A Generic Framework for Representing and Analysing Model Concurrency

Steffen Zschaler, Erwan Bousse, Julien Deantoni, Benoit Combemale

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

A Generic Framework for Representing and Analysing Model Concurrency

Steffen Zschaler · Erwan Bousse · Julien Deantoni · Benoit Combemale

the date of receipt and acceptance should be inserted later

Abstract Recent results in language engineering simplify the development of tool-supported executable domain-specific modelling languages (xDSMLs), including editing (*e.g.,* completion and error checking) and execution analysis tools (*e.g.,* debugging, monitoring and live modelling). However, such frameworks are currently limited to sequential execution traces, and cannot handle execution traces resulting from an execution semantics with a concurrency model supporting parallelism or interleaving. This prevents the development of concurrency analysis tools, like debuggers supporting the exploration of model executions resulting from different interleavings. In this paper, we present a generic framework to integrate execution semantics with either implicit or explicit concurrency models, to explore the possible execution traces of conforming models, and to define strategies for helping in the exploration of the possible executions. This framework is complemented with a protocol to interact with the resulting executions and hence to build advanced concurrency analysis tools. The approach has been implemented within the GEMOC Studio. We demonstrate how to integrate two representative concurrent meta-programming approaches (MoCCML/Java and Henshin), which use different paradigms and underlying foundations to define an xDSML's concurrency model. We also demonstrate the ability to define an advanced concurrent omniscient debugger with the proposed protocol. The paper, thus, contributes key abstractions and an associated protocol for integrating concurrent meta-programming approaches in a language workbench, and dynamically exploring the possible executions of a model in the modelling workbench.

Keywords Language Engineering · Model execution · Model concurrency · Simulation · Concurrent Analyses / Debugging

S. Zschaler
King's College London
Department of Informatics
Bush House, 30 Aldwych
London, UK
E-mail: szschaler@acm.org

E. Bousse
University of Nantes
2 Chemin de la Houssinière, BP 92208
Nantes, France
E-mail: erwan.bousse@ls2n.fr

J. Deantoni
University Cote d'Azur
Sophia Antipolis, France
E-mail: julien.deantoni@inria.fr

B. Combemale
University of Rennes
Rennes,France
E-mail: benoit.combemale@irisa.fr

1 Introduction

To realise the vision of model-driven engineering [41] (MDE) and language-oriented programming [51] (LOP), where domain-specific modelling languages (DSMLs) are defined and used for software development, we need to make the development of such DSMLs and the corresponding tool support as easy and cost-effective as possible. Over the last decade, the research community has invested substantial effort into developing so-called language workbenches [20], which provide generic tool support parametrised over language specifications (syntax, semantics...) that can be instantiated by interpretation of, or generation from, a largely declarative language specification. This work has substantially simplified the development of new languages and tool sup-

port, making the MDE and LOP vision more feasible in practice.

While, initially, work on language workbenches focused on supporting the syntax and static semantics of DSMLs (and providing editors and static analysers), leaving execution primarily to the development of template-based code generators, more recently there has been a growing interest in language workbenches for executable DSMLs (xDSMLs, *e.g.,* see chapter 26 of [21] or [8]). Here, in addition to a specification of the language syntax, language engineers provide a specification of the DSML's execution semantics (aka. behavioral semantics) and the language workbench uses this to provide additional services such as (omniscient) debuggers and analysis tools. This has enabled the efficient development of execution and analysis support for new DSMLs.

To date, most language workbenches support the specification of an execution semantics in the form of a sequence of steps (i.e. total order), leading to a sequential execution of the conforming models. However, modern software systems and execution platforms involve complex concurrency concerns. Most modern software systems are distributed and involve complex communications, and current execution platforms are involving complex parallel architectures. When a DSML captures knowledge from a domain where concurrent aspects are important, its operational semantics must capture the concurrent aspects so that they can be handled during the execution of a model. Systematic and generic support for concurrent languages is still missing; although some specialized implementations have been developed (*e.g.,* [32,53]). While the execution semantics can be specified using different paradigms (e.g., imperative or declarative rewriting rules), one of the key challenge is to enable language workbenches to plug-in different meta-programming approaches, with a common execution engine able to interpret a behavioral semantics, possibly with concurrency.

In this paper, we demonstrate how key abstractions can be used to handle the concurrency in a behavioral semantics independently of the way it is actually encoded. Based on these abstractions, we demonstrate how a generic execution engine supporting a given set of common services can be implemented. We show how this generic execution engine is able to embrace two very different ways to specify the concurrency in an operational semantics. We show how such an implementation can be used to provide a protocol for analysis techniques such as concurrent omniscient debugging. We also show how we can give additional control over concurrency to the language engineer and language user to enable the dynamic exploration of a language's concurrency model. We have implemented our approach in the GEMOC Studio language workbench [8], but the overall approach is applicable to any language workbench, possibly using different technological spaces [30].

Specifically, we make the following contributions:

1. A set of *key abstractions* based on studies about multi form logical time to embrace concurrent aspect in a technology independent way;
2. A *generic interface* for both explicit and implicit concurrent models and a *generic execution engine*;
3. The concept of *concurrency strategy* to support the dynamic exploration of the concurrency model for a given conforming and running model;
4. A set of *specific concurrency strategies* that we have found useful for the exploration of concurrency; and
5. A *prototypical implementation* demonstrating the new concepts and the overall approach.

The remainder of this paper is structured as follows: We provide a motivating example in Sect. 2 before introducing our key abstractions in Sect. 3. Section 4 then gives a high-level overview of our approach together with a description of the generic framework for concurrent model execution. Section 5 introduces the concept of concurrency strategies and discusses how they can be used to dynamically explore the concurrency model. We then present the prototypical implementation in Sect. 6 and an evaluation of our approach in Sects. 7 and 8. Finally, we discuss related work in Sect. 9 and conclude the paper in Sect. 10.

2 Background and Motivating Example

The main ingredients of an executable domain-specific modelling language (xDSML) are its *abstract syntax* and its *operational semantics*.[1] In this section, we scope the xDSMLs we are considering in our approach, namely metamodel-based xDSMLs with concurrent operational semantics. At the same time, we introduce an xDSML for production-line systems as a motivating and running example used throughout the paper. Its concurrent operational semantics is defined twice with very different state-of-the-art meta programming approaches.

2.1 Abstract Syntax

We assume that the abstract syntax of an xDSML is defined using a *metamodel*, which is an object-oriented model composed of interconnected metaclasses, each

[1] We deliberately leave out the concrete syntax since it does not impact the executability of a DSML, and our approach is independent of any, textual or graphical, concrete syntax.

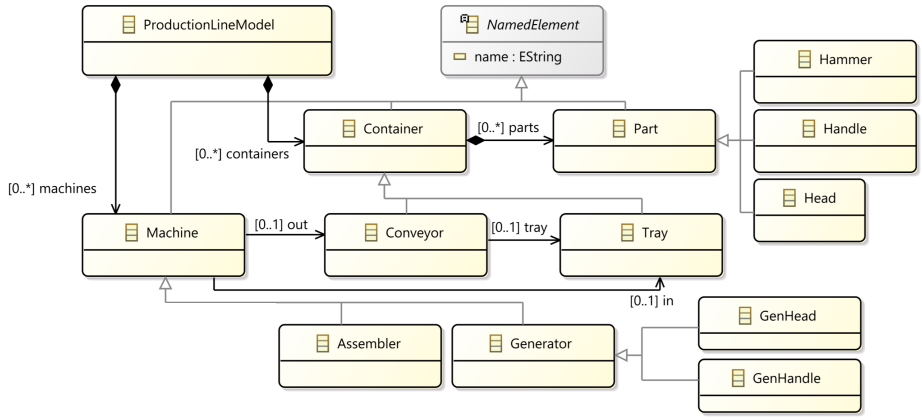

Figure 1: Metamodel of the production-line language

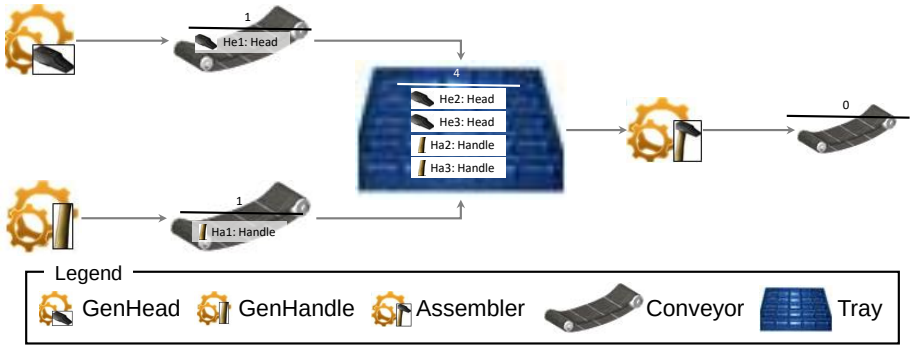

Figure 2: An example production line

capturing a concept of the domain of interest. To illustrate our proposal, we introduce an xDSML[2] that allows the modelling of simple production-line systems (PLS). As can be seen in the metamodel in Figure 1, such production-line systems consist of `Machine`s manipulating `Part`s and connected to `Container`s (which can be `Tray`s holding `Part`s ready to be manipulated by a machine or `Conveyor`s taking `Part`s from a `Machine` to a `Tray`).

Our production-line xDSML has been specialised to a very narrow domain, namely for describing production lines that produce `Hammer`s from `Head`s and `Handle`s. Consequently, appropriate subclasses are defined for the `Part` metaclass and suitable specific types of `Machine`s have also been defined in the metamodel. The xDSML, then, allows combining these elements into suitable production-line models, using a concrete syntax defined in Sirius [48]. Figure 2 shows an example model of a simple production line. On the left, there are two machines producing handles and heads, respectively, and depositing them onto conveyors that eventually will move them into a shared tray. An assembler machine then takes handles and heads from this tray and will produce hammers in turn. The example model shows a state of the system, where 3 heads and 3 handles have been produced and 2 of each are awaiting assembly in the shared tray.

[2] This xDSML has previously been developed for the *e-Motions* system [39].

2.2 Defining a Concurrent Operational Semantics

Once the abstract syntax of the language is specified, it is important to define the behavioural semantics of the language to enable execution and analysis support for new DSMLs. Existing language workbenches often overlook the concurrency aspect of the DSML behavioural semantics, leading to poor support of concurrency analysis. For now, we consider a concurrent operational semantics to be an operational semantics that allows exploration of concurrency related concerns; typically allowing to explore different execution paths due to interleavings. While there are many ways to define such concurrent operational semantics, we use, for illustra-

tion purposes, in this paper two different approaches applied to the production-line xDSML: one using declarative rewriting rules defined using a graph transformation approach (specifically, Henshin [43]), and one using imperative rewriting rules together with a modular and formal description of how and when the rewriting rules can be applied (specifically, MoCCML [17,16]).

To define the concurrent operational semantics of our xDSML, we first need to differentiate the runtime state of a model from the static parts of the model. In any executable model, some parts are static (aka. abstract syntax tree), and some others correspond to the *runtime state*, also called the dynamic state. For instance, if one considers the concept of Variable in a language, its type and its initial value are static parts of the model while the *current value* of the variable is part of the runtime state. While both the model and the runtime state will be accessed by the semantics, only the runtime state can be changed during execution. The GEMOC studio [10] supports a modular definition of both parts (i.e., separate cross-referenced metamodels that are subsequently woven together). However, for the sake of simplicity, we present a combined version of the metamodel: the runtime state is captured by meta-class `Part` and its sub-classes and their associations with other meta-classes. Everything else captures the static part of any production-line model (*i.e.,* the structure of the production line itself rather than what parts are currently being produced).

