Softw Syst Model (2019) 18:691–720
https://doi.org/10.1007/s10270-017-0590-0

REGULAR PAPER

Reusable specification templates for defining dynamic semantics of DSLs

Ulyana Tikhonova[1]

Received: 5 October 2016 / Revised: 25 February 2017 / Accepted: 2 March 2017 / Published online: 21 March 2017

Abstract In the context of model-driven engineering, the dynamic (execution) semantics of domain-specific languages (DSLs) is usually not specified explicitly and stays (hard) coded in model transformations and code generation. This poses challenges such as learning, debugging, understanding, maintaining, and updating a DSL. Facing the lack of supporting tools for specifying the dynamic semantics of DSLs (or programming languages in general), we propose to specify the architecture and the detailed design of the software that implements the DSL, rather than requirements for the behavior expected from DSL programs. To compose such a specification, we use specification templates that capture software design solutions typical for the (application) domain of the DSL. As a result, on the one hand, our approach allows for an explicit and clear definition of the dynamic semantics of a DSL, supports separation of concerns and reuse of typical design solutions. On the other hand, we do not introduce (yet another) specification formalism, but we base our approach on an existing formalism and apply its extensive tool support for verification and validation to the dynamic semantics of a DSL.

Keywords Domain-specific language · Dynamic semantics · Specification template · Generic programming · Aspect-oriented programming

Communicated by Prof. Tony Clark.

✉ Ulyana Tikhonova
ulyana.tihonova@gmail.com

1 Technische Universiteit Eindhoven, P.O. Box 513, 5600, MB, Eindhoven, The Netherlands

1 Introduction and motivation

A *domain-specific language* (DSL) is a computer language specialized for a specific (application) domain. The idea of using DSLs for software development and/or software configuration is not new, and DSLs have been known and applied in various forms (such as subroutine libraries, frameworks, and dedicated languages) for a long time. Recently, DSLs became a central concept of *Model Driven Engineering* (MDE). In the context of MDE, a DSL determines a class of models that can be constructed in the domain, and model-to-model transformations and code generators assign meaning to such models by automatically translating them into various artifacts, such as documentation, source code, visualizations, formal specifications.

The role of DSLs is twofold. On the one hand, a DSL captures domain knowledge, which supports its reuse via domain notions and notation and raises the abstraction level of solving problems in a particular domain. In other words, the DSL realizes a so-called *vertical domain* [24]. On the other hand, the implementation of the DSL (such as its translation to the source code, or via interpretation) captures software solutions (algorithms, architecture, and techniques) that are commonly used in the domain, which supports their reuse and, thus, raises the efficiency of the software development process. In this way, the DSL realizes a so-called *horizontal domain* [24].

In this work we look into the definition of the *dynamic semantics* of DSLs. Dynamic semantics maps each DSL model (program) to the corresponding execution behavior. Thus, we consider DSLs that can be used for programming, that is, for specifying programs that can be executed. The DSL dynamic semantics is implemented as a translation from the DSL to the input language of a target execution platform. From a semantics point of view, the gap bridged by this translation can be quite wide, as such a translation implements both the horizontal and vertical aspects of the DSL and does it in terms of both high-level concepts of the DSL and low-level concepts of the execution platform. By giving an explicit definition of the dynamic semantics of a DSL, we aim to manage the complexity of the DSL translation, which in the context of MDE is usually (hard)coded in model transformations and code generation.

Moreover, in practice a DSL implementation can include a number of such DSL translations, targeting different execution platforms with the purpose of achieving diverse technological goals. For example, one translation generates C/C++ or Java source code for execution of DSL programs; another translation targets various formalisms for verification and formal analysis of DSL programs; and a third translation constructs diagrams visualizing DSL programs [54]. Generally speaking, there is no guarantee that different translations implement the DSL dynamic semantics in a coherent way. The desire to have such translations implemented in a consistent way poses a maintenance problem. We strive toward a definition of the dynamic semantics of a DSL that provides a common ground for different translations and in this way facilitates their consistency.

There exist a number of approaches for defining the dynamic semantics of *general purpose languages* (GPLs), such as denotational and algebraic semantics [33,56], action semantics [30], and structural operational semantics (SOS) [39]. Compared to GPLs, DSLs are smaller languages: a DSL covers a smaller set of problems and, as a consequence, has a smaller audience of practitioners (those who use the DSL) and/or a smaller group of developers (those who design and implement the DSL). Next to the known advantages of using such small languages [14], the main disadvantages are determined by the costs of developing and learning a DSL. An explicit definition of the DSL dynamic semantics can mitigate these disadvantages by providing various practical outcomes of having a formal specification of the dynamic semantics of a DSL. However, the listed approaches for defining dynamic semantics are hardly suited for realizing such goal, as they do not have practically applicable tool support.

Aiming for practical benefits of having a formal definition of the DSL dynamic semantics, in our previous work [51,52], we employed a formalism that has extensive tool support. Specifically, we defined the dynamic semantics of a real-life industrial DSL as a translation to the Event-B formalism [2]. The Rodin platform [3] and its various plug-ins offer a wide range of functionality that can be applied to an Event-B specification of the DSL. For example, the DSL dynamic semantics can be prototyped and then analyzed using automatic provers and model checkers; DSL programs can be simulated and debugged using animators and visualization tools. We have observed that although the available tools facilitate design and usage of the DSL, the semantic gap between the DSL and Event-B is quite wide, since this formalism is not designed for specifying dynamic semantics of DSLs (or GPLs), and the definition of the dynamic semantics is kept (coded) in the DSL-to-Event-B translation (in our case, model transformation).

To manage the wide semantic gap between a DSL and a specification formalism, we break it down and introduce an intermediate step (modeling layer) in such a translation. As the intermediate layer, we use software (design) solutions that implement the DSL, *i.e.,* the DSL horizontal domain. As a result, the first step of our translation (defining the dynamic semantics of a DSL) is a mapping of DSL vertical concepts onto its horizontal concepts. The second step of our translation captures the DSL horizontal concepts in the form of *reusable specification templates*–a library of (Event-B) specifications, each of which formalizes a separate software (design) solution. We implement this approach in our language, *Constelle*, which allows for defining the DSL dynamic semantics as a composition of such specification templates and, in this way, implements the two-step translation of the DSL to the specification formalism.

In this paper, we first reflect on the concept of the dynamic semantics of a DSL, identify the use cases of its definition, and set the corresponding criteria (requirements) for a definition of the dynamic semantics (Sect. 2.1). The idea of reusable specification templates arises from our decision to define the dynamic semantics as a software solution implementing the dynamic semantics, rather than as a set of requirements on the behavior resulting from such an implementation. Specification templates realize the *generic programming* paradigm [32] for (thorough mathematical-based) formal specifications (Sect. 2.2). For this, a specification is parameterized so that it can be further specialized with concrete (domain) data and, in this way, reused during construction and analysis of another specification. For using (invoking) such specification templates, we apply the ideas of *Aspect-Oriented Programming* [22] and consider each invocation of a (specialized) specification template as a *crosscutting concern* constituting the resulting definition of the DSL dynamic semantics (Sect. 2.3).

In Sect. 3 we explain our approach and develop the corresponding design in the form of a metamodel of a specification template, which realizes parametrization of (Event-B) specifications, and a metamodel of the Constelle language, which implements specialization and weaving of specification templates. In Sect. 4 we give a formal definition of Constelle by specifying its semantics in the form of a mapping (or practically, a QVTo model transformation) of a Constelle model to an Event-B specification. The corresponding implementation and the results that it allows to achieve are discussed in Sect. 5. In Sect. 6 we position our approach in relation to the existing work from two points of view: defining dynamic semantics as a composition of (reusable) building blocks and reuse of formal specifications. Section 7 concludes the paper and highlights directions for the future work.

2 Motivation and introduction of the proposed approach

2.1 Dynamic semantics of a DSL and its formal definition

The dynamic semantics of a DSL determines the behavior of DSL programs. For this, a definition of the dynamic semantics of a DSL consists of the following two components:

– A *semantic domain* providing terms to define the dynamic semantics;
– A *semantic mapping* mapping the DSL (metamodel or abstract syntax) to the semantic domain.

To achieve unambiguous understanding of a DSL and to enhance the DSL development with formal analysis and tool support, we would like to have the definition of both these components to be both *precise and executable*. Precision of a definition is achieved by employing a formalism based on a solid mathematical theory. Executability of a precise definition is achieved by employing tools that implement this theory.

In current practice usually at least one of the two components is either not precise or not executable: see for example definitions of the dynamic semantics of DSLs presented in [49] (the semantic mapping is defined using SOS, which is precise but not executable) and [54] (the semantic mapping is defined using Xtend, which is executable but not precise). As an exception to this practice, a precise definition can be implemented (realized) in an executable formalism: for example, in [34] all definitions of the dynamic semantics of a programming language are formalized using the proof assistant Isabelle.

As an instance of software, a DSL can be explored, designed, and described in the form of two different artifacts:

– *Requirements* that define expected (or intended) behavior of DSL programs;
– *Solution* that defines actual implementation (or software architecture) of the DSL.

Ideally, each possible solution refines the (predefined) requirements (in the general meaning of the refinement relation as properties implication). To check whether this relation actually holds, one can apply manual and automatic techniques–validation and verification correspondingly. For performing automatic verification of this refinement relation, one needs to have a formal specification of the requirements. Manual validation involves a human who can interpret the informal description of requirements (potential misinterpretation is possible). In practice, requirements are specified formally only if it is required by a standard of the development process – for example, for critical or life-threatening systems, such as railway signaling (the CENELEC standards) or automotive systems (the ISO 26262 standard).

Classical approaches of algebraic and denotational semantics allow for formally specifying the dynamic semantics of a programming language in the form of requirements rather than in the form of a solution. An operational semantics gives more insight in how a program is executed, but still abstracts from implementation strategies and machine architectures [33]. Using these formal techniques requires scientific expertise and, thus, is not expected from an average software engineer. At the same time, the costs of employing scientific expertise might not be justified if a DSL is used in a non-critical domain. Thus, the usual situation that the dynamic semantics of a DSL is not specified formally might be determined by the same circumstances as the common in software development practice of not having a formal specification of requirements.

In our approach we propose to specify a DSL dynamic semantics as a solution rather than requirements. To stress the difference between a specification of requirements and a specification of a solution, we illustrate our approach using a tangible and domain-independent analogy: the LEGO Technic construction kit. LEGO Technic allows for construction of models of moving mechanisms using LEGO pieces such as gears, pins, axles, pneumatic systems, motors and principles of mechanical engineering to assemble them together. An example of such LEGO Technic model is presented in Fig. 1a.[1] When considering requirements of the system modeled by this LEGO construction (a Jeep car), one can think about the following requirement: the car should make a smooth left (or right) turn when the steering wheel is turned left (or right). The construction presented in Fig. 1a is a model of a real-life solution: it abstracts away some implementation details and captures the key elements of the solution, which guarantee realization (refinement) of the requirements listed above. The key elements of the pictured solution are bevel gears steering system, two differentials that enable the car to turn smoothly, etc.

In the same way as a LEGO model captures the principal mechanical solution, a definition of a DSL implementation captures the DSL dynamic semantics as a principal design solution. If such a definition is precise and executable (as introduced above), then we can use it in the following use cases:

[1] Pictures of LEGO models used in the paper are taken from the web sites brickshelf.com and sariel.pl and from the book [25].

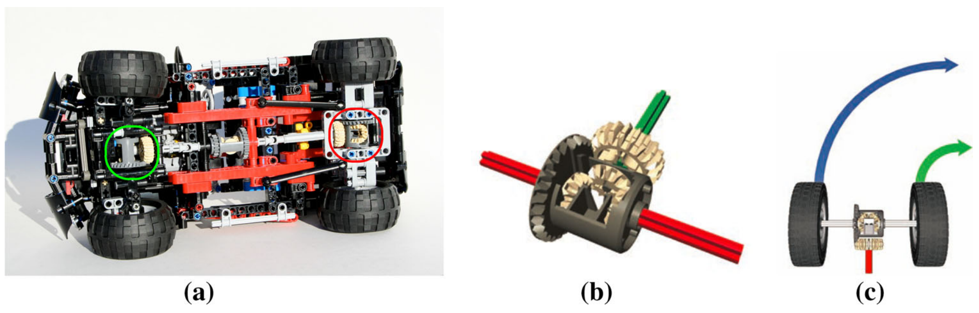


Fig. 1 Example of a LEGO Technic model and a differential pattern. **a** LEGO Technic model and the differential pattern applied, **b** differential pattern, **c** function of the differential pattern

- To prototype the DSL implementation (in LEGO: let's construct a Jeep car);
- To validate the prototyped DSL implementation against (informal) requirements by executing the definition (in LEGO: does the car drive? does the steering mechanism work as expected?);
- To check consistency of the prototyped DSL by analyzing the definition (in LEGO: how two differentials are combined in a four-wheel drive without blocking the car from moving);
- To verify the prototyped DSL against formalized requirements (or other high-level properties that the DSL should fulfill) by analyzing the definition (in LEGO: a combination of gears meshed together supports the speed and/or friction ratio as specified in requirements);
- To simulate and debug DSL programs by executing the definition (in LEGO: debug why the gears get loose after 10 min of exploitation).

