        SELECT SIZE Task
    }
}
Data tasks()= tasks;
```

***Type***: Single query ***Return***: Number

**Figure 17** *TraceDQL* queries to compute numbers of products and tasks in an SMS model execution trace.

***Analysis of TraceDQL execution times*** Most query types demonstrate good scalability with different trace sizes. For example, the *Nested* type performs efficiently, with execution times remaining under one second, even if it comprises four *TraceDQL Queries*. The *Combined* type shows similar performance. The only *TraceDQL Query* that exhibits a performance limitation as trace size increases is the one containing the *Suppression* instruction. For the larger trace (5,000 model state captures), execution time $t_{exec}$ reaches 446 seconds, which is consistent with the O(n²) complexity of the *Suppression* instruction. Consequently, future work should focus on optimizing such *TraceDQL* instructions.

### 6.2. Discussion

It would be possible to use the Object Constraint Language (OCL)[13] queries directly on the execution trace without resorting to *TraceDQL* queries. The problem remains in the complexity of the OCL queries for the domain expert with poor programming skills. Besides, the *TraceDQL* suggests instructions that embed logics and shorten a long OCL query.

For example, considering the execution trace of the Hammer SMS model (see Fig. 3). An example of data to extract from this execution trace is the quantity of hammer heads generated during the simulation (see algorithm in Fig. 6). However, the same hammer head is captured several times (see Fig. 1). Each capture is a new instance. Therefore, to remove duplicates and keep only one instance for each hammer head when counting their quantity, a simple *asSet()* OCL operation is not sufficient. One way to remove this redundancy, using OCL, may be found in Listing 3. In this example, the variable *c* represents the OCL query that retrieves *Product* instances from the execution trace. Its two parts are illustrated in Listings 1 and 2.

```
#OCL query to remove redundancy of products by id
c->iterate(p;product:Sequence(runtime::Product)=c
   | let idList:Sequence(Integer) = product
   ->iterate(pr;acc:Sequence(Integer)=Sequence{}
     | acc->including(pr.id))
   in if(idList->count(p.id)>1)
      then product->excluding(p)
      else product
      endif )
->size()
```

**Listing 3** OCL query to remove redundancy of instances of *Product* relying on the value of the *id* attribute.

Two OCL queries are identified to extract the required data: (1) the OCL query *c* to select *Product* instances (Listings 1 and 2), and (2) the OCL query to remove redundancy (Listing 3). The OCL query *c* implements the two parts (1 and 2) of the algorithm in Fig. 6, whereas the OCL query to remove redundancy implements the rest of the algorithm (part 3) dedicated to exclude the products duplicated by their *id* when counting. The OCL query to remove redundancy is complex and it becomes larger as much as the OCL query *c* becomes larger. Therefore, the domain expert may use two *TraceDQL* instructions, using the domain terms, instead of the two OCL queries: (1) `SELECT SIZE` *Product*, (2) `REMOVE REDUNDANCY BY` *id*. Note that these two *TraceDQL* instructions with their options are internally transformed into OCL queries in Listings 1, 2 and 3.

Furthermore, asking the domain experts to directly write OCL queries to query the execution trace forces them to understand the trace structure. In addition, this method does not offer any reusability aspect, each time the domain experts have a new request for data, they will have to rewrite either a portion or the entire OCL query.

***Scalability*** The scalability aspect has been a major concern when dealing with trace generation (Freitag et al. 2002). The efficiency of trace generation impacts the scalability of the approaches. In this work, we do not deal directly with the generation of the execution trace, we reuse existing components, on which our scalability depends. In the same way, we inherit from the scalability and performance of OCL querying tasks for large execution traces. This has been discussed (Chimiak_Opoka et al. 2008), and some solutions have been provided (Kolovos et al. 2013; Daniel 2016). But, the frameworks that we have chosen for the implementation of *TraceDQL*, either the EMF framework or the OCL framework, are all frameworks that are widely used and well-researched with very active communities. This is important to tackle scalability aspect in future work.

## 7. Related Work

There have been a considerable amount of research that was considering extracting important data from an execution trace. A prominent work in this field was the proposal of Goldsmith et al. (Goldsmith et al. 2005); they have suggested a query language *Program Trace Query Language (PTQL)*, that exempts the developer from manually instrumenting the program to understand program behavior. The *PTQL* allows the developer to write expressive and declarative queries to answer her/his question about the program. The article suggests a compiler that instruments the program automatically and dynamically to generate a "runtime trace" based on what the query required. The *PTQL* queries are supported by a data model based on the structures of a program: class, method, and parameters etc. As a result, *PTQL* queries are restricted to this program structrure. It is not the case for *TraceDQL*, where the queries are structured with a specific domain terminology which is related to the parameter used for *TraceDQL*. *PTQL* helps developers to better understand the structure and behaviour of their program. However, it has a sophisticated syntax, and therefore unlike *TraceDQL*, it requires significant programming skills.