Next, on the basis of the metamodel (both the static part and the runtime state), we need to define the actual concurrent operational semantics. We implemented a first version using our Henshin engine for the GEMOC Studio [53]. Henshin provides a graph transformation tool [43]. The rewriting rules can then be implemented in a declarative way (cf. Figure 3, focusing on the rules relevant for our example[3]). Hence, we provide a structured operational semantics using graph-transformations to describe the individual steps—called graphical operational semantics by Corradini [14]. Rules `generateHandle` and `generateHead`, respectively, specify that a `GenHandle` and `GenHead` machine can produce a new `Handle` or `Head` at any time. Rule `moveAlong` describes that any `Part` on a `Conveyor` can move to the corresponding `Tray`, if any. Finally, rule `assemble` shows how an `Assembler` machine takes a `Head` and a `Handle` and produces a new `Hammer` from them. The Henshin engine comes with a rule application system that computes the applicable rules for a given runtime state, and provides options to apply one or several of them simultaneously as decided by the user, this way allowing exploration of different acceptable execution paths [53].

[3] While not shown in these example rules, Henshin also supports rules that read or modify attribute values of model elements.

Listing 1: A simple Kermeta aspect for the generate head rule

```
@Aspect(className=GenHead)
class GenHeadAspect{
  def void work(){
      var aHead = PLSFactory.eINSTANCE.createHead()
      _self.out.currentParts.add(aHead)
  }
}
```

Listing 2: A MoCCML excerpt to constrain the call (partial) order of rewriting rules

```
context Machine
  def : doWork : Event = self.work()
context Conveyor
  def : doMoveAlong : Event = self.moveAlong()
context Conveyor
  inv moveAfterMachineProductionNoInitial:
  (self.parts->size() = 0) implies
  Relation Precedes(
   self.Machine->first().doWork, self.doMoveAlong)
```

We also implemented a second version of the same concurrent operational semantics for the motivating example using MoCCML [17,16], a dedicated metalanguage to formally define partial orders, and Kermeta 3 [24] to define the rewriting rules as object-oriented and imperative methods. For instance, the *generateHead* in Kermeta 3 is shown in Listing 1.

Nothing in Listing 1 specifies when the rewriting rule should be called. This is the goal of the MoCCML model [10]. For this purpose, each rule is associated to an event and the events are constrained together based on the static information in the model. For instance, in Listing 2, two events are defined (*doWork* and *moveAlong*). Then, a relation specifies that, if there are no initial parts on the conveyor, then the machine feeding the conveyor must work before the conveyor can move the item along. This way, if we consider the top left conveyor of Figure 2, the head generator can work at any time but the conveyor can move along only the number of times the generator worked.

There is a fundamental difference between the two ways we have specified the concurrent operational semantics—specifically, how the semantics determine when a particular event can occur. In graphical operational semantics, this is implicit in the rules defining the different kinds of events: if a rule's left-hand side matches the current state, the event can potentially occur, possibly multiple times if there are multiple matches and possibly concurrently with some other rules if its right hand side does not overlap with the left hand side of

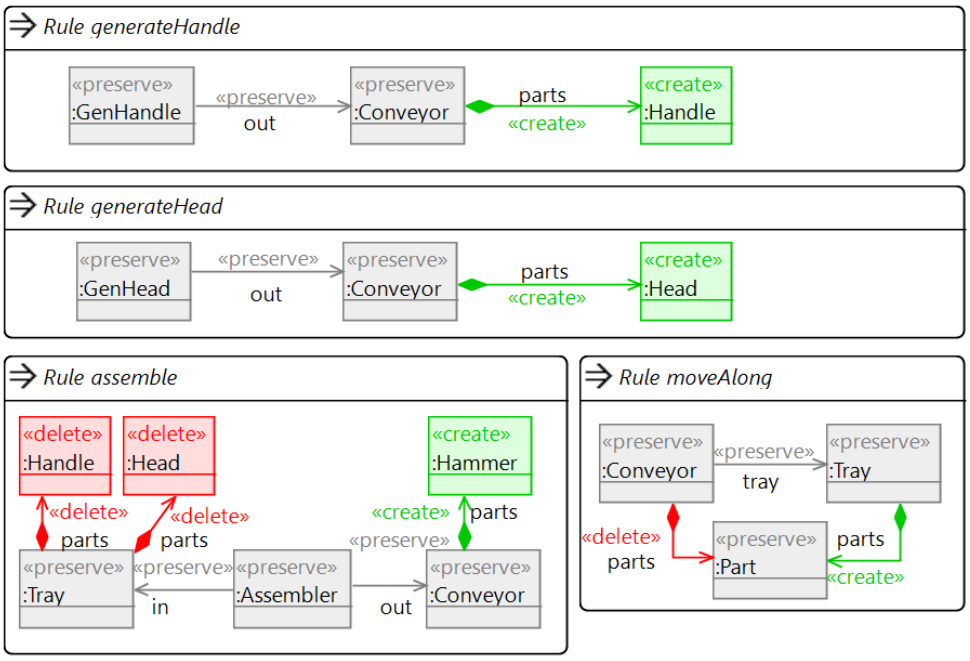

Figure 3: Operational semantics of the production-line xDSML specified using Henshin. In each rule, grey elements marked *preserve* represent model elements that need to be present for the rule to be applicable and that won't be changed by the rule. Red *delete* elements represent model elements that must be present and will be removed and green *create* elements mark elements that will be newly created in the model when the rule is executed.

the other one(s). In contrast, with MoCCML, the conditions under which an event can occur and the relationships between event occurrences are specified explicitly.

The two different approaches, namely Henshin and MoCCML, come with different underlying paradigms, leading to different implementations of the same concurrent operational semantics. They both provide interesting features, with different pros and cons. While a full comparison of these two approaches is out of scope for this paper, the differences motivate the need for key abstractions that allow different approaches to be handled uniformly. In this paper, we introduce a generic framework in which concurrency exploration can be done independently of the approaches used for the description of the concurrent operational semantics.

3 Key Abstractions to Embrace Concurrency

In order to define a generic framework with concurrency specific services independently of any specific technology used for specifying the operational semantics, we need to rely on key abstractions to represent the concurrent part of the operational semantics. In this section, we introduce the notions of *concurrency model*, *logical steps* and their relation in our definition of a *concurrent operational semantics*.

3.1 Concurrency Model

In order to introduce the concepts of concurrency model and concurrent execution trace, we rely on an introductory example. Let us consider the simple language $\mathcal{S}$ composed of `Statements` where a statement can be an `Action`, a `Fork` with its set of `Blocks` of statements that can be executed concurrently, or a `Join`.

```
A;
fork f1:
    | -> { B; C; }
    | -> { D; E; }
join f1;
F;
```

Figure 4: An illustrating program written in the $\mathcal{S}$ language.

Figure 4 provides a program written in the $\mathcal{S}$ language. In this example, the sequence $B;C$ can be executed concurrently with the sequence $D;E$; both after

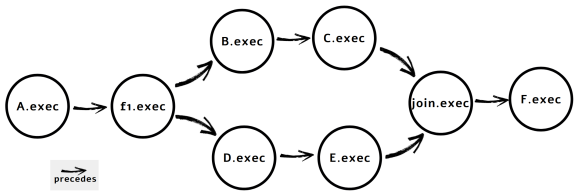

Figure 5: A representation of partial order underlying the simple $\mathcal{S}$ program from Figure 4, as it could be computed by the concurrency model.

A and before F. Additionally, the C must always follow B and E must always follow D. This is a partial order that can be represented like in Figure 5 by a set of *Precedes* constraints between the application of rewriting rules. In this partial order, any total order is a correct execution with respect to the concurrent operational semantics. This total order is more than just a topological sort of the graph; it should consider the concurrent application of rewriting rules. For instance, in program $\mathcal{S}$ the application of $B.exec$ and $D.exec$ can occur concurrently. Additionally, in more realistic examples, a partial order is usually not expressive enough since conflicts between two sets of actions may be required (*e.g.,* due to an $if - then - else$ statement or due to access to a shared resource). The appropriate expressiveness to specify the set of acceptable executions in a concurrent execution context is out of the scope of this paper and has been theoretically studied for a long time [52,34,3,2,33,15]. We took advantage of Multiform Logical Time [2] to embrace these formalisms and defined a *concurrency model* as an artifact which, given a specific program at a given runtime state (*i.e.,* at a given step of its execution), can provide the exclusive sets of rewriting rules that can be applied to move to the next step. Each of these sets specifies the rewriting rules that can be applied concurrently. From the runtime state point of view, the application of a set of rewriting rules is seen as a unique operation. These sets are the eligible futures of the execution. The application of their rewriting rules leads to different execution branches and they can consequently be used to explore, to understand or to analyse the intrinsic concurrency of the model and its implication (*e.g.,* deadlock or functional non-determinism). The sets of rewriting rules proposed by a concurrency model at a given step take both the causalities and the conflicts into account. Note that conflicts can result in different execution branches that will never merge again due to their effect on the runtime state or on the opposite it can result in different execution branches representing different interleavings of rewriting rules that result in a same runtime state where the branches merge. It is worth noting that the concurrency model used in this example assumes that all actions are *atomic*; that is they do not take any time. Durative actions—actions that take time and where an action B may start partway through the execution of an action A—are an important concept in concurrency modelling. Such durative actions can be captured on top of our concurrency model—for example, by providing an explicit model of actions under execution (and possibly of time) and translating durative actions into an explicit atomic start and end action (cf. [39,40]), but the details of any such encoding are out of scope for our paper.

3.2 Logical Steps

While executing a concurrent program, there is a need to make explicit what rewriting rules have actually been applied between two runtime states. For this purpose, we introduced the notion of *Logical Step*. A logical step is abstract and defined as a set of changes realized in the runtime state. This means that here also we consider a general form of concurrency similar to, for instance, tagged signal [34] or logical time [15], where the partial ordering of event occurrences is the primary concern, independently of the actual process behind each event occurrence. A logical step can be either an *atomic step* or a *parallel step*. An *atomic step* is a specific logical step linked to one rewriting rule of the transition system, whose execution realizes a set of changes in the runtime state—for example, the execution of an action in an $\mathcal{S}$ program. To execute an atomic step, we need to access and read different parts of the model and of the runtime state and, then, change specific parts of the runtime state. We call *atomic step footprint* [23] the set of elements of both the model and the runtime state that are read or changed as well as the set of metaclasses of the runtime state of which new instances are created during the execution of an atomic step. A *parallel step* is composed of a set of *atomic steps* that are executed concurrently. The *parallel step footprint* is the union of internal atomic step footprints.

To illustrate, each topological sort of the directed graph from Figure 5 is a valid trace of the illustrative program from Figure 4; where each edge is a logical step and each node a runtime state. Figure 6 can be seen as a Labelled Transition System [27], where labels are structured by using logical steps. On the left of Figure 6, we can see that any execution starts with an atomic step A. It means that there is a single rewriting rule called in the first step of the execution, which corresponds to the execution of an action in the program. Then the execution continues with another

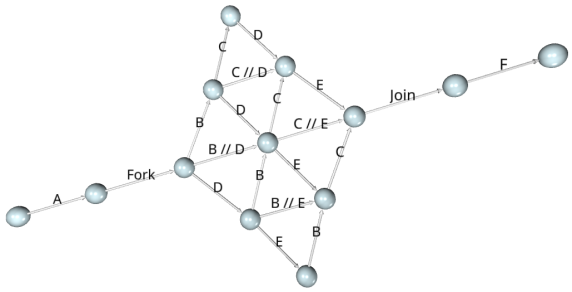

Figure 6: All possible interleaving of actions from the $\mathcal{S}$ program from Figure 4

single action executed: *Fork*. At this point, three different logical steps are eligible futures of the execution: one atomic step where only action B is executed, one where only action D is executed and one parallel step where both actions B and D are executed in parallel. If, for instance, the execution follows the step where only B is executed, then three new logical steps are eligible futures: C, $C \parallel D$, or D alone. Let us consider that $\bullet$ is a runtime state; $\text{-}|.|\text{-}$ represents a logical step and that the name of an action represents the execution of the underlying rewriting rule. One possible execution trace is: $\bullet\text{-}|A|\text{-}\bullet\text{-}|Fork|\text{-}\bullet\text{-}|B|\text{-}\bullet\text{-}\left|\begin{matrix}C\\D\end{matrix}\right|\text{-}\bullet\text{-}|E|\text{-}\bullet\text{-}|Join|\text{-}\bullet\text{-}|F|\text{-}\bullet$. The *diamond* from Figure 6 actually represents all the acceptable interleavings between the execution of the $B;C$ sequence and the $D;E$ sequence that has been constructed by querying the concurrency model and by visiting all logical steps.[4]

To summarize, the concurrency model is an artefact that can be used to figure out what is the next acceptable set of exclusive logical steps that can be taken at any time during the execution. It acts as a scheduler of the rewriting rules that relies on a foundational logical time model. The logical steps are then a way to 1) store what are the rewriting rules that have been called between two runtime states; and 2) make explicit the footprint of the executed rewriting rule(s).