Note that all these use cases require the definition of the DSL dynamic semantics (that is used for analysis and execution) to be consistent with and reflecting the actual DSL implementation. Without this consistency we cannot extend the results of analyzing and executing the definition to the actual DSL implementation and to DSL programs being debugged.

2.2 Reusable specification templates

When defining a DSL dynamic semantics as a design solution rather than requirements, we target a semantic domain that is sufficiently rich to model the programming language (or environment, or system, or platform) in which DSL programs are executed. Thus, this semantic domain represents concepts that are commonly used in software development practice (rather than in a particular DSL). In our LEGO analogy this means that the same semantic domain of plastic LEGO pieces (which model real-life details: gears, axles, pneumatic systems, etc.) is used to construct mechanisms (*i.e.,* DSLs) from various domains - from cars and trucks, to robotic arms. This is different to the definition of the dynamic semantics of a DSL (or mechanisms) in the form of requirements: then we target the semantic domains that are (substantially) different from each other. For example, for cars we would model concepts of speed and acceleration; and for robotic arms we would model concepts of gripping, spinning, and positioning.

On the one hand, this kind of semantic domain is low level and results in an increasing complexity of a semantic mapping that bridges the semantic gap between a DSL and the semantic domain. On the other hand, the commonality of the semantic domain makes it possible to reuse. When constructing LEGO mechanisms, one does not need to reinvent the wheel. There is a set of custom mechanical solutions (built of LEGO pieces), patterns, and principles that one can reuse for constructing linkages, transmissions, suspensions, pneumatic devices, etc. For example, a collection of such patterns is provided in the book by P.Kmieć [25]. Note that these mechanical solutions are not restricted to LEGO constructions, but are rather distilled and explained in terms of LEGO (Fig. 1c). The same idea can be applied to defining a DSL dynamic semantics via introduction of *reusable specification templates*.

Specification templates are introduced for the reuse of common (successful) design solutions, which appear when constructing (defining or prototyping) DSL implementations (solutions). In our work, we concentrate on reusable specification templates for defining the dynamic semantics of DSLs. However, further research can be performed (or has been performed - see related work in Sect. 6) to investigate the possibility to apply a similar approach for defining other aspects of DSLs: abstract syntax (metamodel), concrete syntax (grammar), or static semantics (type system).

Reusable specification templates realize the well-known approach of *generic programming* [32]. In generic programming, many concrete implementations of the same algorithm are captured in the form of a *generic algorithm* via abstracting from the concrete data types appearing in the algorithm. Such abstraction is expressed as requirements on the *parameters* of a generic algorithm. For example, Listing 1.1 (lines 1–4) depicts a generic algorithm implemented as a C++ function template for calculating the larger of two objects, `a` and `b`. The only parameter of this generic algorithm is the data type `Type`. From the source code, we can infer that the requirements on this parameter are as follows: this type should support copying of an object value and the operator

```
template <typename Type>
Type max(Type a, Type b) {
    return a > b ? a : b;
}
...
int z = max<int>(x, y);
```

Listing 1.1 Example of a C++ template and its invocation

greater-than (>). [2] Line 6 of Listing 1.1 demonstrates how this template can be (re)used: for this, the type parameter is specified (`int`) and the resulting specialized template is invoked as a usual C++ function (in the example, we calculate it for the variables `x` and `y`).

Similar to a source code template, a specification template is a piece of specification code parameterized for reuse via abstracting from the concrete data types and/or data. Unlike a template written in a programming language, a specification template is written in a formalism that is based on a solid mathematical theory. Therefore, we identify the following key features that characterize reusable specification templates.

– A specification template is a specification where some specification elements are considered as *template parameters*, and thus can be substituted by other elements of the same nature.
– Requirements on template parameters are specified explicitly as a part of the specification template.
– A specification template can be reused together with the results of its verification, such as: proof of the specification consistency and/or proof of some properties holding for this specification.
– After specializing a specification template, it can be invoked as a building block for constructing another (composite) specification. The verification results of the specification template hold after composing the template with other parts of the composite specification.

A framework that allows for composing a design solution using specification templates is based on a specification formalism, its tool support (verification and validation tools), and on a front-end that wraps the formal methods. The ingredients of such a framework should fulfill the following requirements.

– The specification formalism provides the possibility to parameterize specifications. Such parametrization allows for specialization of a specification into another one with possibility to reuse its verification results.
– The specification formalism provides the possibility to compose specifications in such a way, that the constituent specifications hold their verification results after being composed together.
– The front-end provides a language that allows for describing a design by configuring the parametrization and composition of reusable specification templates. Moreover, the front-end supports feedback from the formal methods tools to the language level.

[2] Note that in some programming languages it is possible to specify such requirements explicitly, for example, as an interface that should be realized by the type parameter.

The first two requirements are fulfilled by the Event-B formalism and the techniques of generic instantiation and shared event composition. The third requirement is partially contributed by this paper: the Constelle language.

2.3 Specification templates for composing DSL semantics

We aim to use specification templates as the semantic domain, and Constelle as the language for defining a semantic mapping (from a DSL to the semantic domain). In the classical approaches of denotational semantics or operational semantics [33], a semantic mapping is defined as a set of so-called semantic functions each of which gives a meaning to a separate construct (statement) of the language being defined. A meaning of a language construct determines how this construct changes the state of the program being executed. In other words, the semantics of each of the language constructs is defined separately and independently from other language constructs in terms of changes to the program state. This style of a semantic mapping can be characterized as a one-to-many relation (from DSL constructs to state changes).

However, our decision to define a DSL dynamic semantics as a solution (or implementation, as described in Sect. 2.1) leads to the following situation. In a solution, multiple DSL constructs (statements) can be implemented via the same specification template invocation, and one DSL construct can be implemented by multiple specification template invocations. In this case, the meaning of a DSL construct determines how multiple template invocations change the state of the program being executed. For example in our LEGO allegory (in Fig. 1a), two different 'DSL constructs': driving and turning - are implemented by the following common set of mechanical templates: the differential pattern (invoked twice) and the drive-train. The state of the car is changed via interaction of these template invocations. This style of a semantic mapping can be characterized as a many-to-many relation (from invocations of specification templates to state changes). As a consequence, when defining a semantic mapping we may face the problem of *scattering* and

```
1  void Point :: moveBy (int dx, int dy) {
2      x = x + dx; y = y + dy;
3      display.update();
4      log(MOVE_BY, this, dx, dy);
5  }
6
7  void Point :: setColor (int c) {
8      color = c;
9      display.update();
10     log(SET_COLOR, this, c);
11 }
```

Listing 1.2 Example of a C++ code with different aspects

tangling: the definition of a DSL construct is scattered over multiple invocations of specification templates, and an invocation of a specification template participates in the definitions of multiple DSL constructs (and, thus, tangles them).

The problem of scattering and tangling of software code are considered and managed by aspect-oriented programming (AOP) [22]. The AOP technique allows for modularization of *aspects* that crosscut a system's basic functionality. Examples of such aspects are synchronization, memory management, localization, logging, etc. For example, in the C++ code depicted in Listing 1.2 the basic functionality of updating the point's state (lines 2 and 8) is crosscut by the aspect of notifying the display about the new point's state (lines 3 and 9) and by the aspect of logging (lines 4 and 10). According to the AOP paradigm, these aspects can be extracted into separate (explicit) modules and then weaved into the basic functionality.

We use the AOP approach to express how a DSL dynamic semantics is composed of specification templates. We consider (specialized) specification templates as aspects and weave them together to form the functionality (behavior) of DSL constructs. In other words, the DSL semantic is defined as the weaving of aspects each of which is a specialized specification template. The major difference of our approach from the classical AOP is that we define the DSL semantic mapping only in terms of aspects, i.e., the basic functionality is a composition of aspects.

Moreover, comparing to the classical AOP realized in programming languages, aspects in our approach are formal specifications. Thus, the compatibility of aspects composed together can be analyzed using tools that support the specification formalism. An example of the compatibility of aspects is demonstrated in our LEGO example in Fig. 1a. Here two differential patterns are composed together into a 4 × 4 vehicle's drive-train. To ensure compatibility of these two template invocations, the front and rear differentials must be oriented in opposite directions so that the front and rear wheels rotate in the same direction.

3 The Constelle language

In this section, we develop a (meta)model of reusable specification templates and a metamodel of the Constelle language. Constelle allows for configuring the parametrization and the composition of specification templates when applying them for defining the dynamic semantics of a DSL. As a carrier formalism to express specification templates we have chosen Event-B, which is both precise and executable.

3.1 The Event-B formalism

Event-B is an evolution of the B method, both introduced by Abrial [1,2]. Event-B employs set theory and first-order logic for specifying software and/or hardware behavior. The Rodin platform [3] and its plug-ins provide various tool support for the formalism: one can create and edit Event-B specifications, verify their consistency using automatic or interactive provers, animate and model check Event-B specifications. Thus, Event-B and Rodin allow for formal modeling and development of correct-by-construction hardware and software systems.

An Event-B specification consists of *contexts* and *machines*. For example, Fig. 2 shows the Event-B contexts and machines of two specification templates that we use as examples in this paper: *Queue* and *Request*.

An Event-B context describes the static part of a system: *sets*, *constants*, and *axioms*. A machine uses ('sees') the context to specify the behavior of a system via a state-based formalism. *Variables* of the machine define the state space. *Events*, which change values of these variables, define transitions between the states. An event consists of *guards* ('where'-section) and *actions* ('then'-section), and can have *parameters* ('any'-section). An event can occur only when its guards are true, and as a result of the event its actions are executed. Parameters represent existentially quantified variables local to the event, *i.e.,* used in its guards and actions. The properties of the system are specified as *invariants*, which should hold for all reachable states. The properties can be verified via proving automatically generated *proof obligations* and can be debugged (examined) via animation (*i.e.,* execution) of the Event-B machine.

Figure 2c shows the Event-B specification of a well-known abstract data type, a queue. In this specification, the collection of elements is modeled as a partial function *queue* from natural numbers to *ElementType* (see invariant `inv1`), where *ElementType* is a set of all possible elements that can be stored in the queue (see the Event-B context in Fig. 2a). The possible operations on the collection are specified as the events *enqueue* and *dequeue*. In the *enqueue* event, a new pair *index* $\mapsto$ *element* is added to the collection (see the action `act2`) if the *index* is bigger than any other index used in the queue (see the guard `grd3`). In the *dequeue* event, a

Fig. 2 Event-B code of two specification templates: queue and request. **a** Event-B context for the Queue specification template, **b** Event-B context for the Request specification template, **c** Event-B machine for the Queue specification template, **d** Event-B machine for the Request specification template

```
CONTEXT  template_queue_context
SETS
      ElementType
END
```

(a)

```
CONTEXT  template_request_context
SETS
      ElementType
END
```

(b)

```
MACHINE  template_queue_machine
SEES  template_queue_context
VARIABLES
      queue
INVARIANTS
inv1 : queue ∈ ℕ ⇸ ElementType
EVENTS
Initialisation
  begin
    act1 : queue := ∅
  end
Event  enqueue ≙
    any element, index
    where
     grd1 : element ∈ ElementType
     grd2 : index ∈ ℕ
     grd3 : queue ≠ ∅ ⇒
                 (∀i·i ∈ dom(queue) ⇒ index > i)
    then
     act2 : queue := queue ∪ {index ↦ element}
  end
Event  dequeue ≙
    any element, index
    where
     grd4 : index ↦ element ∈ queue
     grd5 : ∀i·i ∈ dom(queue) ⇒ index ≤ i
    then
     act3 : queue := queue \ {index ↦ element}
  end
END
```

(c)

```
MACHINE  template_request_machine
SEES  template_request_context
VARIABLES
      request_body
INVARIANTS
inv1 : request_body ∈ ℙ(ElementType)
EVENTS
Initialisation
  begin
    act1 : request_body := ∅
  end
Event  request ≙
    any elements
    where
     grd1 : elements ∈ ℙ(ElementType)
     grd2 : request_body = ∅
    then
     act2 : request_body := elements
  end
Event  process ≙
    any element
    where
     grd3 : element ∈ request_body
    then
     act3 : request_body :=
                 request_body \ {element}
  end
END
```

(d)

pair *index* ↦ *element* is removed from the collection (see the action `act3`) if the *index* is smaller than any other index used in the queue (see the guard `grd5`). In this way, the First-In-First-Out (FIFO) property of the data structure is realized.

The attractive simplicity of Event-B is enhanced by the following techniques that support scalability and reuse of Event-B specifications [4].

– *Generic instantiation* allows for replacing sets and constants in an Event-B machine by new sets and constants that conform to the corresponding properties (axioms) of the former ones;
– *Shared event composition* allows for the composition of Event-B machines via their events (with no common variables allowed in the constituent machines).

We discuss these techniques in more detail in Sect. 4.