In their work, Dou et al. (DOU et al. 2014) proposed to query an offline trace considering a model-driven approach with OCL. The article presents *TemPsy*, a language for the specification of temporal requirements of business processes. Here, it is an offline process based on execution traces. These traces conform to a conceptual model and capture events that occur during the considered business process. They implement a trace checking procedure where the temporal requirements written in *TemPsy* are mapped into OCL constraints on execution traces; the latter are then evaluated with an OCL cheker. Mainly, regarding trace checking, compared to *TraceDQL* which has a state-based trace model, the trace checking approach of *TemPsy*

[13] https://www.omg.org/spec/OCL/

| Trace Size (in MB) | Model State Capture | Number of Objects | | Single | | | Nested | Combined | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | S | SC | SR | (4 queries) | Q1 | Q2 |
| 3.316 | 500 | 20350 | $t_{load}$ | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | |
| | | | $t_{transform}$ | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s |
| | | | $t_{exec}$ | $<$ 1s | $<$ 1s | 179s | $<$ 1s | $<$ 1s | $<$ 1s |
| | | | $t_{total}$ | $<$ 1s | $<$ 1s | 179s | $<$ 1s | $<$ 1s | |
| 33.820 | 5000 | 133975 | $t_{load}$ | 1s | 1s | 2s | 1s | 1s | |
| | | | $t_{transform}$ | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s | $<$ 1s |
| | | | $t_{exec}$ | $<$ 1s | <1s | 446s | $<$ 1s | $<$ 1s | <1s |
| | | | $t_{total}$ | 1s | 1s | 450s | 1s | $<$ 1s | |

**Table 1** Execution times (in seconds) for different types of *TraceDQL* queries on two trace sizes.

is event-based and requires less works as it relies on OCL cheker. However *TraceDQL* offers more trace navigation facilities. Indeed, *TraceDQL* queries navigate through traces that include states filled with data, in order to offer greater freedom of interrogation to domain experts. *TraceDQL* may be used to check trace properties as *TempPsy*, but can also be extended to compute performance indicators (KPI), for diagnostics, etc. This possibility of using *TraceDQL* in multiple contexts is supported also by its parametric aspect.

In the literature, there are languages designed to query specific models rather than traces. For instance, Kling et al. (Kling et al. 2012) suggested a textual domain-specific language *MoScript*, for querying mega-models. It allows writing and executing queries on model repositories to retrieve model, information, and to manipulate them, and if necessary generated results and store them in the repository. *MoScript* uses the OCL query language by extending its abstract syntax. The syntax of *MoScript* is an OCL syntax and its queries are applied to the mega-model domain-specific language, hence such language cannot be manipulated with modest programming skills. Another example is the one introduced in (Deutsch et al. 1999). This article proposes a query language for eXtensible Markup Language (XML) specific language, named *XML-QL*. The syntax of this language combines query operators with XML fragments. The language consists of two parts, one for querying and retrieving the results and the other for storing the result in a new XML file. Although the *XML-QL* queries allow the use of XML-related elements with XML syntax, writing these queries is not a simple undertaking and requires a good understanding not only of *XML-QL* but also of XML. This language is devoted to XML and cannot be used in another language context.

Other research has mentioned querying the trace as an auxiliary work of their main work, which was based on constructing the trace. Andjelkovic et Artho (Andjelkovic & Artho 2011) had built a trace of the program execution; their approach suggest to store the trace information in an existing database (e.g., neo4j database) and the query part was delegated to the query mechanism of the database. Montplaisir et al. (Montplaisir et al. 2013) work was devoted more to trace the behavior of a system based on its states not the occurred events. But states and their history are constructed from the traces of events, made up of various changes, previously stored. The part of the article that discusses querying the trace is rather poor. The authors have explained how to query data from the trace in both tracing mode, online and offline, and mentioned that the procedure requires the timestamp and the key of the data (i.e., the key is computed using the multilevel hash-map mechanism) since the trace has been organized in a tree structure. Therefore, their approach cannot provide a query capability when the required data is the timestamp where specific conditions are met. The *TraceDQL* querying facilities allows extracting such data which are relevant in many cases, from the execution trace; an example may be found in Fig. 16.