3.3 Concurrent Operational Semantics

We call a concurrent operational semantics an operational semantics that provides a specification of the concurrency semantics (of the constructs defined within the syntax), such as we can reason about it (e.g., exploring impact of different interleavings). Indeed, a common approach in the literature is to rely on the meta-language provided to specify the operational semantics to implicitly describe this concurrency semantics, often intertwined with the operational semantics. This means that the transition system corresponding to the operational semantics is mixed up with the concurrency model that would describe possible interleavings or parallelism. For instance, one would use Java for defining the execution semantics of a given DSML (e.g., in the form of a visitor), and to rely on the thread Java library to specify the concurrency semantics of the DSML constructs that require it. Hence, the DSML semantics is not only described in the DSML semantics' specification, but also implicitly inherited from the one from Java thread (thus, from the concurrency model of the JVM). This makes the concurrency model depend on the concurrency model provided by the meta-languages to define DSML execution semantics. As a result, it is difficult to reason over the concurrency concern since this is mixed up within the operational semantics.

In this paper, we investigate how concurrency models derived from concurrent operational semantics expressed using different mechanisms can be captured by a common generic protocol. Once we have this generic protocol, we can use it to define new concurrency-specific services. In this paper, we explore an example of such a service by introducing the new concept of concurrency strategies, which can be used to dynamically explore concurrent execution traces. As we will discuss in Sect. 7, this enables new opportunities for language engineers around the flexible specification of concurrent semantics and for language users around more efficient interactive exploration.

4 A Generic Framework for Concurrent Model Execution

4.1 Approach Overview

Figure 7 presents an overview of the proposed framework for representing and analyzing model concurrency. It is illustrated on the basis of our implementation within the GEMOC Studio (with dashed lines in Figure 7), but can be broadly adopted in any language workbench supporting the specification of the DSL execution semantics with possible concurrency.

The GEMOC Studio subsumes the *Eclipse Modeling Framework* [42] and its ecosystem which provides the *Model Editing Server* either for textual editing (thanks to Xtext[5] which supports the *Language Server Protocol (LSP)*) or graphical editing (thanks to Sirius[6] which supports the *Graphical Language Server Protocol (GLSP)*).

[4] More details about this example can be retrieved from `http://github.com/jdeantoni/simpleConcurrentLanguage`.

[5] cf. `https://www.eclipse.org/Xtext`

[6] cf. `https://www.eclipse.org/sirius`

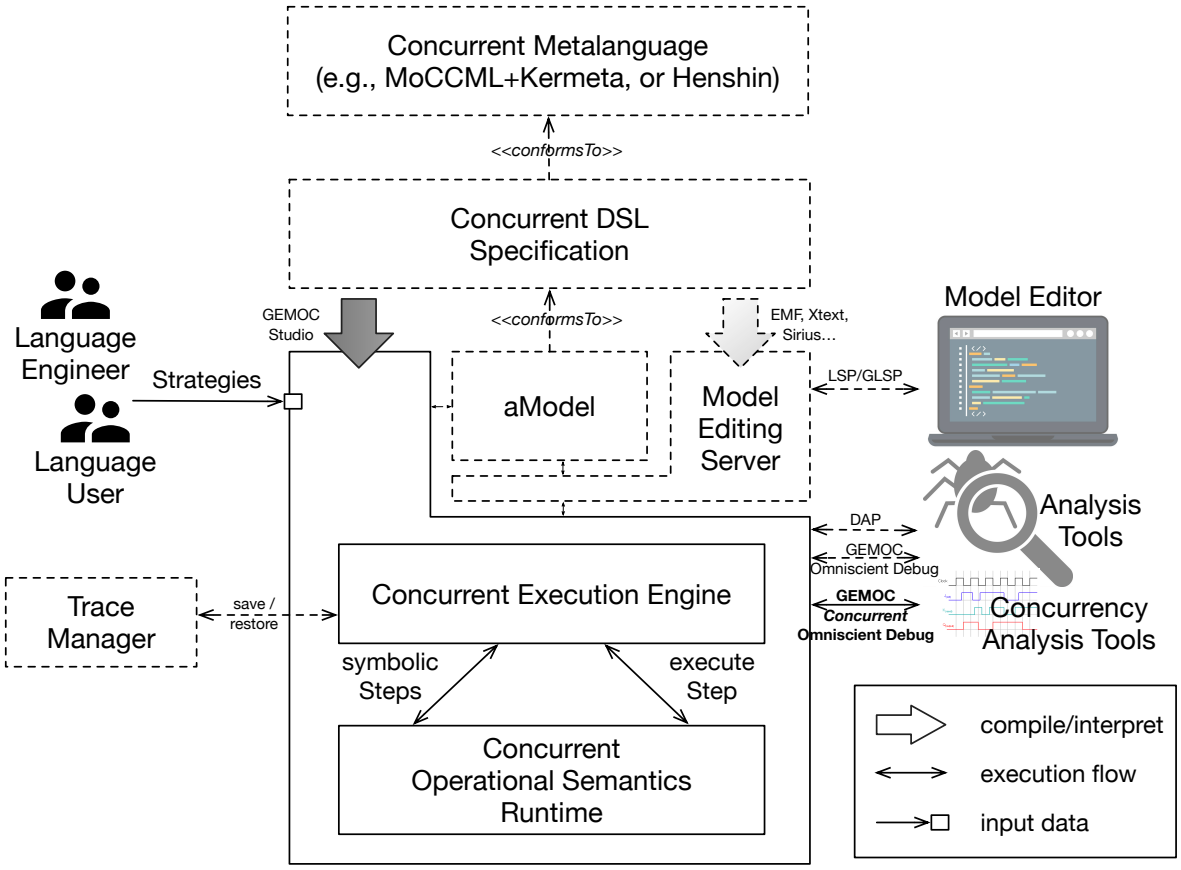

Figure 7: Approach overview (existing elements in dashed lines and contribution in solid lines)

Within the GEMOC Studio, two concurrent metalanguages are already included to specify DSL execution semantics with possible concurrency, namely MoCCML combined with Kermeta [10] and Henshin [43]. While each approach provides unique constructs leading to some differences in the expressivity as discussed in the previous section, the proposed framework offers a unified way to interact with the resulting execution engine and drive the possible executions of a conforming model.

The proposed framework is customised according to a given *Concurrent DSL Specification* expressed with one of the *Concurrent Metalanguages*, and is responsible for the execution of a given conforming model. It includes a generic *Concurrent Execution Engine*, interacting with i) a *Concurrent Operational Semantics Runtime*, specific to a meta-programming approach and the associated meta-language initially used, which is in charge of interpreting a given concurrent operational semantics from the concurrent DSL specification; and ii) a *Trace Manager* in charge of managing the concurrent execution trace of a given model run.

The framework offers two ways to interact with the execution:

1. The language engineer or the language user can provide *strategies* to drive the resolution of the concurrency of a given model.
2. The framework provides an interface that can be used according to the *GEMOC Concurrent Omniscient Debugging* protocol, such that language-agnostic *Concurrency Analysis Tools* can be developed and used. This protocol subsumes the *Debug Adapter Protocol* (DAP)[7], and covers all the provided facilities for analyzing the concurrency, including the definition, usage and configuration of strategies.

4.2 Framework Description

We now review the main ingredients (cf. Figure 7) provided by the generic framework for concurrent model execution.

[7] cf. `https://microsoft.github.io/debug-adapter-protocol`

Model and runtime state As explained in Sect. 2, we consider that the executed model conforms to an xDSML and that there is a separation between model and its runtime state. In the proposed execution flow, we show in a simplified fashion two high-level services called *read* for reading the model and runtime state, and *write* for changing the runtime state.

Representation of steps In the proposed framework, an atomic or parallel step is an explicit object that embodies a possible execution step, yet to come, in the execution trace. In particular, step objects are created by the concurrent operational semantics runtime to publically announce what are the next possible steps (see below).

Moreover, let us remind that at every point of the execution, there can be *multiple* valid combinations of atomic steps, and thus multiple possible parallel steps. To better represent such a set of possible parallel steps, the framework provides a *symbolic* representation that comprises two pieces of information: (1) A set of all the atomic steps that can occur at this point of the execution; and (2) A set of propositional constraints specifying which combinations of these atomic steps can legally occur concurrently. To simplify the writing, we use the term *symbolic steps* for parallel steps represented symbolically with this pair of elements. Note that this representation is independent of how the concurrency model of the semantics is specified and evaluated.

Concurrent operational semantics runtime As explained in Sect. 2, we assume that the considered xDSML has a specification of a concurrent operational semantics as part of its concurent DSL specification. We call *concurrent operational semantics runtime* the executable software artefact obtained from this specification (*e.g.,* using a compiler, or a generic runtime parameterized by the specification), along with all third-party software required to execute these artefact (*e.g.,* interpreters or solvers). We remind that executing a model using a concurrent operational semantics runtime results in a sequence of *parallel steps*, each composed of a valid combination of *atomic steps* that can be executed concurrently.

To be able to drive an execution using a concurrent operational semantics runtime, we consider that it must provide at least two services: *computeSymbolicSteps*, which returns the set of eligible parallel steps at the current runtime state in the form of a pair $\langle$*atomic steps, constraints*$\rangle$ to avoid enumerating all the eligible steps; and *executeAtomicStep*, which executes one of the atomic steps contained in a parallel step, which will result in changes in the runtime state.

Strategies In Sect. 5, we will show how the concurrency model can be dynamically explored using a set of concurrency strategies. Different types of concurrency strategies will be introduced in Sect. 5. For the purposes of describing the generic framework, it is sufficient to understand that concurrency strategies is a non intrusive way to reduce the interleavings presented to the user compared to the one proposed by the concurrency model, itself produced by the underlying concurrent operational semantics runtime.

Concurrent Execution Engine At the core of our proposal is the *engine*, which brings together all the parts when conducting the execution of the model. It is the only part that an external client—for example, a language user or engineer through a modelling environment—must use to manage the execution of a model. The engine provides three main services: **(a)** *start* takes a model and a concurrent operational semantics runtime, and triggers the initialization and the beginning of the main execution loop; **(b)** *computePossibleParallelSteps* uses both the concurrent operational semantics runtime and a set of enabled strategies to compute the next set of possible parallel steps for the executed model—this service is further detailed in Sect. 5.2, after the presentation of the different types of strategies; **(c)** *executeParallelStep* tells the engine to execute a given parallel step, chosen from the set of possible steps.

Note that, among other possibilities, these services aim to serve as a key piece for *concurrent omniscient debugging*. An omniscient debugger is a type of tool enabling the interactive execution of a model in a similar fashion to a traditional interactive debugger, but with the extra possibility to jump back and revisit previously reached execution states. When used for a concurrent execution, a *concurrent* omniscient debugger must also provide the possibility to choose which execution steps to execute among the possible next ones, *e.g.,* when revisiting prior states. Our proposed interface and protocol provides exactly this missing piece, thus enabling concurrent omniscient debugging.