3.2 Metamodel of a specification template

To consider Event-B code as a specification template, we need (1) a mechanism to parameterize it into a generic template and (2) a mechanism to invoke this template when defining the dynamic semantics of a DSL. To keep the approach universal, we separate these mechanisms from a concrete carrier formalism (in our case, from Event-B). For this, we introduce the concept of *template interface* that supports the mechanisms of parametrization and invocation, independently from the concrete specification formalism. A *specification template* connects a template interface and the specification code that implements this interface.

In the metamodel depicted in Fig. 3, concepts related to the template interface are shown on the left; concepts of the formalism (Event-B) are shown on the right; and concepts of the specification template that connect these two parts are shown in the middle. Figure 4 shows the metamodel of an Event-B specification. This is a subset of the metamodel provided by the EMF framework for Event-B, one of Rodin plug-ins [48].

We distinguish two possible template interfaces: *structural interface* and *semantic interface*. For parametrization of Event-B code into a generic template, a structural interface defines a collection of template parameters that can be substituted by concrete data when specializing the template. As these parameters do not change their values during the execution of a composed system, we name them *static parameters*. In our LEGO example depicted in Fig. 1b, the static parameters are the sizes of the gears used in the differential: after the sizes are chosen and the corresponding gears are assembled into the mechanism, they are not changed any more (during driving).

The Queue specification template depicted in Fig. 2c is generic with respect to the type of elements stored in the queue. Thus, the corresponding structural interface includes one static parameter: `ElementType` (see Listing 1.3, structural interface `template_basic`). In the metamodel depicted in Fig. 3, we distinguish three possible static parameters: Constant, Type, and Relation. We note that the completeness of such a classification with respect to various specification formalisms and metamodeling languages requires further investigation. Therefore, we leave a possibility to extend our metamodel by adding new kinds of static parameters.

For invocation of the behavior specified in the template, a semantic interface provides a set of signatures: Operations with DynamicParameters in Fig. 3. Dynamic parameters allow for transferring data to and from the template behavior. In our LEGO example depicted in Fig. 1b, the red and green axles are dynamic parameters: they connect the differential with other parts of a system and transfer the rotation to (the green axle) and from (the red axles) the differential.

The behavior specified in the Queue specification template can be invoked via operations `enqueue` and `dequeue`. The data that are transferred into and from these operations are an element that should be added to or has been removed from the queue. Thus, the corresponding semantic interface consists of two operations: `enqueue` and `dequeue`, with `elements` as their dynamic parameters (see Listing 1.3, semantic interface `template_queue`).

Not all elements of the specification template should appear in the template interface. Some elements are encapsulated in order to hide details of the template design. For example, the *index* parameters of the events *enqueue* and *dequeue*, which are used for determining a position of the element being added/removed, are specific to the way the queue is modeled (a partial function from natural numbers to *ElementType*). Therefore, we encapsulate *index* and do not add it to the dynamic parameters of the semantic interface. The same applies to the *Initialisation* event of the Event-B specification.

```
structural interface template_basic {
    types ElementType
}

semantic interface template_queue uses template_basic {
    operation enqueue (element)
    operation dequeue (element)
}

semantic interface template_request uses template_basic {
    operation request (elements)
    operation process (element)
}
```

Listing 1.3 Structural and semantic interfaces of the specification templates Queue and Request

Finally, a SpecificationTemplate stores a collection of SpecificationElements, each of which references an EventBNamedCommentedElement (see the metamodel depicted in Fig. 3). An EventBNamedCommentedElement can be an Event-B variable, an event, a parameter, etc. (according to the Event-B metamodel in Fig. 4). All these specification elements constitute the template, and therefore, are explicitly referenced in it.

A specification element can be either public (PublicElement) or private (PrivateElement), see Fig. 3. A public element links an element of the template interface (InterfaceElement) with an element of the Event-B code that implements this interface element. When the specification template is applied, this Event-B element is substituted by another element of the same kind according to the specialization and invocation of the interface element. For example, the `enqueue` operation of the semantic interface `template_queue` (Listing 1.3) is linked to the event `enqueue` of the Event-B machine *template_queue_machine* (Fig. 2c); and the `ElementType` type of the structural interface `template_basic` is linked to the set *ElementType* of the Event-B context *template_queue_context*. The elements of the Event-B specification which do not appear in the template interface, are referenced through PrivateElements. For example, the *index* parameters of the events *enqueue* and *dequeue* are not included in the template interface, and thus, are referenced through the corresponding objects of the class PrivateElement.

As an Event-B specification is organized as a context for the static part and a machine for the dynamic part, it is natural to split a specification template into a StructuralTemplate linking a structural interface and an Event-B context, and a SemanticTemplate linking a semantic interface and an Event-B machine. Theoretically, the right part of the metamodel

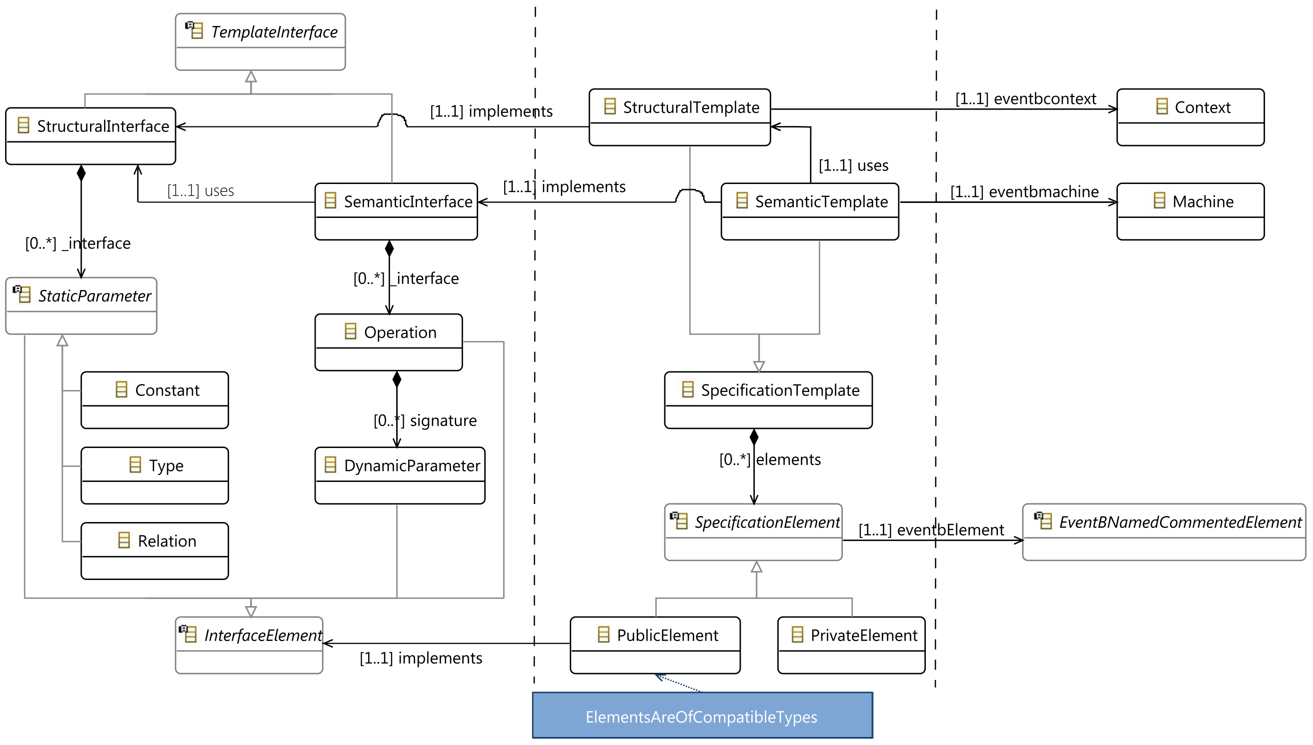


Fig. 3 Metamodel of a specification template

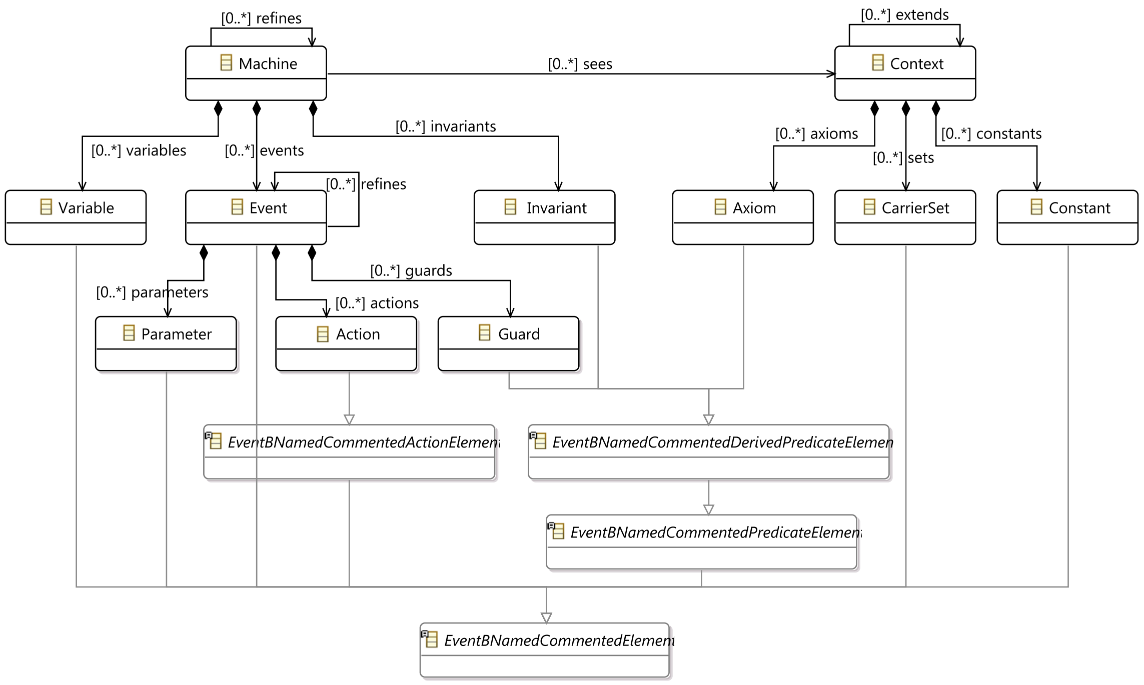


Fig. 4 Metamodel of an Event-B specification

depicted in Fig. 3 can be replaced by concepts of another specification formalism, with a necessary adjustment of the middle part.

3.3 Design of the Constelle language

To use specification templates for defining the dynamic semantics of a DSL, we develop the Constelle language.

Fig. 5 An industrial robot and a DSL program that controls it. **a** Robotic arm, **b** a DSL program

(a)

```
1  task Demo {
2    arm action (turnright , 60);
3    arm action (movedown, 10);
4    hand action (grab),
5    arm action (moveup, 10);
6    hand action (rotateleft , 180);
7    arm action (turnleft , 60);
8    arm action (movedown, 10);
9    hand action (release);
10 }
```

(b)

Constelle applies the ideas of generic programming and aspect-oriented programming described in Sect. 2.3. Namely, in Constelle the dynamic semantics of a DSL is defined as a composition of aspects, each of which is a specification template specialized by substituting its (static) parameters with the DSL constructs. The semantics of Constelle maps such a definition to the corresponding (Event-B) specification of the dynamic semantics of the DSL by substituting and composing the specification templates. Such substitution and composition raise certain proof obligations in the resulting specification. We discuss the Constelle-to-Event-B mapping and how the corresponding proof obligations can be identified and discharged in Sect. 4.

To realize our approach, we need to have a *library of specification templates*, which we can use in a definition. This library can exist beforehand or can be created and extended during the process of designing the DSL. The purpose of such a library is to collect and store the knowledge and expertise of designing and developing a DSL (for a concrete domain or for a general broad usage).

In this section, we explain and design the Constelle language through the following example: we consider a DSL for controlling an industrial robot and define (a subset of) its dynamic semantics in Constelle using a library of two specification templates, Queue and Request. The industrial robot and an example of a DSL program for controlling it are depicted in Fig. 5. The specification templates Queue and Request were introduced in Sect. 3.1.

The industrial robot consists of two major mechanical parts: a hand, responsible for manipulating objects, and an arm, responsible for moving the hand to a certain position. Our example DSL allows for programming tasks for such a robot using a set of actions that can be performed by these parts, such as: actions `turn left`, `turn right`, `move up`, `move down` performed by the arm; and actions `grab`, `release`, `rotate left`, `rotate right` performed by the hand. The DSL program in Listing 5b specifies the task `Demo` that should be executed by the robotic arm. While some actions in a task should be performed in a certain order, some other actions can be performed in parallel, as the arm and the hand can operate independently. For example, in lines 5–7 in Listing 5b, the actions `move up` and `turn left` of the arm can be performed in parallel with the action `rotate left` of the hand. However, the action `release` of the hand should be performed only after the action `move down` of the arm.

The dynamic semantics of the example DSL realizes the parallel execution of mutually independent actions of the robot parts, and the sequential execution of mutually dependent actions. In Constelle we define these two types of execution in two separate *semantic modules*. First we define the parallel (independent) execution of actions in the semantic module `Robotic Arm Parallel` using the specification templates Queue and Request. Then we define the sequential execution of actions in the semantic module `Robotic Arm Sequential` using the module `Robotic Arm Parallel` and other specification templates.