## 8. Conclusion and Future Work

This paper examines how domain experts can extract relevant data from an execution trace to fulfill many purposes. For that reason, we have studied the parametric aspect and the structure of an execution trace, and we have introduced an approach that assists the domain experts to write queries with an expressive syntax close to the natural language, which incorporates OCL and SQL keywords, along with self-explanatory terms. The approach provides the parametric language *TraceDQL* with textual facilities and operational semantics to ensure that domain experts may write queries using the domain terminology and without the need to manage the entire structure of the execution trace, which may include elements not related to their specific domain. Our proposal allows querying an execution trace for which the executed model conform to a given xDSL. We illustrate the feasibility of the proposal with two cases: the SMS xDSL and the Arduino xDSL.

In the present article, we have limited our research to the execution trace generated by the *SimpleTrace* framework. We are also considering the use of *TraceDQL* to extract data needed for on-demand KPI computation. The idea here is an automatic generation and execution of *TraceDQL* queries to provide specific data for the KPI computation process. Among the future work, we plan to investigate a way to generalize our approach and *TraceDQL* operational semantics to consider any given trace language provided that it met given criteria, such as strong typing, hierarchical relationship between object types, etc. This will open the possibility to use *TraceDQL* queries on other generated execution traces.

### Acknowledgments

This work was supported by the French National Research Agency (ANR)[grant number ANR 21 CE10 0017].

## References

Ajabri, H., Mottu, J.-M., Attiogbe, C., & Berruet, P. (2025). *Tracedql tool on zenodo.* Retrieved from https://doi.org/10.5281/zenodo.15183458 (Accessed: 2025-04-11)

Ajabri, H., Mottu, J.-M., & Bousse, E. (2024). Defining KPIs for Executable DSLs: A Manufacturing System Case Study. In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering.* Rome, Italy: SCITEPRESS - Science and Technology Publications. doi: 10.5220/0012361000003645

Akehurst, D. H., Howells, G., & McDonald-Maier, K. D. (2005). Supporting OCL as part of a Family of Languages. In *Proceedings of the MoDELS* (Vol. 5).

Alawneh, L., & Hamou-Lhadj, A. (2009). Execution Traces: A New Domain That Requires the Creation of a Standard Metamodel. In *Advances in Software Engineering* (Vol. 59). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-10619-4_31

Aljamaan, H., Lethbridge, T. C., Badreddin, O., Guest, G., & Forward, A. (2014). Specifying trace directives for UML attributes and state machines. In *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD).*

An, K., Trewyn, A., Gokhale, A., & Sastry, S. (2011). Model-Driven Performance Analysis of Reconfigurable Conveyor Systems Used in Material Handling Applications. In *2011 IEEE/ACM Second International Conference on Cyber-Physical Systems.* doi: 10.1109/ICCPS.2011.12

Andjelkovic, I., & Artho, C. (2011). Trace server: A tool for storing, querying and analyzing execution traces. In *JPF Workshop 2011.*

Bousse, E., Combemale, B., & Baudry, B. (2014). Towards scalable multidimensional execution traces for xDSMLs. In *11th Workshop on Model Design, Verification and Validation Integrating Verification and Validation in MDE.*

Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., & Combemale, B. (2016). Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering.* Amsterdam Netherlands: ACM. doi: 10.1145/2997364.2997384

Chimiak_Opoka, J., Felderer, M., Lenz, C., & Lange, C. (2008). Querying UML Models using OCL and Prolog: A Performance Study. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop.* doi: 10.1109/ICSTW.2008.33

Cleenewerck, T., & Kurtev, I. (2007). Separation of concerns in translational semantics for DSLs in model engineering. In *Proceedings of the 2007 ACM symposium on Applied computing.* Seoul Korea: ACM. doi: 10.1145/1244002.1244218

Combemale, B., Cregut, X., & Pantel, M. (2012). A Design Pattern to Build Executable DSMLs and Associated V&amp;V Tools. In *2012 19th Asia-Pacific Software Engineering Conference.* Hong Kong, China: IEEE. doi: 10.1109/APSEC.2012.79

Daniel, G. (2016). Efficient Persistence and Query Techniques for Very Large Models. In *ACM Student Research Competition (MoDELS'16).* Saint-Malo, France.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1999). A query language for XML. *Computer Networks*, *31*. doi: 10.1016/S1389-1286(99)00020-1

DOU, W., BIANCULLI, D., & BRIAND, L. (2014). *A Model-Driven Approach to Offline Trace Checking of Temporal Properties with OCL* (Tech. Rep.). SnT Centre - University of Luxembourg.