4.3 Generic Concurrent Language Execution Flow

Figure 8 is a sequence diagram showing the execution flow resulting from the execution of a model using the protocol of the proposed framework. From the left, the first lifeline represents the *client* that wishes to execute a model. This client is typically a language user or engineer that uses the engine interactively through a modelling environment, or a dynamic analysis tool that automatically explores the state space. The first task is to configure the engine and start the execution:

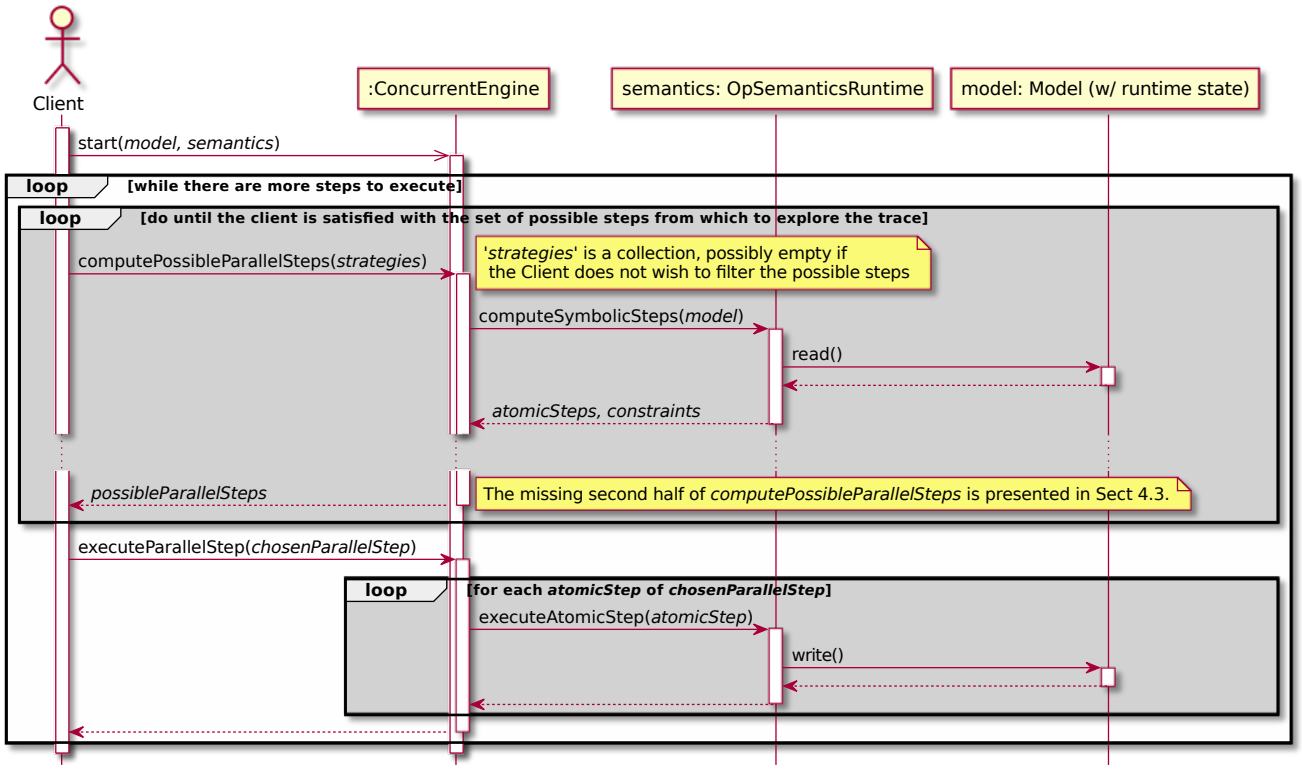

Figure 8: Sequence diagram of the generic concurrent language execution flow.

the client must provide both a model and a concurrent operational semantics runtime in order to launch the execution (*start*). Then the engine runs a loop while there are more steps to execute. In each iteration, the client asks the engine to compute the next set of possible parallel steps (*computePossibleParallelSteps*). The client can decide to provide this service with a set of concurrency strategies in order to filter the possible parallel steps, and can trigger the service as many times as required to try different combinations of strategies, until it is satisfied with the set of possible steps from which to explore the trace. To fulfill this request, the engine asks the concurrent operational semantics runtime to compute the set of symbolic steps that can legally occur given the model and the current runtime state (*computeSymbolicSteps*). This computation requires accessing the model and the runtime state (*read*). Note that, as this part greatly depends on how the different types of strategies operate, we postpone its complete description to Sect. 5.2 where strategies are explained in detail.

Once the set of possible parallel steps has been determined, the client makes a decision and asks the engine to execute one parallel step among the possible steps (*executeParallelStep*) Finally, the concurrent execution engine asks the operational semantics runtime to execute each of the atomic steps that comprise the chosen parallel step (*executeAtomicStep*), which changes the runtime state in turn (*write*). Note that because the atomic steps can occur concurrently, the order in which their corresponding changes are applied to the runtime state does not matter.

5 Dynamic Exploration of Execution Traces

This section presents how the proposed framework can be used for the dynamic exploration of the concurrency model for a given model.

We first present concurrency strategies, a novel concept to define what are the interleavings that should be presented to the user in order to ease the support of such dynamic exploration. It is important to notice that this exploration should be done without altering the concurrent operational semantics of the language. Rather, it is a mechanism that can be enabled and disabled on the fly to ease the navigation into specific execution paths of interest. After we presented concurrency strategies, we detail how they can be applied through a generic interface, independently of how the concurrent operational semantics is expressed.

5.1 Concurrency strategies

As explained in the previous section, our proposed framework relies on a symbolic representation of the set of possible parallel steps. Such a representation can effectively abstract specific implementations of concurrent operational semantics in terms of propositional constraints over sets of atomic steps. This opens the opportunity to provide additional support for language engineers and language users in order to dynamically filter and explore the language concurrency when executing a specific conforming model. In our framework, this support is provided through the concept of *concurrency strategies* used to restrict the potential concurrency. This allows the human or tool that conducts the execution to focus on a set of possible futures that are of interest at a given time. Multiple strategies can be applied together to further reduce the size of the subset of steps. We differentiate two types of strategies for filtering the concurrency model dynamically:

1. *Symbolic concurrency strategies* add additional constraints to the propositional formula provided by a concurrent operational semantics runtime. One class of symbolic concurrency strategies adds mutual exclusion constraints between pairs of atomic steps, specifying that these steps cannot occur concurrently. More general symbolic concurrency strategies add more general constraints—for example they might constrain the number of atomic steps that can occur concurrently. Note that these strategies already have access to the set of atomic steps, so they might generate constraints based on properties of these steps, including by inspecting runtime state that the steps access or modify.
2. *Operational concurrency strategies* are applied after a concrete set of parallel steps has been computed and algorithmically filter this set of steps. This is inherently less efficient than a symbolic concurrency strategy because it requires a constraint solver to enumerate all potential parallel steps only for some of them to be later filtered out. However, it allows operational concurrency strategies to compare different parallel steps and make decisions based on the comparison result.

Because these strategies can be defined on top of our symbolic step representation, they can be used for any language, independently of the formalism used for specifying the operational semantics. MoCCML internally encodes the concurrency model as a propositional formula, which is what symbolic concurrency strategies manipulate. As a result, some symbolic concurrency strategies could also be statically encoded in a MoCCML semantics.[8] However, by making them available at the level of the generic concurrency engine, these strategies can also be applied for languages with other semantics specifications (*e.g.,* using Henshin). Operational concurrency strategies cannot be expressed natively in either semantic formalism.

Moreover, all strategies can be selected and deselected dynamically during model execution, allowing for the dynamic exploration of execution traces. This cannot be achieved if the strategies are statically defined within the semantics. Thus, we could envision a work flow where a language engineer or language user might want to explore the effect of different choices in the concurrency model on the possible set of execution traces before properly encoding the final choice in the semantics (for a language engineer) or the model (for a language user). Furthermore, the selection and configuration of strategies could also be made accessible to analysis tools via a suitable extension of the proposed protocol (see next section for a brief discussion of the protocol implemented in the GEMOC Studio), enabling these tools to perform a more focused analysis of the overall state space.

Consider the example runtime state shown in Figure 2. In this state, the operational semantics from Figure 3 will generate the following atomic steps:

- `GHa: GenHandle`,
- `GHe: GenHead`,
- `Me: MoveAlong(He1)`,
- `Ma: MoveAlong(Ha1)`,
- `A22: Assemble(He2, Ha2)`,
- `A23: Assemble(He2, Ha3)`,
- `A32: Assemble(He3, Ha2)`,
- `A33: Assemble(He3, Ha3)`.

Not all of these steps can occur concurrently. In particular, there are only two valid combinations of Assemble steps: A22 can be combined with A33, and A23 can be combined with A32. Other combinations would require one `Part` to be used twice. This is captured by the propositional formula generated by the concurrent operational semantics runtime:

$$\begin{aligned}(GHa \vee GHe \vee Me \vee Ma \vee A22 \vee A23 \vee A32 \vee A33)\\ \wedge (A22 \implies \neg (A23 \vee A32))\\ \wedge (A23 \implies \neg (A22 \vee A33))\\ \wedge (A32 \implies \neg (A22 \vee A33))\\ \wedge (A33 \implies \neg (A23 \vee A32))\end{aligned}$$

[8] Although, for example the overlap strategy we describe below cannot statically compute the additional constraints.

From this, a range of parallel steps can be constructed, including, for example:

$(GHa, GHe, Me, Ma, A22, A33)$

or

$(GHa, GHe, Me, Ma, A23, A32)$.

Next, we describe some examples of strategies that have been implemented within the framework provided in the GEMOC Studio (see next section); note that videos that illustrates the use of strategies and more complex examples are referenced from the companion webpage (see footnote #14):

– *Symbolic Concurrency Strategies:*
  – *Set of Events.* This strategy allows only atomic steps realising one of a particular set of events to be executed concurrently. An "event" is a category of atomic steps: all steps that correspond to the same rewriting rule (*e.g.,* a graph transformation rule in the Henshin case or a Kermeta operation in the MoCCML case) are said to realise the event named after that rewriting rule (in MoCCML, events are explicitly declared in the specification of the concurrency model, *cf.* Listing 2). For example, we could specify that only `GenHead` and `GenHandle` steps can be executed concurrently (and that all others can only be executed individually). In our example, this adds the following constraint to the propositional logic formula:

$$\begin{aligned}
Me &\implies \neg(GHa \vee GHe \vee Ma \vee A22 \vee A23 \vee A32 \\
&\quad \vee A33) \wedge \\
Ma &\implies \neg(GHa \vee GHe \vee Me \vee A22 \vee A23 \vee A32 \\
&\quad \vee A33) \wedge \\
A22 &\implies \neg(GHa \vee GHe \vee Ma \vee Me \vee A23 \vee A32 \\
&\quad \vee A33) \wedge \\
A23 &\implies \neg(GHa \vee GHe \vee Ma \vee Me \vee A22 \vee A32 \\
&\quad \vee A33) \wedge \\
A32 &\implies \neg(GHa \vee GHe \vee Ma \vee Me \vee A22 \vee A23 \\
&\quad \vee A33) \wedge \\
A33 &\implies \neg(GHa \vee GHe \vee Ma \vee Me \vee A22 \vee A23 \\
&\quad \vee A32)
\end{aligned}$$

    As a result, only the atomic steps and the parallel step (GHa, GHe) are kept.
  – *Overlap.* This strategy allows concurrency only where two atomic steps fully overlap in their static footprint (that is, the part of the model they query but do not change). In the PLS example, this is useful to focus on concurrency at individual machines. For the example runtime state, this would add the following constraint.