In other words, in Constelle the dynamic semantics of a DSL is split into separate semantic modules, each of which encapsulates a behavioral aspect and/or certain design decision(s) and hides it from the rest of the semantics definition. Each of these semantic modules is split into smaller semantic modules – and so on till we arrive at the library of specification templates, which have the corresponding (Event-B) implementations. Thus, a definition is structured as a *directed acyclic graph (DAG)* of semantic modules, where the edges represent the relation ‘composed of’ and the graph sinks represent specification templates from the library. Such a design allows for a scalable, modular, and formalism-independent definition of the dynamic semantics of a DSL.

Table 1 introduces the semantic module `Robotic Arm Parallel`, and shows how this semantic module is composed of the specification templates Queue and Request. The semantic interface of `Robotic Arm Parallel` is shown in the leftmost column of the table. The other columns show invocations of the specification templates: `driver1` and `driver2` both invoke the Queue template, and `distributor` invokes the Request template. The rows of the table show different elements of the corresponding

Table 1 Semantic module `Robotic Arm Parallel`

Robotic Arm Parallel	driver1 : Queue	driver2 : Queue	distributor : Request
taskStm			request
• task			• elements
armActionStm	enqueue		process
• action	• element		• element
handActionStm		enqueue	process
• action		• element	• element
executeArm	dequeue		
• action	• element		
executeHand		dequeue	
• action		• element	
Actions			ElementType
ArmActions	ElementType		
HandActions		ElementType	

semantic interfaces: operations (non-shaded rows) and their parameters (shaded rows). The intersections of the rows and the columns show the mapping of the elements of the semantic module to the elements of the constituent template invocations. The bottom part of the table shows the mapping of the structural interfaces used in these semantic interfaces. Below we explain this Constelle definition in detail.

As we stated in Sect. 1, the main idea behind our approach is to define the dynamic semantics of a DSL as a two-steps semantic mapping: first, from the DSL constructs to an intermediate semantic domain; second, from the intermediate semantic domain to the target execution platform. The choice of such an intermediate semantic domain is not arbitrary: it is formed by the typical (design) solutions that are used for handling the target execution platform (in other words, by concepts of the horizontal domain of the DSL, as discussed in Sect. 1). For example, a robotic arm is typically controlled via the drivers of its parts. In our semantics definition, we represent such drivers as queues to emphasize that the drivers have buffers for storing actions that should be executed. The third aspect of the `Robotic Arm Parallel` is a `distributor`, responsible for assigning actions to the drivers.

A Constelle table, such as Table 1, represents a mapping from the DSL (vertical) concepts, depicted in the leftmost column, to the intermediate semantic domain (*i.e.,* the DSL horizontal concepts), depicted in the other columns. For the example DSL we distinguish the following DSL concepts: the task statement, two types of action statements, and two types of action executions (by the arm and by the hand). In other words, we separate the concept of an action statement in a DSL program from the concept of the resulting execution of the action by the robotic arm. These (vertical) concepts appear in the semantic interface of `Robotic Arm Parallel` (leftmost column in Table 1) as the operations `taskStm`, `armActionStm`, `handActionStm`, `executeArm`, and `executeHand`.

The operation `taskStm` is a starting point of an execution and is responsible for initializing a task. We define this operation as the `request` operation of the Request template (the right column in Table 1). The `elements` parameter of the `request` operation corresponds to the task that is requested for the execution (parameter `task` in the left column). According to the Event-B specification of the Request template depicted in Fig. 2d, this means that the task is saved in the internal variable *request_body*; and a new task can be requested only after the current task is processed (see `grd2` in Fig. 2d).

After initializing a task, we process it action by action (or statement by statement) using the `process` operation of the Request template. Each action is assigned for execution to the hand or to the arm – depending on the type of the action. Thus, the operations `armActionStm` and `handActionStm` are composed of the `enqueue` operation of the corresponding `driver` and the `process` operation of the `distributor`. Moreover, the action that is processed in the `distributor` is the same action that is queued in a `driver`. This is depicted by putting the parameters `element` of `enqueue` and `process` in the same row as the parameter `action` of `armActionStm` or `handActionStm`.

An execution of an action corresponds to the dequeuing of this action. Therefore, the operations `executeArm` and `executeHand` are defined as the operations `dequeue` of the template invocations `driver1` and `driver2` correspondingly.

Table 2 Semantic module `Robotic Arm Sequential`

Robotic Arm Sequential	core : Robotic Arm Parallel	sequential : Partial Order
taskStm	taskStm	NewPartialOrder
• task	• task	• poset
• order		• order
armActionStm	armActionStm	GetMaximalElement
• action	• action	• maximal
handActionStm	handActionStm	GetMaximalElement
• action	• action	• maximal
executeArm	executeArm	RemoveElement
• action	• action	• element
executeHand	executeHand	RemoveElement
• action	• action	• element
Actions	Actions	PosetElement
ArmActions	ArmActions	
HandActions	HandActions	

Finally, we specialize static parameters of the invoked specification templates. For this, we use the following constructs of the example DSL. The set `Actions` contains all the predefined actions of the DSL. As actions can be performed either by the arm or by the hand, the set `Actions` is partitioned by the sets `ArmActions` and `HandActions`:

$$\mathtt{Actions} = \mathtt{ArmActions} \cup \mathtt{HandActions},\quad \mathtt{ArmActions} \cap \mathtt{HandActions} = \varnothing$$

The substitution of the static parameters is depicted in the bottom rows of Table 1. Namely, the `Actions` type substitutes the `ElementType` of the Request template (the right column in Table 1). This means that the *request_body* of the Request specification (depicted in Fig. 2d) becomes a subset of `Actions`. Moreover, the `task` parameter of the `taskStm` operation is a subset of `Actions` too:

$$\mathtt{task} \in \mathbb{P}(\mathtt{Actions})$$

The `ArmActions` type substitutes the `ElementType` of the `driver1`. The `HandActions` type substitutes the `ElementType` of the `driver2`. These mean that only the actions of the corresponding type are stored in the queues: `ArmActions` are stored in `driver1 : Queue`, and `HandActions` are stored in `driver2 : Queue` (see guard `grd1` in Fig. 2c).

In the resulting semantics of the example DSL, actions of the arm and the hand are executed independently from each other in order of arrival to a corresponding queue. Moreover, according to the way actions are processed, the order of actions within a task does not matter. However, the order of requesting tasks does matter, as a new task cannot be initialized until all the actions of the current task are sent to the queues.[3]

Table 2 defines the semantic module `Robotic Arm Sequential`. In this module, we use the same names of operations and parameters (the leftmost column in Table 2) as we used in the semantic module `Robotic Arm Parallel`. The semantic module `Robotic Arm Sequential` is composed of the semantic module `Robotic Arm Parallel` (the second column of the table) and the template Partial Order (the rightmost column). The latter imposes a partial order on the actions forming a task.[4] This aspect restricts processing of actions to the maximal elements of the order and removes the executed actions from the order. For the sake of brevity, we do not discuss the details of this table here.

3.4 Metamodel of the Constelle language

The metamodel that allows for such a Constelle definition as described above is presented in Fig. 6: the part depicted on the left duplicates (the subset of) the metamodel of a template interface from Fig. 3, and the part depicted on the right shows concepts related to the definition of DSL semantic modules using specification templates (i.e., their semantic interfaces).

To implement both the specialization and the invocation of specification templates, we use substitution of the interface elements of constituent semantic interfaces with the interface elements of the composite semantic interface. This

[3] The complete Event-B specification of the semantic module `Robotic Arm Parallel` is given in 'Appendix 2'.

[4] The Event-B specification of the partial order template is given in 'Appendix 1.'

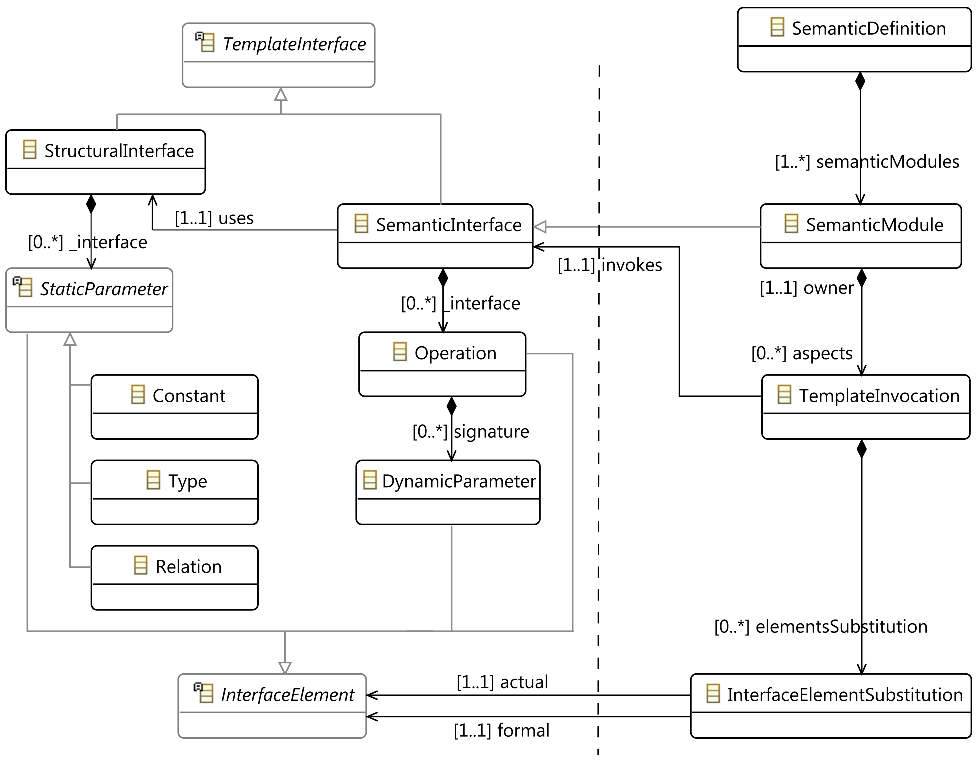


Fig. 6 Metamodel of the Constelle language

means that in a Constelle table, an interface element from the leftmost column substitutes all interface elements situated in the same row in other columns. This mechanism is realized through InterfaceElementSubstitutions (Fig. 6, on the bottom), each of which substitutes a formal interface element of the invoked (constituent) semantic interface with an actual interface element of the semantic module (*i.e.,* composite semantic interface).

This applies to all interface elements introduced earlier: static parameters (that are used in the semantic interfaces), operations, and dynamic parameters. By substituting static parameters we specialize the templates with the DSL constructs and synchronize them with respect to the data types. By substituting operations we invoke these templates and weave them together in the aspect-oriented style (as it was discussed in Sect. 2.3). By substituting dynamic parameters we realize transferring of data between the templates. As mentioned earlier, all these substitutions raise certain proof obligations in the resulting formal specification. In Sect. 5.2, we discuss how such proof obligations can be identified and discharged.

4 Semantics of the Constelle language

In the Constelle language, the dynamic semantics of a DSL is defined using specification templates. Such templates are collected in a library, which facilitates reusing design solutions. As a carrier of such design solutions, *i.e.,* as an implementation formalism for our specification templates, we use Event-B. This means that Constelle realizes a semantic mapping of the DSL to the semantic domain of Event-B. In Fig. 7 this process is represented as the leftmost T-diagram (the notation known from the compiler theory [5]): the semantic mapping of the DSL to Event-B is realized in Constelle.

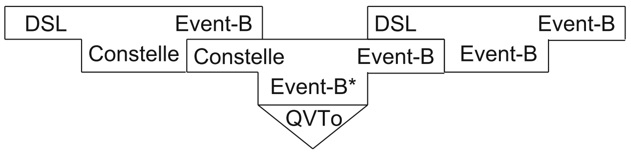


Fig. 7 T-diagrams of the Constelle semantics definition

In Constelle, the dynamic semantics of a DSL is defined as a composition of specialized specification templates. The engine of the Constelle language realizes code substitution in the templates and composes the resulting specifications into the whole Event-B specification of the DSL dynamic semantics. The semantics of Constelle defines such a translation from a composition of specialized specification templates to the corresponding Event-B code. In Fig. 7, the definition of the Constelle semantics is represented as the middle T-diagram: the semantic mapping of Constelle to Event-B.

To benefit from a semantics definition in practice and to implement the use cases discussed in Sect. 2.1, we aim to have the semantic mapping from Constelle to Event-B both precise and executable. Precision of the semantic mapping from Constelle to Event-B is achieved through the employment of two Event-B techniques: generic instantiation and shared event composition–introduced briefly in Sect. 3.1 and depicted as Event-B* in Fig. 7. These techniques are theoretically solid [4] and allow for the reuse of verification results of the specification templates in a specification of the DSL dynamic semantics generated from its Constelle definition. In this way, the requirements for a specification formalism listed in Sect. 2.2 are satisfied.