Ezzati-Jivan, N., & Dagenais, M. R. (2012). A Stateful Approach to Generate Synthetic Events from Kernel Traces. *Advances in Software Engineering*, *2012*. doi: 10.1155/2012/140368

Freitag, F., Caubet, J., & Labarta, J. (2002). On the Scalability of Tracing Mechanisms. In *Euro-Par 2002 Parallel Processing* (Vol. 2400). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/3-540-45706-2_10

Goldsmith, S. F., O'Callahan, R., & Aiken, A. (2005). Relational queries over program traces. *ACM SIGPLAN Notices*, *40*. doi: 10.1145/1103845.1094841

Kaiser, B., Reichle, A., & Verl, A. (2022). Model-based automatic generation of digital twin models for the simulation of reconfigurable manufacturing systems for timber construction. *Procedia CIRP*, *107*. doi: 10.1016/j.procir.2022.04.063

Kling, W., Jouault, F., Wagelaar, D., Brambilla, M., & Cabot, J. (2012). MoScript: A DSL for Querying and Manipulating Model Repositories. In *Software Language Engineering* (Vol. 6940). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-28830-2_10

Kolovos, D. S., Wei, R., & Barmpis, K. (2013). An approach for efficient querying of large relational datasets with ocl-based languages. In *XM 2013–Extreme Modeling Workshop.* Citeseer.

Lübke, D., & Van Lessen, T. (2017). BPMN-Based Model-Driven Testing of Service-Based Processes. In *Enterprise, Business-Process and Information Systems Modeling* (Vol. 287). Cham: Springer International Publishing. doi: 10.1007/978-3-319-59466-8_8

Montplaisir, A., Ezzati-Jivan, N., Wininger, F., & Dagenais, M. (2013). Efficient Model to Query and Visualize the System States Extracted from Trace Data. In *Runtime Verification* (Vol. 8174). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-40787-1_13

Plotkin, G. D. (2004). The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, *60-61*. doi: 10.1016/j.jlap.2004.03.009

## About the authors

**Hiba Ajabri** is a Software Engineer and currently a Ph.D. student in the NaoMod team at IMT Atlantique, and the Velo team at Nantes University (France). She is also involved in the RODIC ANR project that consists of assisting the operator of the industry in the reconfiguration of production lines. Her current research targets the performance evaluation using Model-Driven Engineering, Domain-Specific Languages (DSLs) and Query languages. You can contact the author at hiba.ajabri@ls2n.fr or visit https://hiba-coder.github.io.

**Jean-Marie Mottu** is an Associate Professor in Computer Science at Nantes University, where he researches and teaches software engineering. His research interests include MDE, low-code engineering, domain-specific languages, software quality, and test verification, focusing on both functional and non-functional properties. He is a member of the NaoMod group within the LS2N research laboratory.You can contact the author at jean-marie.mottu@ls2n.fr or visit http://pagesperso.ls2n.fr/~mottu-jm/welcome-en.html.

**J. Christian Attiogbe** is Professor of Computer Science at University of Nantes (France). He hold a PhD from University of Toulouse, France in Computer Science (1992). In 1999, he joined University of Nantes as an associate professor. His research interests include formal approaches for software modelling and analysis, heterogeneous systems modelling and correct-by-construction of complex systems using refinement techniques. He published several peer-reviewed papers on these topics and co-organised several workshops and conferences on these topics. He has been involved in several research and development projects involving PhD students and industrial partnerships. He led the Reliable Software Group at the laboratory of Digital Sciences of Nantes (LS2N) from 2007 to 2022.You can contact the author at christian.attiogbe@univ-nantes.fr.

**Pascal Berreut** is full Professor at Université Bretagne-Sud (University of South Brittany, France). From 2012 to 2016, he served as Vice President for Social and Economic Affairs at Universite Bretagne-Sud. His research focuses on supervision and automatic control generation for reconfigurable discrete event systems as part of a Human System Cooperation team, with a particular interest in modelling, simulation, piloting and industrial security. Through these activities, he contributes in collaborative and innovative projects for Energy Efficiency, Smart Home Automation, Ambient assisted living, Reconfigurable Manufacturing and industrial Systems in the context of Industry 5.0. He is also involved in transfer platforms and valorization projects. You can contact the author at Pascal.Berruet@univ-ubs.fr.