$$\begin{aligned}
GHa &\implies \neg(GHe \vee Me \vee Ma \vee A22 \\
&\quad \vee A23 \vee A32 \vee A33) \wedge \\
GHe &\implies \neg(GHa \vee Me \vee Ma \vee A22 \\
&\quad \vee A23 \vee A32 \vee A33) \wedge \\
Me &\implies \neg(GHa \vee GHe \vee Ma \vee A22 \\
&\quad \vee A23 \vee A32 \vee A33) \wedge \\
Ma &\implies \neg(GHa \vee GHe \vee Me \vee A22 \\
&\quad \vee A23 \vee A32 \vee A33) \wedge \\
(A22 \vee A23 \vee A32 \vee A33) &\implies \neg(GHa \vee GHe \\
&\quad \vee Me \vee Ma)
\end{aligned}$$

    Note that this constraint can only be computed once the specific atomic steps are known and their footprint can be established. To calculate the above constraint, the overlap strategy inspects the footprint of the set of all potentially concurrent atomic steps and identifies those with a shared footprint. In the example, the four `Assemble` steps `A22`, `A23`, `A32`, `A33` are the only ones with a shared footprint. Specifically, looking back to the `Assemble` rule shown in Figure 3 the footprint of these steps is given by the instances of `Tray`, `Assembler`, `Conveyor`, and the links between them (`in` and `out`). It is easy to see that the four steps share these instances as they refer to the same `Assembler` machine, which is only connected to one `Tray` and one `Conveyor`. This is reflected in the implications generated in the constraint above: the first four implications state that `GHa`, `GHe`, `Me`, and `Ma` cannot be executed concurrently with any of the other actions in the current runtime state, while the final implication states that the `Assemble` actions cannot be concurrent with any of the other non-`Assemble` actions.[9]
    The constraint is not static, but depends on the specific runtime state of a given model. In MoCCML, such constraints cannot be captured at all, because the concurrency specification does not have access to the footprint of individual events. In a Henshin-based semantics, it is possible to *disallow* concurrent execution of actions with a shared footprint—for example, by adding an access counter that is modified by each rule, creating a write–write conflict between rules. However, specifying that only rules with a shared footprint can be executed cannot be done in a generic way with simple Henshin rules.

[9] This final implication is redundant in this example.

Variants of this strategy would *disallow* concurrency where there was overlap, or might trigger already for partial overlap.
  - *Concurrency Limit.* This strategy limits the maximal concurrency. For example, we could specify that at most three atomic steps should be executed concurrently at any given time (*e.g.,* because we have limited processing capability). The strategy adds a constraint to ensure that at most three atomic steps are selected to form a possible parallel step.
  - *Force Presence/Absence.* It may be important for a user to focus on specific (set of) rewriting rule(s) (e.g., the generation of head in the PLS language). In such a case, it may be helpful to reduce the set of steps to the ones where the rules to investigate are actually called. Similarly, we defined a strategy to focus on the *absence* of a specific set of rules. Note these strategies are different from the 'Set of Events' strategy: that strategy restricts what can happen concurrently, while the strategies here completely remove steps that do not refer to a particular event.
- *Operational Concurrency Strategies:*
  - *Token Elements.* Sometimes, we may wish to consider different parallel steps conceptually equal if their footprint only differs in model elements of a particular type. For example, for the production line system it doesn't actually matter which pair of `Handle` and `Head` are selected to assemble a `Hammer`. The token-elements strategy allows to specify the type of elements which should be considered not to carry identity (in essence, these elements are treated as tokens only[10]). For example, we could specify that any element that is an instance of `Part` should be treated as a token. As a result, steps that only differ in token elements will be treated as equal and only one of these steps will be kept in the set of possible parallel steps. In our example, only one of

    $(GHa, GHe, Me, Ma, A22, A33)$

    and

    $(GHa, GHe, Me, Ma, A23, A32)$

    would be available to be picked.
  - *Maximal Concurrency.* As the number of atomic steps grows, the set of possible parallel steps can become too large to comprehend for a human user. In such a case, it may be helpful to reduce the set to only the maximally concurrent steps. A step s is maximally concurrent in a set $\mathcal{S}$ of steps iff $\nexists s_2 \in \mathcal{S}.\ s \neq s_2 \wedge s.substeps \subset s_2.substeps$.

[10] The term 'token' is inspired by the notion of tokens in Petri nets.

5.2 Concurrency strategies in the execution flow of the protocol

In Sect. 4, we presented the execution flow of the protocol of the proposed framework. A key service required for this flow is *computePossibleParallelSteps*, whose purpose is to use both the concurrent operational semantics runtime and the strategies to compute the set of possible parallel steps offered to the client. In this part, we explain in detail how this service operates within the protocol, in particular regarding the use of concurrency strategies.

To integrate both symbolic steps and concurrency strategies in the execution flow, the concurrent execution engine must provide an additional service called *enumerateAllPossibleParallelSteps*, which takes a set of symbolic steps (*i.e.,* a set of atomic steps and a set of constraints specifying the combinations of these atomic steps that are allowed or required to occur concurrently) and enumerates the set of possible combinations of atomic steps satisfying the given constraints, which yields the set of possible parallel steps—for this task, the engine can rely on an existing external CSP solver such as Choco [25].

Figure 9 is a complete sequence diagram of the *computePossibleParallelSteps* service. The first part is identical to what was presented in Sect. 4.3: the concurrent operational semantics runtime provides the engine with the set of symbolic steps that can legally occur at this point of the execution. Then, the engine provides the symbolic steps to *symbolic concurrency strategies*. In return, these strategies strengthen the set of constraints, and thus reduce the amount of possible parallel steps. The engine then enumerates the set of possible combinations of atomic steps satisfying the given constraints, which yields the set of all possible parallel steps (*enumerateAllPossibleParallelSteps*). The list of possible parallel steps is given to the second set of enabled strategies, namely *operational concurrency strategies*. In return, these strategies simply remove parallel steps that do not satisfy certain criteria.

It is easy to see how this part of the execution flow can be implemented for the different metalanguages we have introduced earlier in this paper for specifying concurrent operational semantics:
- MoCCML already represents a concurrency model as a set of constraints about what events can or must occur concurrently. This can be provided directly to the generic concurrent execution engine.

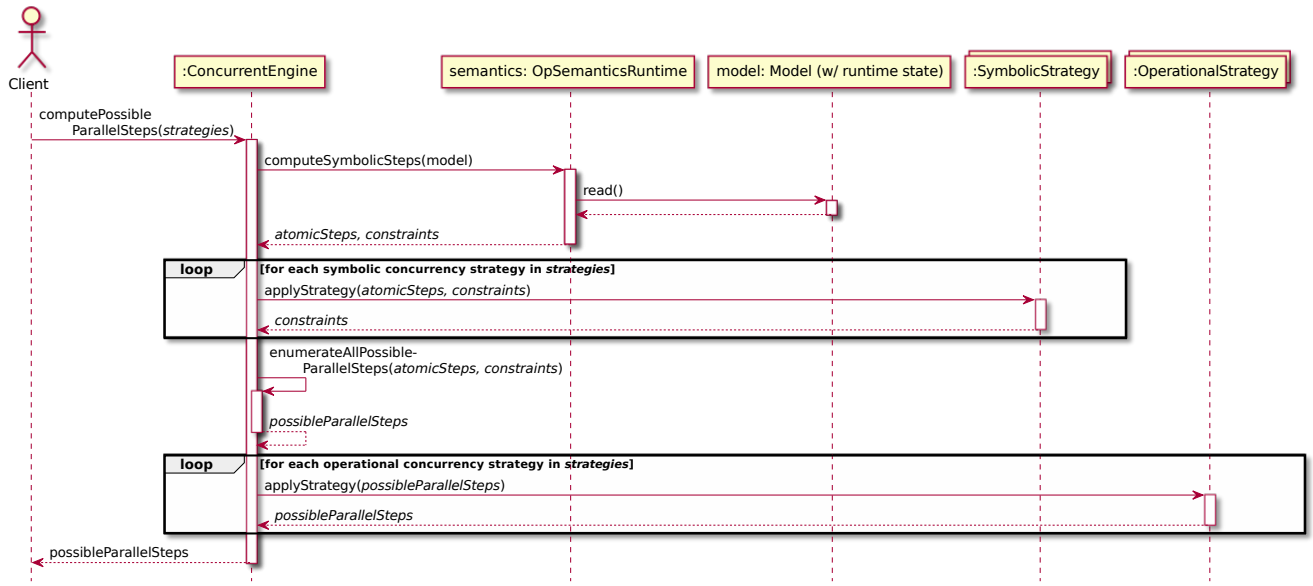

Figure 9: Sequence diagram of the *computePossibleParallelSteps* service.

– Graph-transformation–based operational semantics runtimes can use conflict analysis [31] to identify pairs of rule applications that are in conflict and must, therefore, not be executed concurrently. This information can be encoded as a set of constraints provided to the generic concurrent execution engine.

6 Implementation

We have implemented our complete approach in the GEMOC Studio language workbench for executable domain-specific modelling languages [8]. The code is open-source (EPL-1.0) and can be found on Github[11]. The GEMOC Studio uses the concept of an *execution engine* to separate the operational semantics of an xDSML from generic IDE features such as omniscient debugging or behavioural analysis. Most execution engines available for the GEMOC Studio are *sequential*; that is, they do not support the concurrent execution of steps. We have implemented a generic abstract concurrent execution engine for the GEMOC Studio, which allows support for specific concurrent metalanguages to be developed as sub-classes. Each one of these *metalanguage integrations* provides, for all xDSMLs implemented using the corresponding metalanguage, the complete interface for concurrent operational semantics runtimes presented in Sect. 4 and Sect. 5. Figure 10 shows a screenshot of the GEMOC Studio running the motivating example model.

To integrate a new concurrent metalanguage in the GEMOC studio, a sub-class of the generic abstract concurrent execution engine needs to implement two methods (*cf.* Figure 11):

1. `computeInitialLogicalSteps()` corresponds to *computeSymbolicSteps* in Figure 8. It returns a Choco [25] `Model` encoding a constraint over a set of `SmallStepVariables`—special boolean variables that are each linked to a specific `SmallStep` object.[12]
2. `executeSmallStep(smallStep)` corresponds to *executeAtomicStep* in Figure 8.

The generic concurrent execution engine can be configured with a set of concurrency strategies, which are automatically applied to the symbolic steps. Strategies must implement the appropriate one of three possible interfaces to provide their functionality (*cf.* Figure 11):

1. *Symbolic concurrency strategies:*
   1. `ConcurrencyStrategy::canBeConcurrent(step1, step2)` returns false if the strategy wishes to veto concurrent execution of the two `SmallSteps`.
   2. `SymbolicFilteringStrategy::filterSymbolically (symbolicPossibleSteps)` can add further constraints to the given set of symbolic steps.
2. `EnumeratingFilteringStrategy::filter(steps, stepComparator)` returns an operationally filtered version of the set of `ParallelSteps` provided. The

[11] Links to the Github repositories and to a working build of the implementation can be found in the companion web page: http://gemoc.org/concurrency2021/

[12] In the GEMOC Studio, `SmallStep` objects represent atomic steps.