Executability of the semantic mapping from Constelle to Event-B (according to our definition of executability given in Sect. 2.1) is achieved through its implementation using the MDE technique of model-to-model transformation. Namely, we implement the Event-B* techniques in a Constelle-to-Event-B transformation using the QVTo (Query/View/Transformation-Operational) model transformation language [35]. In Fig. 7 this implementation is represented as the triangle on the bottom of the middle T-diagram.

The semantic mapping from a DSL to Event-B realized in Constelle (leftmost T-diagram in Fig. 7) combined with the semantic mapping from Constelle to Event-B results in a semantic mapping from the DSL to Event-B realized in Event-B. This mapping is represented as the rightmost T-diagram in Fig. 7. In the resulting semantics definition, both the semantic mapping and semantic domain are precise and executable, thus fulfilling our requirement as stated in Sect. 2.1.

In this section, we define the semantic mapping from Constelle (as defined in its metamodel in Fig. 6) to Event-B in terms of the Event-B* techniques. The definition of the Constelle semantic mapping follows its actual implementation, the Constelle-to-Event-B transformation coded in QVTo. In what follows we give an informal description of the Constelle semantic mapping using set theory and functions, aligned with the concepts and notation of the QVTo language. Namely, we define the Constelle-to-Event-B semantic mapping as a set of functions, capturing QVTo model transformations, and defined in terms of the metamodels introduced above: the Event-B metamodel (Fig. 4), the metamodel of a specification template (Fig. 3), and the Constelle metamodel (Fig. 6). Each class in these metamodels is viewed as a set of objects that instantiate this class, and as such is used in the definitions of the QVTo functions. The approach of describing and designing QVTo model transformations using set theory and functions is elaborated in [53]. Below we explain this approach as we apply it to the Constelle-to-Event-B transformation. Note that as we use set theory and functions, we do not discuss some trivial details, such as formal applicability or constraints that should be fulfilled. This is done for the sake of brevity. However, the actual Constelle-to-Event-B transformation performs all necessary checks and identifies the required proof obligations.

4.1 Model transformations from Constelle to Event-B

The model transformation Constelle-to-Event-B transforms (maps, translates) a Constelle definition of the DSL dynamic semantics to a corresponding Event-B specification of this DSL dynamic semantics. For this, the transformation consumes a Constelle model (instance of the Constelle metamodel) and a library of specification templates invoked in this Constelle model. As in Constelle the dynamic semantics of a DSL is defined through a DAG of semantic modules (see Sect. 3.3), the transformation produces an Event-B specification that consists of multiple Event-B machines: an Event-B machine for each semantic module of the definition. Each of these resulting Event-B machines is wrapped into a specification template. This is done for the sake of uniformity (of semantic modules and semantic templates) and for the possibility to construct new specification templates from existing ones using Constelle.

Consequently, we describe the Constelle-to-Event-B transformation by the following function:[5]

$$\begin{aligned}&\mathsf{Constelle\text{-}to\text{-}Event\text{-}B}:\\ &\mathbb{P}(\mathsf{SemanticTemplate}) \rightarrow \mathsf{SemanticDefinition}\\ &\qquad\qquad\qquad\qquad \rightarrow \mathbb{P}(\mathsf{SemanticTemplate})\end{aligned} \tag{1}$$

Here the transformation applies a library (a set) of SemanticTemplates to a SemanticDefinition of a DSL; and generates as an output a collection of SemanticTemplates that implement all semantic modules of the input semantic definition.

As described in Sect. 3.4, each semantic module of a Constelle model substitutes static parameters of the invoked specification templates with the static parameters representing the DSL constructs–such as the types `Actions`, `ArmActions`, and `HandActions` in the example DSL described in Sect. 3.3. For the sake of simplicity, we assume that all semantic modules of a Constelle model use the same structural interface, which introduces all necessary DSL constructs (types, relations, and constants). To ensure that the resulting Event-B machines specify the DSL semantics in terms of these concepts, we assume that this structural interface is implemented in an Event-B context wrapped in the corresponding structural template. We treat such a structural template as a global constant (or environment) of the Constelle-to-Event-B transformation and, therefore, do not include it in the function definitions discussed in this section.

Transformation (1) can be implemented using the following (sub-)transformation that considers a single SemanticModule of a semantic definition and results in a single

[5] All concepts used in the formulas of this section correspond to the similarly named classes depicted in Figs. 4, 3 and 6 (such as SemanticTemplate and SemanticDefinition).

SemanticTemplate correspondingly:

$$\begin{aligned}&\mathsf{ExpandDefinition}:\\ &\mathbb{P}(\mathsf{SemanticTemplate}) \rightarrow \mathsf{SemanticModule}\\ &\qquad\qquad \rightarrow \mathsf{SemanticTemplate}\end{aligned} \quad (2)$$

To transform all semantic modules of a semantic definition using their dependencies on (invocations of) each other, we apply transformation (2) to the nodes of the DAG of semantic modules starting from the sinks toward the sources. In this way we ensure that when a semantic module is to be transformed, all its aspects (invocations of SemanticInterfaces, see Fig. 6) have the corresponding implementations in the form of semantic templates.

The transformation ExpandDefinition realizes the two mechanisms of the specification templates approach described in Sect. 3.2: substitution of parameters of a generic template, and invocation (composition) of specialized templates. These mechanisms are implemented as two (separate) steps: Substitute and Compose–which are connected into a chain of model transformations:

$$\begin{aligned}&\mathsf{ExpandDefinition}(\mathrm{lib})(\mathrm{module}) = \mathsf{Compose}(\mathrm{module})\\ &\quad\Big(\{\mathsf{Substitute}(a, t) \mid a \in \mathrm{module}.\mathsf{aspects} \wedge t \in \mathrm{lib}\\ &\wedge t.\mathsf{implements} = a.\mathsf{invokes}\}\Big)\end{aligned} \quad (3)$$

We discuss the details of the function application of Substitute and Compose [*i.e.,* arguments of the mappings appearing in formula (3)] and their signatures in the following sections.

4.2 Substitution

Substitution of parameters in an invoked semantic template is the first step of the transformation ExpandDefinition. It takes a SemanticTemplate from the library and performs its TemplateInvocation by transforming it into a new SemanticTemplate, which uses (*i.e.,* implements) the semantic interface of the semantic module (being composed) instead of the semantic interface of the template (being invoked). The Substitute transformation can be described by the following function:

$$\begin{aligned}&\mathsf{Substitute}: \mathsf{TemplateInvocation} \rightarrow \mathsf{SemanticTemplate}\\ &\quad \rightarrow \mathsf{SemanticTemplate}\end{aligned} \quad (4)$$

Correspondingly, the transformation ExpandDefinition (3) applies Substitute (4) to all aspects of its input semantic module: Substitute(a, t) for $a \in$ module.aspects [see the second line of formula (3)]. For this, ExpandDefinition finds a corresponding template (t) in the library, *i.e.,* the template that implements the semantic interface invoked in the aspect (a): $t \in \mathrm{lib} \wedge t.\mathsf{implements} = a.\mathsf{invokes}$. Note that in our formulas we use the dot notation (common in object-oriented languages) to represent an associated object: for example, a.invokes represents an object of the class SemanticInterface (according to the metamodel in Fig. 6).

The Substitute transformation realizes three objectives:

- Specialization of the template parameters with the DSL constructs using InterfaceElementSubstitutions of StaticParameters (as described in Sect. 3.4);
- Preparation for further composition of the template by assigning new identifiers to its Operations and DynamicParameters–according to InterfaceElementSubstitutions of these elements with elements of the target (composite) specification;
- Encapsulation of the specification elements that do not implement any interface elements (PrivateElements, as described in Sect. 3.2) through extending their identifiers with a proper namespace identifier (to avoid possible name conflict in the target specification)–using as the namespace the aspect that invokes this template.

All the listed objectives of substitution are realized by renaming Event-B elements in the specification code using the generic instantiation technique. For this, the transformation Substitute (4) performs three steps: an auxiliary transformation that configures renaming; generic instantiation, and an auxiliary transformation that wraps the resulting Event-B machine into the corresponding semantic template. We discuss these steps in the next three sub-subsections.

Configuration of renaming is generated in a transformation ComposeRenaming. This transformation applies a set of InterfaceElementSubstitutions and a namespace extension (String) to a set of SpecificationElements (of a semantic template). The result of the transformation, EventBNamedCommentedElement $\nrightarrow$ String, is a partial function of renamings for the Event-B elements of the specification. The transformation is described by the following function:

$$\begin{aligned}&\mathsf{ComposeRenaming}:\\ &\quad\mathbb{P}(\mathsf{InterfaceElementSubstitution}) \times \mathsf{String}\\ &\quad\rightarrow \mathbb{P}(\mathsf{SpecificationElement})\\ &\quad\rightarrow (\mathsf{EventBNamedCommentedElement} \nrightarrow \mathsf{String})\end{aligned} \quad (5)$$

The transformation ComposeRenaming realizes the three substitution objectives listed above. For this, the transformation is implemented in the following way:

$$\begin{aligned}&\mathsf{ComposeRenaming}(\mathrm{substset}, \mathrm{nmspc})(\mathrm{specset}) =\\ &\{s.\mathsf{eventbElement} \mapsto x.\mathsf{actual.name} \mid\\ &\ s \in \mathrm{specset} \cap \mathsf{PublicElement} \wedge x \in \mathrm{substset} \wedge x.\mathsf{formal}\\ &= s.\mathsf{implements}\} \cup\\ &\{s.\mathsf{eventbElement} \mapsto \mathrm{nmspc} + s.\mathsf{eventbElement.name} \mid\\ &\ s \in \mathrm{specset} \cap \mathsf{PrivateElement}\}\end{aligned} \quad (6)$$

In other words, the ComposeRenaming transformation considers separately public and private specification elements. For the former, it finds an interface element substitution (x) that substitutes the interface element implemented by this specification element (x.formal $=$ s.implements); and uses the name of the target interface element for renaming (x.actual.name). For the private specification elements, the transformation simply extends their names with the namespace: s.eventbElement $\mapsto$ nmspc $+$ s.eventbElement.name.

For example, when we apply this function to the aspect `driver1` of the semantic module `Robotic Arm Parallel` defined in Table 1, we use interface element substitutions depicted in the table lines to substitute (*i.e.,* rename) elements from the second column with elements from the leftmost column. As a namespace extension we use the aspect name, *i.e.,* 'driver1.' And the input set of specification elements is taken from the specification template Queue. As a result we get the renaming function depicted in Fig. 8a.

Generic instantiation is introduced by Abrial et al. [4] and is developed in detail by Silva and Butler [45]. Generic instantiation realizes reuse of an Event-B specification by considering it as a generic model and instantiating it into a more specific model. For this, the context of an Event-B specification $C(s, c)$ is considered as its parametrization. The sets s and the constants c introduced in the context play the role of parameters of the generic model; and the axioms $A(s, c)$ capture their properties, *i.e.,* requirements on the parameters. An instantiation of such a specification uses an Event-B context $D(ds, dc)$ with more specific sets ds and constants dc, featuring more specific properties captured in the axioms $DA(ds, dc)$. An instantiation of the generic Event-B machine is performed by replacing its parameters s and c with the instance elements ds and dc. Moreover, the variables, events, and parameters of the generic machine can be renamed in the instantiated machine.

In practice, generic instantiation is implemented by syntactically replacing generic elements with instance elements (sets, constants, variables, events, and their parameters), or renaming generic elements into instance elements. For example, Fig. 8c shows an Event-B machine instantiated from the Queue template (depicted in Fig. 2c) using the renaming configuration depicted in Fig. 8a.

According to [45], the resulting instantiated machine is *correct by construction* if the following conditions hold.

- The requirements on the parameters of the generic specifications hold for its instantiation. This means that the properties $A(ds, dc)$ of the generic specification can be derived from the properties $DA(ds, dc)$ of the specific (instantiated) specification.
- Each set (in the generic specification) must be replaced by a set or by a valid type expression (in the instantiated specification), and each constant must be replaced by a constant.

In our approach we ensure these requirement by providing a proper implementation of the structural interface used in the Constelle definition (an Event-B context wrapped in a structural template, as discussed earlier in this section) and stating generic properties $A(ds, dc)$ as theorems that need to be proved. An example of such Event-B context for our robotic arm DSL is depicted in Fig. 8b. As the Queue template does not contain axioms (see Fig. 2a), we do not need to prove any generic properties.

To apply (call) generic instantiation in our definition of the semantics of Constelle, we represent it as the following function:

$$\begin{aligned}&\mathsf{GenericInstantiation}:\\ &\quad \mathsf{Machine} \rightarrow (\mathsf{EventBNamedCommentedElement}\\ &\qquad \nrightarrow \mathsf{String}) \rightarrow \mathsf{Machine}\end{aligned} \tag{7}$$

Here a Machine is instantiated using a partial function of renaming Event-B elements (EventBNamedCommentedElement $\nrightarrow$ String), and as a result a new (instantiated) Machine is generated.