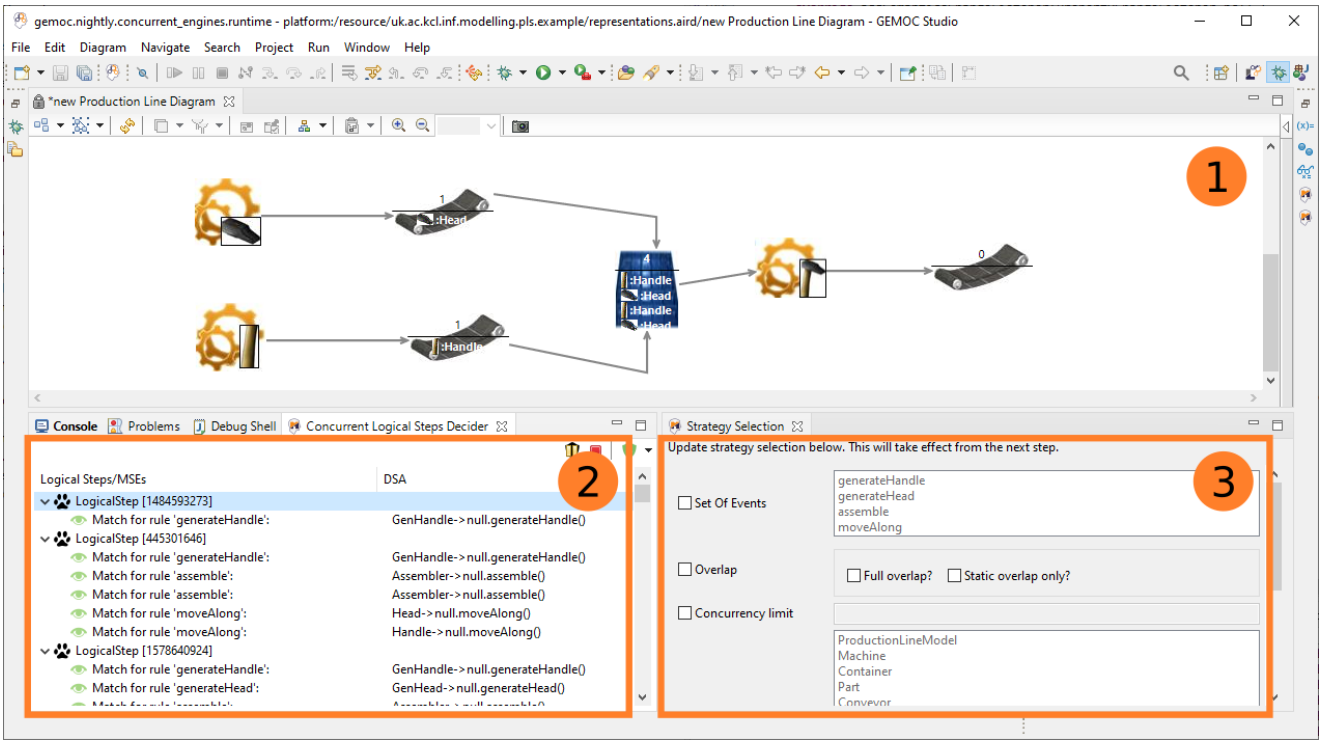

Figure 10: Screenshot of the GEMOC Studio running the motivating example with the Henshin operational semantics. Area (1) shows the current runtime state in the Sirius editor. Area (2) shows the possible logical steps to be taken next and allows the user to select the step to take. Finally, Area (3) allows the concurrency strategies to be selected and configured.

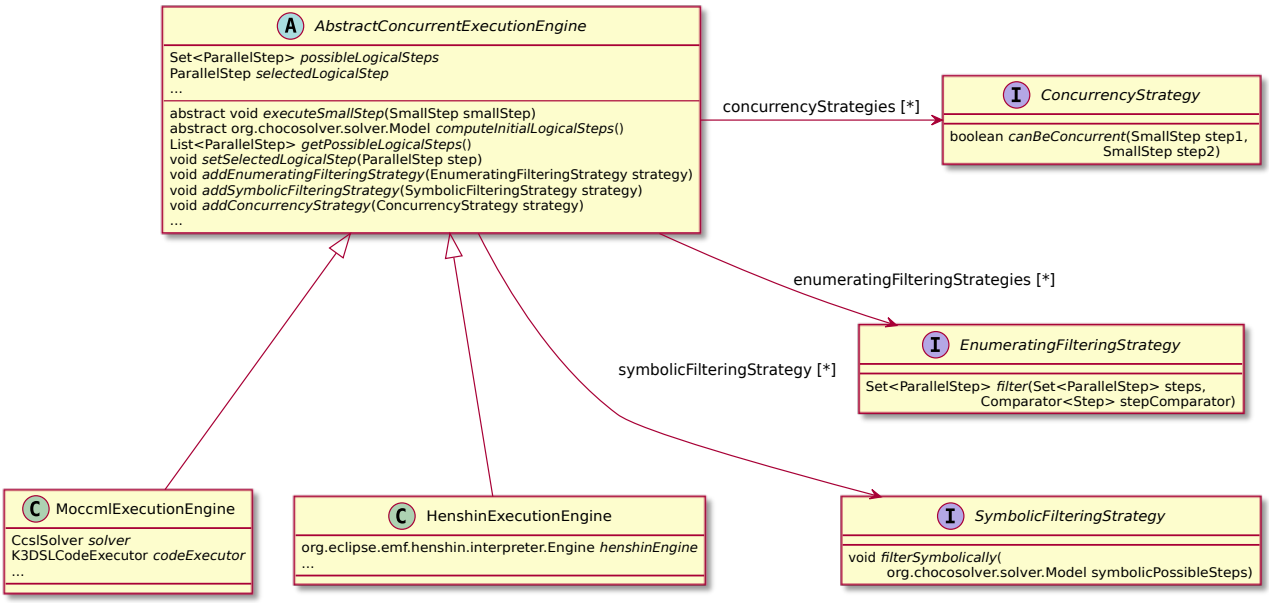

Figure 11: GEMOC concurrent engine API, part of the proposed protocol

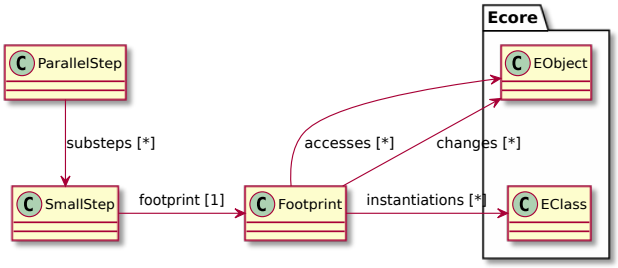

Figure 12: Meta-model for parallel steps in GEMOC

given comparator can be used to check equality of atomic steps.[13]

We have implemented the strategies discussed in Sect. 5. All strategies are dynamically managed by a central strategy registry, so it is easy to add new strategies. Many strategies are parameterized and can be configured. Strategies can be selected and configured initially when a launch configuration is defined. During the model execution, the user can also change dynamically the strategy selection and configuration through a dedicated view (Area (3) in Figure 10).

To allow the definition of strategies that make decisions based on what part of the runtime state will be accessed or changed by a step (*e.g.,* the Overlap strategy), we have extended the internal GEMOC API so that each `SmallStep` is associated with a `Footprint` that records where in the model the step will affect (*cf. Figure 12*). It is the responsibility of the operational semantics runtime to fill this footprint. Currently, the Henshin-based implementations does so already. For MoCCML, we will extend, in future work, Kermeta 3 (the underlying implementation language for atomic steps) to provide annotation of operations indicating the footprint.

We have integrated two concurrent metalanguages in GEMOC Studio, namely Henshin and MoCCML. These are implemented as sub-classes of `Abstract-ConcurrentEngine`. The implementations are available on Github.

7 User Scenarios for Concurrency Strategies

In Sect. 5, we have introduced the idea of concurrency strategies as a tool for dynamically exploring the "raw" concurrency model. In this section, we discuss user scenarios exemplifying how these strategies could be used—providing some evidence of the benefits this new concept offers. We discuss user scenarios from the perspective of two different types of users: 1. *language engineers* design new modelling languages and develop their supporting infrastructure (editors, debuggers, interpreters, compilers, . . .), while 2. *language users* use pre-defined modelling languages to create, manipulate, analyse, and execute models.

7.1 Concurrency strategies for language engineers

Language engineers can use concurrency strategies to enrich the semantics of their languages. Concretely, we envision two such scenarios:

1. *Compensating for limitations in existing semantics-specification formalisms.* Existing formalisms for defining language semantics can have limited expressivity to constrain the potential concurrency in execution of any given model. For example, MoCCML-based semantics struggle to define how concurrency constraints change in response to data values while Henshin-based semantics struggle to express scoping of concurrency to particular areas in a model. Both approaches cannot easily capture limits to the number of steps that can occur in parallel or notions such as token elements.
   Using our concurrency strategies, language engineers can decouple the representation of aspects of the concurrency model from the representation of other aspects of the language semantics, choosing the most appropriate mechanism for each.
2. *Refining the concurrency model of a pre-defined language.* Language engineers may wish to refine the concurrency model of a given language—for example to create a language variant that takes into account the concurrency limitations of a particular execution platform (e.g., a limited number of processors for parallel execution). This could be achieved by redefining the core semantics specification of the language, but this can be cumbersome and may require touching a significant proportion of specification rules. Alternatively, language engineers could refine a language's concurrency semantics by packaging the language with a set of concurrency strategies.

Such language extensions are easily enabled by adding a hook method [22] to the abstract concurrent engine, which can instantiate a set of concurrency strategies to always be enabled for this engine. The second scenario above can then be easily supported by creating a new type of concurrent engine that wraps another engine, applying the given set of concurrency strategies. Figure 13 summarises the necessary additions to the concurrent engine classes.

[13] This may require mechanisms specific to the operational semantics runtime.

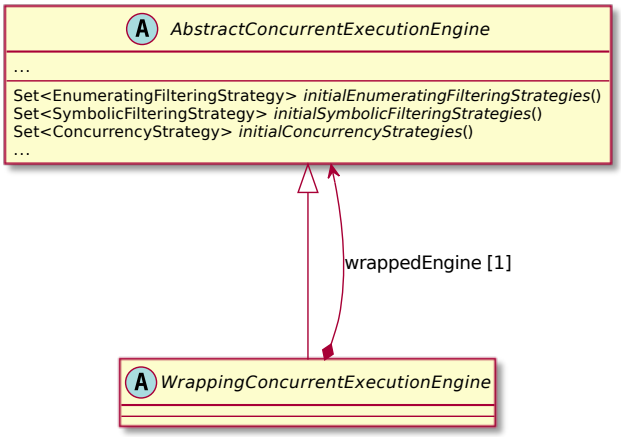

Figure 13: Template methods for pre-defining and refining concurrency strategies

7.2 Concurrency strategies for language users

As a language user, concurrency strategies are useful as part of the interactive exploration of models. Similarly to the "refining concurrency model" scenario above, a language user may wish to explore the appropriate dimensioning of hardware by exploring the impact of different additional concurrency constraints. This can be done by enabling appropriate concurrency strategies before analysing models (e.g., adding a concurrency limit strategy to explore the impact of hardware with a specific maximum number of available cores).

More generally, concurrency strategies can be useful as part of interactive debugging because they can be used to quickly reach a particular runtime state of interest, from which to debug the model behaviour in more detail. Because concurrent models can capture a potentially exponentially large state space using a static constraint similar to conditional breakpoints is not feasible. Instead, it may be necessary to step through a sequence of steps to reach a particular runtime state. Manually stepping through model behaviours for this becomes difficult as the number of possible events rises. Our concurrency strategies allow the language user to limit the choice making it easier to reach a runtime state of interest. The video on the companion web page[14] shows an example of this user scenario.

[14] `http://gemoc.org/concurrency2021/`

8 Genericity of the concurrency model representation

In this section, we present how we evaluated the genericity of our approach through our implementation in the GEMOC Studio language workbench.

8.1 Research questions

As stated early in Sect. 1, the main objective of this work is to provide a *generic* interface for operational-semantics runtimes and a *generic* execution flow for concurrent execution of xDSMLs. In particular, such a generic solution must be able to deal with both concurrency semantics based on *implicit* concurrency models, and concurrency semantics based on *explicit* concurrency models. Accordingly, we evaluated our approach through the following two research questions:

Concurrency model independence: How independent are the proposed framework, execution flow, and strategies of the way the concurrency model of a concurrent operational semantics of a considered xDSML was defined (*i.e.,* defined implicitly or defined explicitly)?

Tools definition: How well can the proposed framework be used to define relevant analysis tools for concurrent model execution, regardless of the metalanguages used to define the concurrent operational semantics of a considered xDSML?