The substitution transformation (4) is defined as a function composition of the transformations ComposeRenaming (5) and GenericInstantiation (7), and an auxiliary transformation ReconstructInstantiatedTemplate (discussed further):

$$\begin{aligned}&\mathsf{Substitute}(\mathrm{inv})(\mathrm{tmpl}) =\\ &\quad \mathsf{ComposeRenaming}(\mathrm{inv.elementsSubstitution},\ \mathrm{inv.name})\\ &\qquad\qquad\qquad (\mathrm{tmpl.elements} \cup \mathrm{tmpl.uses.elements})\\ &\ \circ\ \mathsf{GenericInstantiation}(\mathrm{tmpl.eventbmachine})\\ &\ \circ\ \mathsf{ReconstructInstantiatedTemplate}(\mathrm{substset},\ \mathrm{tmpl})\end{aligned} \tag{8}$$

Here the renaming configuration is generated for elements of both semantic template (tmpl.elements) and of its static template (tmpl.uses.elements). The resulting renaming is applied directly ($\circ$) to instantiate the Event-B machine of the template (tmpl.eventbmachine). The output of GenericInstantiation is translated into the output of Substitute using the auxiliary transformation ReconstructInstantiatedTemplate described by the following function:

$$\begin{aligned}&\mathsf{ReconstructInstantiatedTemplate}:\\ &\mathbb{P}(\mathsf{InterfaceElementSubstitution})\\ &\times \mathsf{SemanticTemplate}\\ &\rightarrow \mathsf{Machine} \rightarrow \mathsf{SemanticTemplate}\end{aligned} \tag{9}$$

The transformation ReconstructInstantiatedTemplate (9) generates a SemanticTemplate that wraps the newly

Fig. 8 Substitution of the Queue template for the `arm` aspect. **a** Renaming function, **b** Event-B context that implements structural interface of the example, **c** fragment of the instantiated Event-B machine

```
ElementType ↦ 'ArmActions',
queue ↦ 'driver1_queue',
enqueue ↦ 'armActionStm',
      element ↦ 'action',
      index ↦ 'driver1_index',
dequeue ↦ 'executeArm',
      element ↦ 'action',
      index ↦ 'driver1_index'
```

(a)

```
CONTEXT  dsl_context
SETS
    Actions, ArmActions, HandActions
AXIOMS
axm1 : partition(Actions,
            ArmActions, HandActions)
END
```

(b)

```
MACHINE  driver1_queue_machine
SEES  dsl_context
VARIABLES
       driver1_queue
INVARIANTS
inv1 : driver1_queue ∈ ℕ ⇸ ArmActions
EVENTS
Initialisation
  begin
     act1 : driver1_queue := ∅
  end
Event  armActionStm ≙
    any  action, driver1_index
    where
     grd1 : action ∈ ArmActions
     grd2 : driver1_index ∈ ℕ
     grd3 : driver1_queue ≠ ∅ ⇒
   (∀i·i ∈ dom(driver1_queue) ⇒ driver1_index > i)
    then
     act2 : driver1_queue :=
         driver1_queue ∪ {driver1_index ↦ action}
  end
Event  executeArm ≙
       ...
```

(c)

generated (instantiated) Event-B Machine. For this, the transformation takes the original template (tmpl) and traces back its specification elements to the Event-B elements of the machine through the set of interface element substitutions (substset) that have been applied to the original template.

4.3 Composition

Composition of the substituted semantic templates is the second step of the transformation ExpandDefinition. It is described by the following function:

$$\mathsf{Compose} : \mathsf{SemanticInterface} \\ \rightarrow \mathbb{P}(\mathsf{SemanticTemplate}) \rightarrow \mathsf{SemanticTemplate} \qquad (10)$$

This transformation composes a set of SemanticTemplates, each of which implements the same (shared) SemanticInterface, into a single SemanticTemplate. Correspondingly, the transformation ExpandDefinition applies Compose to its input semantic module (as it is a semantic interface itself) and to the semantic templates resulting from the substitution of the module aspects (see Formula (2)).

An Event-B machine of the resulting semantic template is composed of the Event-B machines of the input semantic templates using shared event composition. We present shared event composition and how it is used for composing specification (semantic) templates in the next two sub-subsections.

Shared event (de)composition supports modularity of an Event-B specification by structuring it as a collection of independent sub-components interacting with each other [46]. Each sub-component is specified in a separate *constituent Event-B machine*, which does not share its state (*i.e.,* its variables) with the constituent machines of other sub-components. The sub-components interact with each other by sharing (synchronizing) events of the corresponding constituent machines. The synchronization of events is defined in a composition configuration. The synchronized events can exchange data via shared parameters. This mechanism is similar to the exchange of messages between synchronized input and output channels in Communicating Sequential Processes (CSP) [18].

Figure 9 shows an example of a machine composed for the semantic module `Robotic Arm Parallel` according to its definition in Table 1. This machine is composed of three machines: two instances of *template_queue_machine* (introduced in Fig. 2c)–for the aspects `driver1` and `driver2`,–and one instance of *template_request_machine* (introduced in Fig. 2d)–for the aspect `distributor`.

The resulting composite machine is constructed as follows. The composite machine sees the contexts of all constituent specifications. In the example, all constituent machines see the same context, *dsl_context* that implements the structural interface of the Constelle definition. The list of variables of the composite machine is a concatenation of (not

Fig. 9 Fragment of the composite Event-B machine

```
MACHINE  robotic_arm_parallel
SEES  dsl_context
VARIABLES
      driver1_queue, driver2_queue, distributor_request_body
INVARIANTS
driver1_inv1 : driver1_queue ∈ ℕ ⇸ ArmActions
driver2_inv1 : driver2_queue ∈ ℕ ⇸ HandActions
distributor_inv1 : distributor_request_body ∈ ℙ(Actions)
EVENTS
      ...
Event  taskStm ≙
    any  task
    where
     distributor_grd1 : task ∈ ℙ(Actions)
     distributor_grd2 : distributor_request_body = ∅
    then
     distributor_act2 : distributor_request_body := task
  end
Event  armActionStm ≙
    any  action, driver1_index
    where
     driver1_grd1 : action ∈ ArmActions
     driver1_grd2 : driver1_index ∈ ℕ
     driver1_grd3 : driver1_queue ≠ ∅ ⇒ (∀i·i ∈ dom(driver1_queue) ⇒ driver1_index > i)
     distributor_grd3 : action ∈ distributor_request_body
    then
     driver1_act2 : driver1_queue := driver1_queue ∪ {driver1_index ↦ action}
     distributor_act3 : distributor_request_body := distributor_request_body \ {action}
  end
      ...
```

overlapping) lists of variables of the constituent machines: *arm_queue*, *hand_queue*, and *task_request_body*. Possible overlapping of variables, *i.e.,* name conflicts, are avoided via namespace extension performed during generic instantiation of the constituent machines (see Sect. 4.2).

The invariants of the composite machine are a conjunction of the invariants of the constituent machines. The not synchronized (*i.e.,* not interacting) events are copied from the constituent machines without modifications: for example, *taskStm* in Fig. 9. Each set of synchronized events is composed into one composite event (such as *armActionStm* in Fig. 9) by conjuncting guards of the constituent events and concatenating actions of the constituent events. The parameters of a composite event are a union of the parameters of the constituent events with respect to shared (overlapping) parameters, such as *action* in *armActionStm*.

According to [46], the resulting composed machine is *correct by construction*. In other words, the results of proving consistency, feasibility, and well-definedness of the constituent machines directly extend to the composite machine. However, this important theoretical outcome does not exclude possible incompatibility of constituent machines. For example, a shared parameter might have different (and incomparable) types in different constituent machines. Such a situation is possible, as constituent machines are generated as instantiations of other machines according to a Constelle definition. Thus, the corresponding (compatibility) checks should be performed for a Constelle definition. In our approach we delegate such checking to the Event-B tool support, Rodin. It identifies such incompatibilities as syntactical and type errors in the resulting composite machine.

To apply shared event composition in our definition of the Constelle semantics, we represent it as the following function:

$$\begin{aligned}&\mathsf{SharedEventComposition} :\\ &\mathbb{P}(\mathsf{Machine}) \rightarrow \mathbb{P}(\mathbb{P}(\mathsf{Event})\\ &\times \mathbb{P}(\mathbb{P}(\mathsf{Parameter}))) \rightarrow \mathsf{Machine}\end{aligned} \tag{11}$$

Here a set of constituent $\mathsf{Machines}$ is composed into a new (composite) $\mathsf{Machine}$ using a configuration of type $\mathbb{P}(\mathbb{P}(\mathsf{Event}) \times \mathbb{P}(\mathbb{P}(\mathsf{Parameter})))$. A configuration is formed as a set of composite events of the resulting machine. Each of these composite events is composed of a pair consisting of a set of synchronized events $\mathbb{P}(\mathsf{Event})$ coming from different constituent machines and of sets of sets of parameters $\mathbb{P}(\mathbb{P}(\mathsf{Parameter}))$ shared by these events. Thus, an element of type $\mathbb{P}(\mathsf{Parameter})$ represents overlapping parameters of constituent events that form a single (shared) parameter in the resulting composite event.

For example, the machine depicted in Fig. 9 is computed from the machines *driver1_queue*, *driver2_queue*, and *distributor_request* using the following configuration (here we use prefixes of the form 'machine_name/' to show from which constituent machine each event or parameter is taken):

$$\begin{aligned}\{\ &(\{\mathit{distributor\_request/taskStm}\}, \varnothing),\\ &\Big(\{\mathit{driver1\_queue/armActionStm},\\ &\quad \mathit{distributor\_request/armActionStm}\},\\ &\quad \{\{\mathit{driver1\_queue/action}, \mathit{distributor\_request/action}\}\}\Big),\\ &\Big(\{\mathit{driver2\_queue/handActionStm},\\ &\quad \mathit{distributor\_request/handActionStm}\},\\ &\quad \{\{\mathit{driver2\_queue/action}, \mathit{distributor\_request/action}\}\}\Big),\\ &\dots\ \}\end{aligned} \tag{12}$$

Composition of semantic templates uses shared event composition in the following way:

$$\begin{aligned}&\mathsf{Compose}(\text{module})(\text{tmplts}) =\\ &\qquad \mathsf{SharedEventComposition}(\\ &\qquad\ \{t.\mathsf{eventbmachine} \mid t \in \text{tmplts}\})(\text{config})\\ &\qquad \circ\, \mathsf{ReconstructComposedTemplate}(\text{module})\\ &\textbf{where}\ \text{config} =\\ &\quad \Big\{ \Big(\{x.\mathsf{eventbElement} \mid x.\mathsf{implements} = op\ \wedge\\ &\qquad\quad x \in t.\mathsf{elements} \wedge t \in \text{tmplts}\},\\ &\qquad \big\{\{x.\mathsf{eventbElement} \mid x.\mathsf{implements} = dp\ \wedge\\ &\qquad\quad x \in t.\mathsf{elements} \wedge t \in \text{tmplts}\}\\ &\qquad \mid dp \in op.\mathsf{signature}\big\} \Big)\\ &\quad \mid op \in \text{module}.\mathsf{interface} \Big\}\end{aligned} \tag{13}$$

Here the resulting machine is composed of the machines of the input semantic templates (tmplts). The configuration of the composition (config) is derived from the interface of the semantic module (module). Namely, for each operation of the interface ($op \in$ module.interface) we select from different templates ($t \in$ tmplts) specification elements ($x \in t$.elements) that implement these operations (x.implements $= op$). Event-B elements of these specification elements (x.eventbElement) determine which events should be synchronized (*i.e.,* composed). The configuration of sharing Event-B parameters is derived in the same way for each dynamic parameter of the operation ($dp \in op$.signature).

The resulting composite Event-B machine is wrapped into a semantic template using an auxiliary transformation ReconstructComposedTemplate:

$$\begin{aligned}&\mathsf{ReconstructComposedTemplate}:\\ &\mathsf{SemanticInterface} \rightarrow \mathsf{Machine} \rightarrow \mathsf{SemanticTemplate}\end{aligned} \tag{14}$$

To wrap all Event-B elements of the composite Machine, this transformation generates specification elements of the resulting SemanticTemplate based on the SemanticInterface that this machine implements.

5 Implementation and results

Using the MDE techniques, we implemented our approach as a set of tools for creating specification templates and applying them to define the dynamic semantics of a DSL. Our implementation is a proof of concept of the approach presented in this paper. It allows for achieving the practical benefits of having a formal specification of the DSL dynamic semantics, as identified and discussed in Sect. 2.1. In this section we give an overview of our implementation (Sect. 5.1), discuss its results (Sect. 5.2), identify the functionality that is missing to make the approach mature enough for the practical usage, and present our vision on the potentials of our approach and possible candidates for specification templates (Sect. 5.3).

5.1 Implementation of the approach

An overview of our implementation is presented in Fig. 10. Its key components (on the top) are Constelle, which comprises the metamodels presented in Sects. 3.2 and 3.4, and the Constelle-to-Event-B model transformation, which is implemented according to the definition presented in Sect. 4. The library (in the middle) includes both semantic interfaces (library.constellecore) and specification templates (impl.templateslibrary), which implement these semantic interfaces in the form of Event-B code (library.eventb on the right). The dynamic semantics of a DSL is defined in Constelle as a composition of the semantic interfaces, which are specialized using the DSL constructs introduced by the DSL metamodel (dsl.ecore on the left). The execution of the Constelle-to-Event-B model transformation (on the bottom) uses this definition to automatically generate an Event-B specification from each concrete DSL model.

Our tool set (denoted as Constelle workbench in Fig. 10) supports (automatic) creation of specification templates on the basis of Event-B specifications, editing of a Constelle definition in the form of the table notation (introduced in Sect. 3.3), and automatic generation of the corresponding Event-B specifications. Figure 11 shows the semantic module `Robotic Arm Parallel` in the table editor of the Constelle language.

To implement the Constelle workbench, we employed the following MDE techniques provided in the Eclipse platform: Ecore metamodeling tools (EMF), the QVTo model transfor-

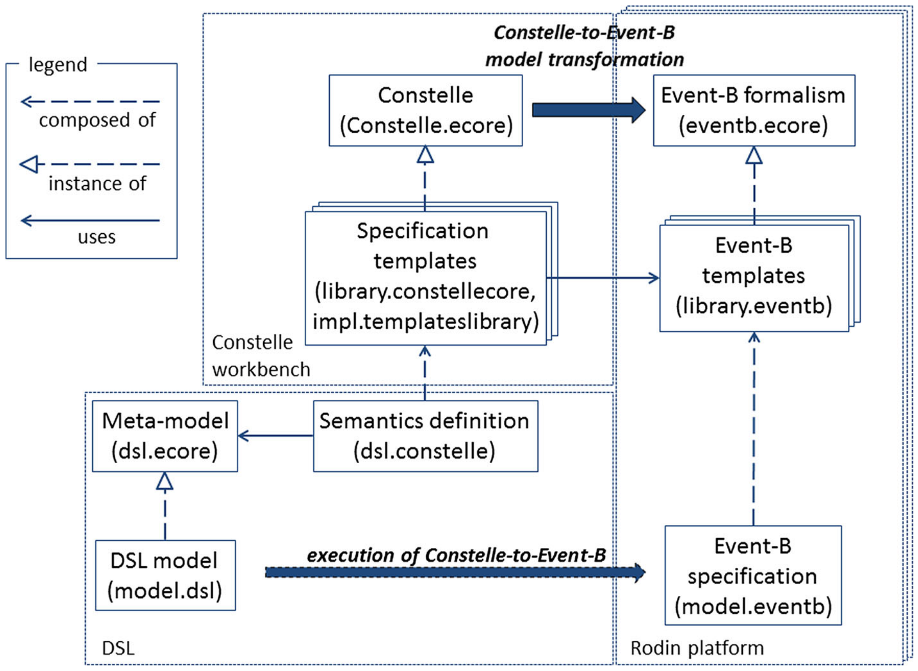
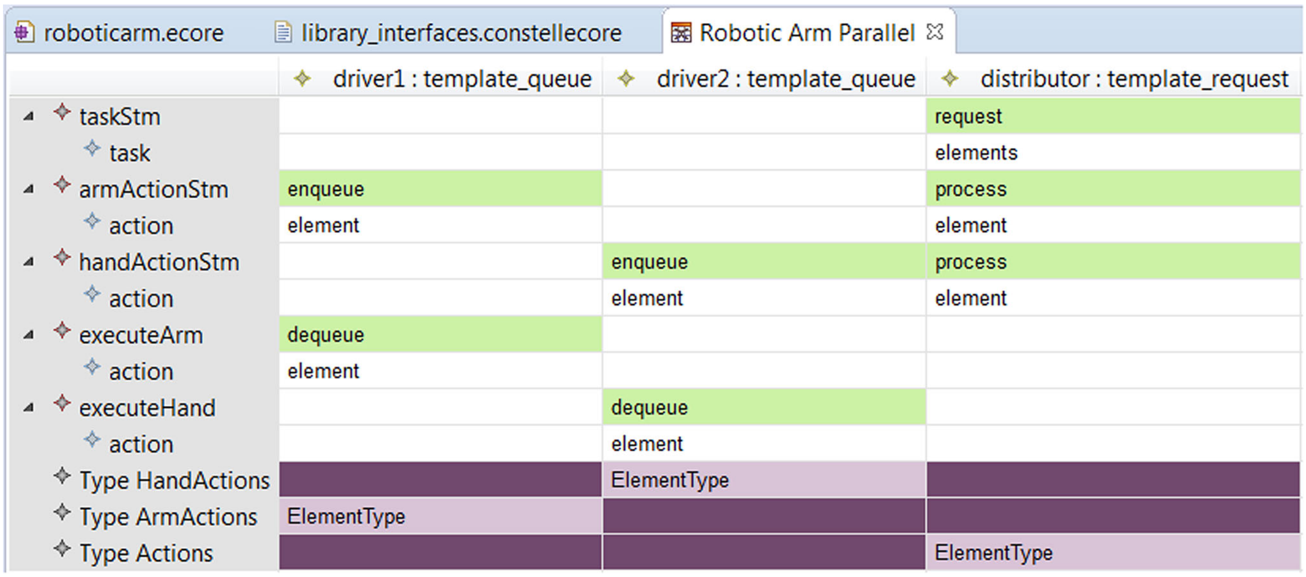

legend
composed of
instance of
uses
Constelle-to-Event-B model transformation
Constelle (Constelle.ecore)
Event-B formalism (eventb.ecore)
Specification templates (library.constellecore, impl.templateslibrary)
Event-B templates (library.eventb)
Constelle workbench
Meta-model (dsl.ecore)
Semantics definition (dsl.constelle)
DSL model (model.dsl)
execution of Constelle-to-Event-B
Event-B specification (model.eventb)
DSL
Rodin platform

Fig. 10 The architecture of the Constelle implementation



	driver1 : template_queue	driver2 : template_queue	distributor : template_request
taskStm			request
task			elements
armActionStm	enqueue		process
action	element		element
handActionStm		enqueue	process
action		element	element
executeArm	dequeue		
action	element		
executeHand		dequeue	
action		element	
Type HandActions		ElementType	
Type ArmActions	ElementType		
Type Actions			ElementType

Fig. 11 Screen shot of the Constelle table editor

mation language, and the Sirius graphical editor generator.[6] Our QVTo model transformation is aligned with the definition of the semantics of Constelle given in Sect. 4 according to the methodology described in [53]. As the reference implementation of the Event-B* techniques, we used the Rodin plug-ins for generic instantiation[7] and for shared event composition.[8]

5.2 Results

The Constelle DSL combined with the tool support of the Event-B formalism, Rodin, implement the use cases listed in Sect. 2.1:

– We can prototype a DSL implementation using Constelle (and generate the corresponding Event-B specifications);
– We can check the applicability and the compatibility of invoked specification templates by ensuring syntactic and logical correctness of the generated Event-B specifica-

[6] https://eclipse.org/sirius/.

[7] https://sourceforge.net/projects/gen-inst/.

[8] http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B.

Table 3 Relative characteristics of specification templates versus semantic modules

Metric	Specification templates			Semantic modules	
	Queue	Request	Partial Order	Robotic Arm Parallel	Robotic Arm Sequential
Lines of code	21	15	33	51	82
Number of proof obligations	5	0	12	10	26
of those can be ignored	0%	0%	0%	100%	100%

tions (the logical correctness of an Event-B specification is defined in the form of proof obligations [2]);

– We can verify that the prototyped DSL fulfills certain properties by analyzing the generated Event-B specifications using (automatic) provers and/or model checkers of Rodin (for example, using the AtelierB prover[9]);
– We can validate the prototyped DSL implementation against (informal) requirements by executing the generated Event-B specifications (for example, using the ProB animator[10]);
– We can wrap the animation of the generated Event-B specifications in a domain-specific visualization (for example, using BMotion Studio as presented in [51]).

Note that in the listed use cases Constelle plays a role only in prototyping the DSL implementation and creating the corresponding specification of the dynamic semantics. All types of the analysis are carried out by the back-end formalism: from the syntactic analysis of the generated Event-B code to model checking. As a result, to benefit from having a definition of the DSL dynamic semantics, one needs to be able to work with the back-end formalism. To mitigate the potential difficulties of employing two different languages (Constelle and the back-end formalism), we rely on techniques that allow for optimizing the verification of the resulting Event-B specification by reusing proof obligations that are already discharged for the invoked specification templates.

Proof obligations determine what should be proved for an Event-B specification in order to ensure that the specified (system) design is consistent, feasible, and complete (*i.e.,* logically correct). In Rodin, a set of proof obligations is generated automatically for each Event-B specification. Proof obligations can be discharged (*i.e.,* proved) using automatic and interactive provers. Generic instantiation and shared event composition support reuse of proof obligations that are already discharged for a generic and/or constituent Event-B machine [44].

For example, Table 3 shows the relative metrics of the Event-B machines implementing the specification templates vs. the Event-B machines generated for the semantic modules (composed of these specification templates): lines of Event-B code and number of proof obligations. The bottom line of the table shows the proportion of the proof obligations that do not need to be discharged for the semantic modules, as they already have been discharged for the invoked specification templates.

5.3 Future work

An inverse mapping from compilation and/or analysis results provided by the back-end tools (back) to the concepts and constructs used in a Constelle definition will complement our implementation of Constelle, supporting a DSL developer who does not know the back-end formalism and the specifics of its tools support. Such a mapping will facilitate interpreting the feedback provided by the Rodin tooling in terms of the definition of the dynamic semantics of a DSL. The resulting set of Constelle mappings (to and from Event-B) will become a mature and self-contained workbench that supports the complete process of developing and maintaining the dynamic semantics of a DSL. We consider investigation of the feasibility and construction of such an inverse mapping as the future work.

Multiple back-end formalisms and platforms can be potentially used in our approach. This fact is depicted in Fig. 10 (on the right) as multiple components representing the Rodin platform. For example, for each specification template (*i.e.,* for each semantic interface) of our library we can assign the corresponding C-code that implements it. A Constelle-to-C code generator would compose these implementations for a definition of the DSL dynamic semantics in Constelle and, thus, construct the corresponding implementation of the DSL in C. In order to be able to use C-code as a back-end formalism, we need to answer the following questions: how to compose fragments of C-code according to a Constelle model; and more particularly, whether the order in which code fragments are composed keeps the dynamic semantics unchanged in the resulting program.

An example of another back-end platform is presented in [51]. There we use BMotion Studio [26] to create a domain-specific visualization of the Event-B specifications of the DSL. The visualization mimics the graphical notation of the DSL and runs on top of the animation of an Event-B specification. As a result, the animation can be performed by DSL engineers who are not familiar with the notation of Event-

[9] http://www.atelierb.eu/en/atelier-b-tools/.

[10] http://www3.hhu.de/stups/prob/.

B. The results of this work can potentially be generalized by identifying and collecting *visualization templates*. If each of the semantic interfaces in our library is coupled with a corresponding visualization template, then a Constelle-to-BMotionStudio transformation would (semi-)automatically generate a domain-specific visualization for an arbitrary DSL based on its dynamic semantics definition in Constelle.

In this way, Constelle and a library of specification templates can serve as a *common semantic domain*, which is used as a (common) source of semantic mappings targeting various execution platforms. As a result, the consistency between different DSL translations is handled within these semantic mappings and can be reused for different DSLs.

The potential candidates for specification templates can be found using various methods and empirics. First of all, these can be well-known software design patterns and architecture styles [15], such as Observer and Blackboard patterns, multi-layered and peer-to-peer architectures, etc. Another source of established software development practice is peer-reviewed source code libraries, such as the Boost libraries of C++ code.[11] For example, such components as Boost Flyweight and Boost Graph Library can be investigated as candidates for specification templates.

Following our idea that specification templates capture a DSL horizontal domain, candidates for specification templates can be derived from constructs of horizontal DSLs. For example, the Reo language [6] uses a common set of primitive communication channels, typical for concurrent applications: synchronous, lossy, buffered, etc. These can be added to our library of specification templates.

6 Related work

The concepts of templates and/or patterns have been applied to various components of DSLs in order to facilitate reuse of their design. The existing work includes studies on metamodel templates (such as [7] and [42]); composition and reuse of concrete syntax (both for textual [29] and graphical notations [38]); and reuse of definitions of the dynamic semantics. For specifying reusable fragments (building blocks) of dynamic semantics and/or weaving/-composing them together, some of the studies use *informal (or semiformal) notations*: transformation languages (such as Epsilon Object Language, EOL, in [29]), UML activity diagrams (in [43]), and UML state and sequence diagrams (in [23]).

There exist a number of formal notations that allow for modular definition of the dynamic semantics of general purpose programming languages (GPLs) using (existing or to be established) libraries of reusable modules. For example, TinkerType [27], Modular SOS (MSOS) [31], DynSem [55], K framework [41] achieve an AOP-like modularity for term reduction (rewriting or inference) rules. The latter two formal notations allow for the automatic generation of an AST (abstract syntax tree) interpreter, which can be used as a reference implementation of the programming language or for formal analysis of the specified dynamic semantics.

In our approach we aim for a precise and executable definition of the dynamic semantics of a DSL that captures software solutions rather than requirements. For this, we employ a formalism that has a solid theory and extensive tools support, but is not specifically designed for defining the dynamic semantics of GPLs or DSLs. In Sect. 6.1 we discuss in detail the existing work that uses various formal methods for defining reusable (intermediate) building blocks for composing the dynamic semantics of DSLs. In the relation to the concept of specification templates, in Sect. 6.2 we look into existing techniques for reusing formal specifications of software systems.