8.2 Experimental setup

The evaluation was done using the implementation of the presented approach for the GEMOC Studio language workbench. This implementation is presented in Sect. 6.

Considered metalanguages To demonstrate that the approach is able to work with both implicit and explicit concurrency models, we considered two very different metalanguages, namely Henshin and MoCCML. As already explained in Sect. 2, these two metalanguages each take a very different approach to the definition of the concurrency model of the operational semantics—one through an implicit definition based on transformation rules, the other through an explicit definition of the conditions under which an event can occur. As such, they aptly cover an interesting part of the spectrum of possible concurrent metalanguages.

Considered xDSMLs and models To actually run and test the implementation of the approach, actual xDSMLs are required, along with executable models conforming to said xDSMLs. For this evaluation, we considered two different xDSMLs: (1) the xDSML for production systems previously introduced in Sect. 2, and (2) SigPML, an xDSML dedicated to data flow processing, based on blocks, ports and connectors. This required implementing one variant of the operational semantics of each xDSML per considered metalanguage, thus one variant in Henshin and one variant in MoCCML for each xDSML. Regarding the executable models, we considered one model per xDSML, including the example model shown in Figure 2 for the production systems xDSML. Details on the considered xDSMLs and models can be found in the companion webpage[10].

Considered execution scenarios Since we are dealing with concurrent xDSMLs, each considered executable model can lead to many different execution traces due to parallelism or interleaving. In addition, the presented aproach provides a set of strategies that, when enabled, may alter the presented set of available parallel steps. This further expands the list of possible user actions during the execution of a model. For this evaluation, we therefore identified a set of interesting execution scenarios for each considered model. Each scenario follows the following structure: (1) start the execution of a model with an xDSML, (2) apply a specific sequence of parallel steps, in order to reach a point where the amount of possible parallel steps to choose from is too large, (3) undo the last performed parallel step, (4) enable a specific set of strategies, in order to explore, from this point on, a specific part of the concurrency model, (5) re-do the parallel step again, observe that the set of possible parallel steps is now reduced, and arguably significantly easier to choose from. More information on the exact scenarios can be found in the companion webpage[10].

8.3 Experiments

Concurrency model independence To answer the first research question, we used the proposed framework to integrate both considered metalanguages in the GEMOC Studio language workbench. The integration of both Henshin and MoCCML is presented in Sect. 6.

Tools definition To demonstrate that the framework proposed in our approach can be used to define relevant tools for concurrent model execution, we implemented a *generic concurrent omniscient debugger*. This omniscient debugger provides a graphical view showing the execution traces obtained from the execution of the model, and shows what are the possible parallel steps at any instant of the execution. More interestingly, it gives the possibility to easily "go back in time" into a previous runtime state, and to choose from there an alternate execution path using another parallel step that was possible then. Thereby, a language user or engineer can explore and compare as needed all the different execution traces that an executed model may produce. The GEMOC concurrent omniscient debugger user interface is shown in Figure 14.

Note that this tool was built using the *addon* mechanism of the GEMOC Studio [8]. An addon is a component attached to an execution engine of the GEMOC Studio, and that is notified of each and every parallel or atomic step executed by the engine. For the concurrent omniscient debugger, these notifications are used to construct an execution trace that can then be used for restoring previous runtime states and previous possible parallel steps.

For both experiments, we manually conducted all considered execution scenarios—each relying on the generic concurrent omniscient debugger for undoing a parallel step—on both considered integrations (*i.e.,* Henshin and MoCCML) and on both considered xDSMLs and models.

8.4 Results

Concurrency model independence Both integrations of Henshin and MoCCML worked successfully: all the generic code used to drive the execution flow and the use of strategies worked as expected in both cases. We can therefore answer that the proposed approach can work for both concurrency semantics based on implicit concurrency models, and concurrency semantics based on explicit concurrency models.

As a complementary note, we can observe that the generic execution engine (*i.e.,* the reusable code that does not have to be re-written for each metalanguage) is made of 589 LoC (Lines of Code), while the Henshin metalanguage integration is only 182 LoC and the Moccml metalanguage integration is only 256 LoC. Thus, the majority of the implementation code is generic and not specific to any metalanguage, and integrations can be built atop the framework with reasonable amounts of efforts.

Tools definition The concurrent omniscient debugger worked as planned after testing, with both variants of each xDSML (*i.e.,* the Henshin variant and the MoCCML variant). No code specific to these metalanguages

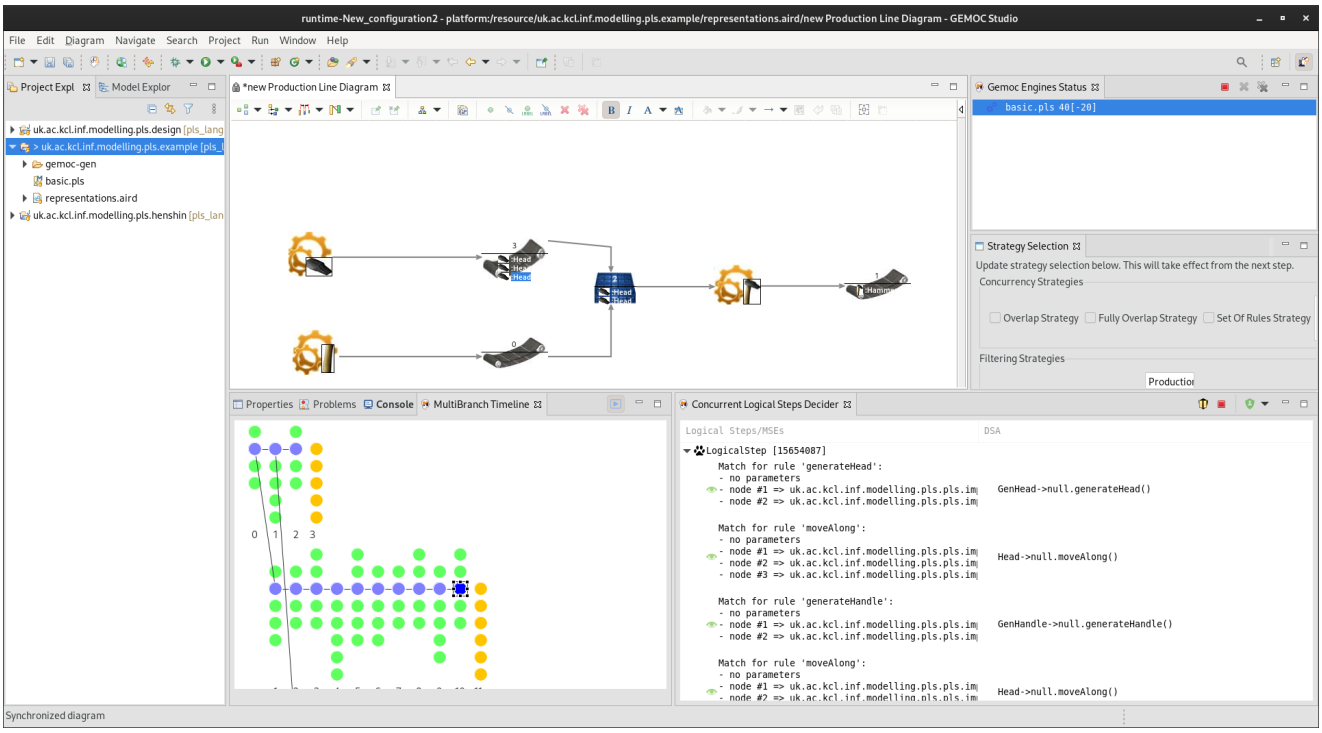

Figure 14: In the bottom left corner, the concurrent omniscient debugger view in the GEMOC Studio, while executing a model using the Henshin integration. A blue circle represent a reached runtime state. A green circle represent a possible parallel step that was not taken during a runtime state. A yellow circle is a possible parallel step that can be executed from the last reached runtime state. Double-clicking on a yellow circle advances the execution with one parallel step, while doucle-clicking on a blue circle restores a previous runtime state and starts a new execution branch.

was needed in the implementation of the tool, which means that this tool can be reused for the concurrent execution of any model executed in the GEMOC Studio, provided that the metalanguages used for the operational semantics were well integrated. We can therefore answer that the proposed framework can be used to define relevant tools for concurrent model execution, regardless how was defined the concurrency model of the the concurrent operational semantics of the considered xDSML.

9 Related work

Much work has been done on the design and implementation of executable DSLs. In this paper, we proposed a conceptual and technical framework to explore concurrency independently of the way the concurrent operational semantics has been defined. This section presents related work in the field of language design and implementation. It also addresses some existing approach in the field of Multi Agent Systems where concurrency has been explicitly managed, and some existing work in the field of graph rewriting.

A language workbench is a software package for designing software languages [50]. For instance, it may encompass parser generators, modern editors (*e.g.*, with completion, quick fix), DSLs for expressing the behavioural semantics and others. Early language workbenches include Centaur [6], ASF+SDF [28], and TXL [12]. More recent proposals include Generic Model Environment (GME) [44], Metacase's MetaEdit+ [45], Microsoft's DSL Tools [11], Krahn et al's Monticore [29], Kats and Visser's Spoofax [26], Jetbrain's MPS [49]. While more and more elaborated, such language workbenches rarely focus on the debugging of models. This preoccupation related to domain specific debugging is recent [36]; and few existing approaches propose debugging as a software package. For instance MetaEdit+ provides a specific API as a crude way to debug/animate models from an external software and Spoofax provides a simple hard coded debugger. We can also cite the Debugger Adapter Protocol (DAP [15]) proposed by Microsoft, which

[15] `https://microsoft.github.io/debug-adapter-protocol/`

defines the services which are required to enable the debugging of a program independently of the language it conforms to. More elaborated approaches like [7,9,47, 13] proposed to augment such protocol to provide omniscient debugging; *i.e.*, a way to navigate (forward and backward) in a sequential execution.

From these approaches, it is not possible to explore the impact of concurrency on the system behaviour since they consider a single execution trace. There exist approaches that focus on the debugging of concurrent systems [37,18,19]. However, all of these approaches focused only on a "computer science notion of concurrency"; that is, they reified the technical artefacts found in traditional operating systems or middleware (*e.g.,* Thread, Process, Fork, Join). In contrast, we relied on the notion of interleavings and parallelism between atomic steps, an atomic step being an abstraction of any change in a model runtime state. This kind of reasoning about the order of relevant events is inspired by Tagged Signal Model [35] and more recent works on logical time [1,15], which proved to be adaptable to different notions of concurrency from different domains. Consequently, we abstracted away from technical artefacts of concurrency to keep only a simple notion of atomic step (comparable to an event) and parallel steps (comparable to synchronous events). This allowed us to align the omniscient and concurrent debug protocol directly on the definition of the language semantics rather than on abstractions over the execution of the underlying models / programs.

In the domain of Multi Agent Systems, concurrency is a main concern. However, like in other domains, most of the existing approaches to debug such systems focused on technical artefacts (*e.g.,* the Mailbox state). However, since these approaches can be massively parallel, the idea of concurrency presentation appeared. In [46], they proposed different abstraction levels for the presentation of the concurrency to the user of the debugger. These abstractions are domain specific (*e.g.,* Agent view or Interaction View) but the goal was to present the relevant information (which is activity dependent) to the user. In comparison, our notion of filtering strategy shares the same goal to provide relevant information to the user that debugs the system. However we used it not only for representation purpose but also to enable more focused exploration of possible executions.

In graph rewriting, semantics of specifications have been based on the notion of *unfolding* [5] for a considerable time. The key idea here is that the semantics of a graph grammar (an initial graph and a set of graph-transformation rules) is given by the set of traces that are given by all sequences of rule applications starting from the initial graph. Work on unfolding explores compact representations of these trace sets for analysis, using occurrence grammars. In practical tooling contexts, this has been used to generate state-space diagrams from a given graph grammar. These are directed graphs, where edges represent rule applications and nodes represent graphs being transformed; where different paths lead to equivalent graphs, these are represented by the same node in the state-space diagram. For example, the Henshin tool used in one of the concurrency engines in this paper, supports the generation of state-space diagrams as well as their integration with a model-checking tool [4]. This is similar to the trace model built up by the omniscient debugger in our work. In contrast to the works cited, however, in our approach the trace model can be used in a generic omniscient debugger to go backwards and forwards in "time". Moreover, we can construct trace models irrespective of whether the semantics are captured using graph transformations. Finally, we introduce the possibility of using concurrency strategies to flexibly and dynamically shape the concurrency model. While graph transformation systems offer higher-level control structures (*e.g.,* units in Henshin [4]), these need to be statically coded and do not allow the kind of control of concurrency offered by our concurrency strategies. Using such control structures to capture behaviour patterns to match for to select traces of interest has been shown in the context of Maude in [38]. This is similar in intent to our concurrency strategies but focuses on capturing sequential patterns rather than filtering the concurrent occurrence of steps.

10 Conclusions

We presented a generic interface for concurrent operational semantics runtimes, allowing them to be plugged into a language workbench for concurrent xDSMLs. This enables any such xDSML to be supported by concurrent omniscient debugging and dynamic analysis services with minimal additional implementation effort and based on an explicitly modelled operational semantics.

Specifically, we have demonstrated how the generic interface enables the introduction of strategies that can be used flexibly and dynamically to explore a language's concurrency model. This can be useful in multiple scenarios (and we aim to further explore each of these scenarios in future work):

1. *Language engineering.* Language engineers need to ensure they have specified the right semantics for their language. This is best achieved in an incremental manner, where different alternative semantics are explored with example models before the final

choice is made. The ability to dynamically restrict the concurrency model of a language without having to rewrite the formal specification allows quicker exploration of alternatives. An interesting research question is whether it is possible to take a given choice of strategies and (semi-)automatically suggest changes to the underlying semantics specification that will achieve the same effect.

2. *Model engineering.* Modellers need to ensure the models they have created do indeed capture the behaviour they are interested in. This can be understood using model debuggers. Where problems related to concurrency are encountered, the different strategies provide a useful tool for clarifying the concurrency that is actually intended before trying to understand how the model can be changed to achieve this intended concurrency. An interesting research question is whether a given set of strategies and a model can be (semi-)automatically translated into a set of proposed model changes to encode the intended concurrency in the model.
3. *Dynamic analysis.* Dynamic analysis tools often need to explore a large number of execution traces, and limiting the set of traces that need exploring can substantially increase the efficiency of the analysis. An interesting research question is how our concurrency strategies can be used by dynamic analysis tools to tactically constrain the set of traces to be analysed so that problematic traces can be identified more efficiently.

Our generic interface is based on a concurrency model of atomic actions. This concurrency model is generic enough to express different concurrent semantics, including those with durative actions. However, it may be too low level to provide a convenient interface for understanding, analysing, or debugging models. An interesting research question is how one can build on our generic interface to support more sophisticated concurrency models—for example, in the way specifically prototyped for durative and periodic actions in the context of *e-Motions* [39].

Acknowledgements

The authors wish to thank Kinga Bojarczuk for contributing to an early prototype of these ideas.

References

1. André, C.: Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA (2009). URL `https://hal.inria.fr/inria-00384077`
2. André, C., DeAntoni, J., Mallet, F., de Simone, R.: The Time Model of Logical Clocks Available in the OMG MARTE Profile, pp. 201–227. Springer US, Boston, MA (2010). DOI 10.1007/978-1-4419-6400-7_7. URL `https://doi.org/10.1007/978-1-4419-6400-7_7`
3. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (eds.) Model Driven Engineering Languages and Systems, pp. 559–573. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
4. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: D. Petriu, N. Rouquette, Ø. Haugen (eds.) Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MoDELS'10), *LNCS*, vol. 6394, pp. 121–135. Springer (2010). DOI 10.1007/978-3-642-16145-2_9. URL `https://doi.org/10.1007/978-3-642-16145-2_9`
5. Baldan, P., Corradini, A., Montanari, U., Ribeiro, L.: Unfolding semantics of graph transformation. Information and Computation **205**(5), 733–782 (2007). DOI 10.1016/j.ic.2006.11.004. URL `http://www.sciencedirect.com/science/article/pii/S0890540106001611`
6. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: 3rd ACM software engineering symposium on Practical software development environments, pp. 14–24. ACM (1988)
7. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xdsmls. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 137–148 (2015)
8. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: Proc. ACM SIGPLAN Int'l Conference on Software Language Engineering (SLE'16), pp. 84–89 (2016)
9. Bousse, E., Leroy, D., Combemale, B., Wimmer, M., Baudry, B.: Omniscient debugging for executable dsls. Journal of Systems and Software **137**, 261–288 (2018)
10. Combemale, B., DeAntoni, J., Larsen, M.V., Mallet, F., Barais, O., Baudry, B., France, R.B.: Reifying concurrency for executable metamodeling. In: M. Erwig, R.F. Paige, E.V. Wyk (eds.) Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 8225, pp. 365–384. Springer (2013). DOI 10.1007/978-3-319-02654-1_20. URL `https://doi.org/10.1007/978-3-319-02654-1_20`
11. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional (2007)
12. Cordy, J.R., Halpern, C.D., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. In: Conf. Int Computer Languages, pp. 280–285 (1988)
13. Corley, J.A.: Exploring efficient and scalable omniscient debugging for mde. Ph.D. thesis, University of Alabama Libraries (2016)
14. Corradini, A., Heckel, R., Montanari, U.: Graphical operational semantics. In: Proc. Workshop on Graph Transformation and Visual Modelling Techniques (2000)
15. Deantoni, J., André, C., Gascon, R.: CCSL denotational semantics. Research Report RR-8628, Inria (2014). URL `https://hal.inria.fr/hal-01082274`
16. Deantoni, J., Diallo, P.I., Champeau, J., Combemale, B., Teodorov, C.: Operational Semantics of the Model

of Concurrency and Communication Language. Research Report RR-8584, INRIA (2014). URL https://hal.inria.fr/hal-01060601
17. Deantoni, J., Issa Diallo, P., Teodorov, C., Champeau, J., Combemale, B.: Towards a Meta-Language for the Concurrency Concern in DSLs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE). Grenoble, France (2015). URL https://hal.inria.fr/hal-01087442
18. Dotan, D., Kirshin, A.: Debugging and testing behavioral uml models. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA '07, p. 838–839. Association for Computing Machinery, New York, NY, USA (2007). DOI 10.1145/1297846.1297915. URL https://doi.org/10.1145/1297846.1297915
19. Elmas, T., Burnim, J., Necula, G., Sen, K.: Concurrit: A domain specific language for reproducing concurrency bugs. SIGPLAN Not. **48**(6), 153–164 (2013). DOI 10.1145/2499370.2462162. URL https://doi.org/10.1145/2499370.2462162
20. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Computer Languages, Systems & Structures **44**, 24–47 (2015). DOI 10.1016/j.cl.2015.08.007. URL http://www.sciencedirect.com/science/article/pii/S1477842415000573. Special issue on the 6th and 7th Int'l Conf Software Language Engineering (SLE 2013 and SLE 2014)
21. Fowler, M.: Domain-specific languages. Pearson Education (2010)
22. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison Wesley Professional (1995)
23. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: Proceeding of the 33rd international conference on Software engineering - ICSE '11. ACM Press (2011). DOI 10.1145/1985793.1985875. URL https://doi.org/10.1145/1985793.1985875
24. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of metalanguages and its implementation in the kermeta language workbench. Software & Systems Modeling **14**(2), 905–920 (2015)
25. Jussien, N., Rochart, G., Lorca, X.: choco: An open source java constraint programming library. In: CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08), pp. 1–10 (2008). URL https://hal.archives-ouvertes.fr/hal-00483090
26. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA '10, pp. 444–463. ACM (2010). DOI 10.1145/1869459.1869497. URL http://doi.acm.org/10.1145/1869459.1869497
27. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM **19**(7), 371–384 (1976)
28. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM **2**(2), 176–201 (1993)
29. Krahn, H., Rumpe, B., Volkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns, LNBIP. Springer (2008)
30. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track (2002)
31. Lambers, L., Kosiol, J., Strüber, D., Taentzer, G.: Exploring conflict reasons for graph transformation systems. In: E. Guerra, F. Orejas (eds.) Proc. 12th Int'l. Conf. on Graph Transformations (ICGT'19), pp. 75–92. Springer International Publishing (2019)
32. Latombe, F., Crégut, X., Combemale, B., Deantoni, J., Pantel, M.: Weaving concurrency in executable domain-specific modeling languages. In: Proc ACM SIGPLAN Int'l Conf Software Language Engineering (SLE'15), pp. 125–136 (2015). DOI 10.1145/2814251.2814261. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84962533506{&}partnerID=40{&}md5=9c4a9c76eb479a24bb42a8e8ef371e4a
33. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for system design. Journal of Circuits, Systems, and Computers **12**(03), 261–303 (2003)
34. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on computer-aided design of integrated circuits and systems **17**(12), 1217–1229 (1998)
35. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Transactions on computer-aided design of integrated circuits and systems **17**(12), 1217–1229 (1998)
36. Mannadiar, R., Vangheluwe, H.: Debugging in domain-specific modelling. In: International Conference on Software Language Engineering, pp. 276–285. Springer (2010)
37. Marr, S., Torres Lopez, C., Aumayr, D., Gonzalez Boix, E., Mössenböck, H.: A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools. In: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, DLS 2017, p. 3–14. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3133841.3133842. URL https://doi.org/10.1145/3133841.3133842
38. Rivera, J.E., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation **85**(11–12), 778–792 (2009). DOI 10.1177/0037549709341635
39. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'09), pp. 51–55. IEEE (2009). DOI 10.1109/VLHCC.2009.5295300
40. Rivera, J.E., Durán, F., Vallecillo, A.: On the behavioral semantics of real-time domain specific visual languages. In: P.C. Ölveczky (ed.) Rewriting Logic and Its Applications, pp. 174–190 (2010)
41. Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
42. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2 edn. Eclipse Series. Addison-Wesley (2009). URL https://www.safaribooksonline.com/library/view/emf-eclipse-modeling/9780321331885/
43. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: J. de Lara, D. Plump (eds.) Proc. 10th Int'l Conf on Graph Transformations (ICGT'17), pp. 196–208. Springer (2017)

44. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer **30**(4) (1997)
45. Tolvanen, J., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference OOPSLA, pp. 92–93. ACM (2003)
46. Van Liedekerke, M.H., Avouris, N.M.: Debugging multi-agent systems. Information and Software Technology **37**(2), 103 – 112 (1995). DOI https://doi.org/10.1016/0950-5849(95)93487-Y. URL `http://www.sciencedirect.com/science/article/pii/095058499593487Y`
47. Van Mierlo, S.: A multi-paradigm modelling approach for engineering model debugging environments. Ph.D. thesis, Universiteit Antwerpen (2018)
48. Viyović, V., Maksimović, M., Perisić, B.: Sirius: A rapid development of DSM graphical editor. In: IEEE 18th Int'l Conf Intelligent Engineering Systems (INES'14), pp. 233–238 (2014). DOI 10.1109/INES.2014.6909375
49. Voelter, M., Solomatov, K.: Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In: SLE, LNCS. Springer (2010)
50. Volter, M.: From Programming to Modeling-and Back Again. Software, IEEE **28**(6) (2011)
51. Ward, M.P.: Language-oriented programming. Software-Concepts and Tools **15**(4), 147–161 (1994)
52. Winskel, G.: Event structures. advances in petri nets, lncs 255: 325–392 (1987)
53. Zschaler, S.: Adding a HenshinEngine to GEMOC Studio: An experience report. In: Proc. 6th Int'l Workshop on The Globalization of Modeling Languages (GEMOC'18) (2018)