Note that here we do not consider reuse of a DSL and its formal analysis via embedding this DSL into another DSL, as it is done by Ratiu et al. [40]. In their work, the reuse of the semantic mapping of a DSL to a verification formalism is achieved through the clear separation of the DSL concepts from its environment, rather than through the composition of the embedded DSL (sub-language) with the hosting DSL (which realizes its environment).

6.1 Reusable building blocks for specifying dynamic semantics of DSLs

In [11] Dagand et al. propose Filet-o-Fish (FoF) as a semantic language for composing a DSL out of semantically rich building blocks. Technically, FoF is a ‘safe abstraction of C embedded in a functional language’ (in their case, Haskell). For this, Haskell functions wrap various string concatenations that can generate fragments of C-code. As a result, FoF abstracts from the details of the C syntax and provides building blocks for specifying the dynamic semantics of a DSL. Such building blocks are invoked and composed together in the form of (higher-order) functions using the standard combinators of Haskell (such as *folding* for traversing an abstract syntax tree, AST). Thus, in FoF the dynamic semantics of a DSL is defined as a Haskell program using available code generators. On the one hand, the FoF-to-C compiler generates the corresponding C-code from such a definition. On the other hand, various techniques and tools for Haskell allow for validation (for example, random testing) and verification (*i.e.,* proofs of correctness) of the definition of the DSL dynamic semantics at the level of this definition. Unfortunately, the authors do not discuss the nature of their building blocks. Therefore, it is not clear if we can use FoF to introduce and to invoke the building blocks proposed in our approach: soft-

[11] www.boost.org.

ware design solutions commonly used in the implementation of DSLs.

In [10] Chen et al. propose *semantic units* as an intermediate common language for defining dynamic semantics of DS(M)Ls. Semantic units capture the formal operational semantics for a set of basic models of computations. These can be either basic behavioral categories, such as finite state machine (FSM), timed automaton (TA), and hybrid automaton (HA); or basic component interaction categories, such as synchronous data flow (SDF), communicating sequential process (CSP) and process networks (PN). The semantic units are specified using the Abstract State Machines (ASM) formalism. The dynamic semantics of a DSL is defined as a model transformation between the metamodel (abstract syntax) of the DSL and the metamodel that captures the syntax of the ASM abstract data model of a selected semantic unit. The authors call such a technique *semantic anchoring*. Comparing to our semantic templates, semantic units are general purpose computation models, rather than specific software solutions forming the horizontal domain of a DSL or a family of DSLs.

In [9] Chen et al. develop a method for the composition of semantic units. In the same way as in our approach, the dynamic semantics of a DSL is built hierarchically as a composition of primary semantic units and newly derived semantic units (composed of the primary ones). To specify such a composition the authors use the composition mechanisms of the ASM formalism, such as invocations of primary ASM specifications and adding new (ad-hoc) constraints. As a result, the interaction of constituent semantic units and the mapping between their data structures are tangled over the ASM code. In our approach we overcome this issue by using the table notation for composing specification templates when defining the dynamic semantics of a DSL.

Mannadiar and Vangheluwe [28] in their position paper elaborate on the work by Chen et al. and describe an idea of a *semantic template*–a combination of a metamodel template (*i.e.,* a parametrized metamodel fragment) with its semantic anchoring (*i.e.,* its model transformation to the ASM formalism). The authors propose to define a DSL as a combination of such semantic templates, thus automatically constructing the DSL metamodel and the dynamic semantics specification. However, there is no follow-up work and/or proof of concept for the proposed approach.

In [47] Simko extends the approach of semantic anchoring to *denotational specification* of the dynamic semantics of cyber-physical systems (CPS) modeling languages. The author identifies the following semantic units, typical for the CPS domain: differential algebraic equations, difference equations, and finite state machines. Comparing to the FORMULA formalism used in this work, Event-B does not allow for expressing differential algebraic and difference equations. Our approach is based on providing an *operational specification* of the dynamic semantics of DSLs. In particular, we specify a DSL dynamic semantics as a solution rather than requirements (as discussed in Sect. 2.1).

In [36,37] Pedro et al. propose a compositional and incremental approach for prototyping DS(M)Ls, where they focus on reuse of metamodel fragments (which they call *domain concepts*) together with the model transformations that capture the dynamic semantics of these fragments. A domain concept is a brick that represents a basic idea that can appear in one or several DS(M)Ls. A domain concept is defined as a metamodel, a set of the metamodel elements that can be parameterized (*i.e.,* replaced by *effective* parameters), and a model transformation to a formal executable language (in their case, to concurrent object-oriented Petri nets, CO-OPN) that captures the dynamic semantics of this domain concept. The definition of a DS(M)L consists of the metamodel composition and the transformation composition. The metamodel composition is an iterative replacement of formal parameters of some domain concept with elements (*i.e.,* classes, attributes) of another metamodel. In this way, various domain concepts can be composed with each other or with specific constructs of the DS(M)L being defined. The corresponding replacement of parameters (*i.e.,* instantiation) takes place in the model transformation of the original domain concept. Moreover, the instantiated transformation is extended with additional transformation rules that capture a more precise and more specific semantics of the DS(M)L. As a result, the transformation language (in their case, ATL) serves as a main formalism for specifying the dynamic semantics of a DSL. Moreover, different from Constelle, this approach does not provide any formal theory behind the instantiation and composition of dynamic semantics of constituent building blocks (in their terms, domain concepts).

A formal theory for composition of the dynamic semantics out of constituent building blocks can allow for reuse of verification results obtained for these building blocks. This important result is the main focus of product lines of programming languages [12] and modular monadic meta-theory (3MT) [13] by Delaware et al. These approaches allow for reuse of proofs implemented in the Coq proof assistant for various *language features* in the *denotational semantics* of a (functional) programming language composed out of these features. In Constelle we employ Event-B generic instantiation and shared event composition to achieve reuse of proof obligations.

6.2 Composition and reuse of formal specifications

The idea of applying the principles of AOP to the formal specification of software systems appeared shortly after the introduction of AOP. In [21] Kellomaki and Mikkonen not only propose the gradual introduction of *aspects of collective behavior* in a specification of reactive distributed system, but also describe how such aspects can be stored as *generic templates*, allowing for reuse of both design and verification effort.

Kellomaki and Mikkonen use the DisCo specification language and in their later studies introduce the Oscid specification language [19], an experimental variant of DisCo. An aspect is defined as a *superposition step*, which refines an existing specification by introducing new state variables, invariants, and actions. Comparing to Event-B, the superposition mechanism resembles shared event composition, rather than the Event-B refinement. Particularly, superposition preserves safety properties by construction. To be able to archive and reuse such a superposition step, the authors turn it into a template by introducing template parameters and specifying what behavior these parameters should realize. In Constelle only static parameters (of a structural interface) can be used for instantiating a template. In contrast, Kellomaki and Mikkonen include actions into their template parameters and specify these actions. As a result, an instantiated template (an aspect) can be imposed (applied) only if the original specification realizes certain behavior.

Using templates of superposition steps, one can design a distributed system adding new aspects to it one-by-one, forming a *specification branch*. In [20] Kellomaki extends this approach with the possibility to merge (compose) specification branches together. Comparing to the table notation of Constelle, both DisCo and Oscid use the '…' symbol as a weaver notation: to indicate where the old code appears in the new specification. In [16] the Oscid specification language is applied for specifying and instantiating two (OOP) design patterns: Observer and Memento. Unfortunately, there is no follow-up work.

We base our method on the Event-B techniques of generic instantiation and shared event composition. In [45] Silva and Butler propose to instantiate *chains of refinements* of Event-B machines. *Refinement* is an Event-B (formal) technique that allows for gradual introduction of details in an Event-B specification. A chain of refinements is a sequence of Event-B machines, where each next machine is a refinement of a previous one. This technique (potentially) allows for extending our method to the *templates of chains of refinements* and, thus, for the reuse of a system design specified on different levels of abstraction (from an overview of the required behavior to the implementation details).

In [17] Hoang et al. propose a concept of the *Event-B pattern*, which is similar to the superposition step of Kellomaki and Mikkonen. An Event-B pattern is a (generic and/or reusable) refinement step that introduces new details to the abstract machine of the pattern. An application of the pattern (*i.e.,* of the instantiated refinement step) requires syntactical matching the abstract machine of the pattern with the Event-B machine under construction. The authors identify the following patterns of communication protocols: Single Message Communication, Request/Confirm/Reject, and Asynchronous Multiple Message Communication (with or without Repetition). Comparing to the approach of Kellomaki and Mikkonen (and to Constelle), an Event-B pattern does not have an explicit description of template parameters, as Hoang et al. use purely the Event-B notation. The syntactic matching of the pattern is semiautomated. This makes it hard to reuse patterns in other formalisms, and to capture design of a specification (*i.e.,* of a system under specification) in terms of pattern applications (as it is done in Constelle tables).

The ancestor of the Event-B formalism, the B method is used in [8] to specify (OOP) design patterns and to realize different reuse mechanisms for them. Particularly, instantiation of a design pattern is implemented in B by the inclusion of the machine specifying the design pattern and by redefining (in essence, renaming) its variables. Composition of multiple design patterns is achieved through the invocation of the operations of different patterns in a new (composite) operation and/or linking or merging the variables of the patterns. Extension of a design pattern is realized using the B refinement mechanism. This study shows that, in principle the *design patterns-based approach* can be realized in formal methods through proper code conventions, in the same way as it is done in software development for general purpose programming languages. The practice shows that this approach requires discipline and good understanding of a chosen formalism.

7 Conclusion

In this work, we developed and demonstrated a new method for defining the dynamic semantics of DSLs. The key point of our method is an intermediate semantic domain that splits the semantic mapping from a DSL to an execution platform (or a specification formalism) into two steps. As such an intermediate semantic domain we use software design solutions that are typically used in the DSL implementation, *i.e.,* concepts that form the horizontal domain of the DSL. Thus, we define the dynamic semantics of a DSL as a mapping from the language constructs (forming the vertical domain of the DSL) to the horizontal concepts. In this way, we do not propose a (yet another) intermediate language, universal for defining dynamic semantics of all possible DSLs, but we rather propose an intermediate step in the definition of the dynamic semantics of a DSL and support this step with the corresponding expressive means.

To capture the mapping from the DSL constructs to the intermediate semantic domain, we use the notation of a table: the DSL vertical domain is represented in the table rows, the DSL horizontal domain is represented in the table columns, and the mapping is represented in their intersections. The second step of the semantic mapping, from the intermediate semantic domain to the specification formalism, is realized through specification templates.

We implemented this method in the form of the Constelle language and employed the Event-B formalism as a carrier for specifying behavior. Constelle applies ideas of generic programming and aspect-oriented programming to the world of formal methods and provides a front-end that wraps the formalism. Specifically, the Constelle-to-Event-B model transformation automatically generates the corresponding Event-B specifications from a Constelle model. From this point of view, the approach of specification templates and the Constelle language are not restricted to the scope of dynamic semantics of DSLs and can facilitate application of formal methods in software development process.

The proposed method requires further evaluation. In particular, the following research questions should be addressed in future work. How to identify specification templates and how reusable are these specification templates across various application domains? What is the scope of Constelle, *i.e.,* what kind of DSLs can be defined using Constelle and specification templates? How scalable is the proposed approach, *i.e.,* whether the benefits of applying it to a real (industrial-size) DSL are worth the effort?

As the first step, we have designed and performed a validation study on applying Constelle for defining the dynamic semantics of another DSL by a third party. The validation study gives valuable insights into pragmatics of the proposed approach and confirms that defining the dynamic semantics of a DSL is beneficial and certain design solutions can be reused in the form of specification templates. The details of the study setup and the analysis of its results can be found in our PhD dissertation [50].

Acknowledgements This work would not have been possible without guidance, reviews, and thorough discussions of our supervisor Tim Willemse (TU/e). We thank our colleagues from the COREF project (ASML and TU/e) and especially the project leader Mark van den Brand. We would also like to show our gratitude to Marc Hamilton (Altran) for sharing very interesting insights into the world of DSL engineering.

Appendix 1: Event-B specification of the partial order template

See Figs. 12 and 13.

```
CONTEXT   template_partialorder_context
SETS
      PosetElement
END
```

Fig. 12 Event-B context for the partial order specification template

Fig. 13 Event-B machine for the partial order specification template

```
MACHINE template_partialorder_machine
SEES template_partialorder_context
VARIABLES
      posetBody, posetOrder
INVARIANTS
inv1 : $posetBody \subseteq PosetElement$
inv2 : $posetOrder \in posetBody \leftrightarrow posetBody$
inv3 : $\forall x, y \cdot x \mapsto y \in posetOrder \Rightarrow x \neq y$
inv4 : $\forall x, y \cdot x \mapsto y \in posetOrder \Rightarrow y \mapsto x \notin posetOrder$
inv5 : $\forall a, b, c \cdot a \mapsto b \in posetOrder \land b \mapsto c \in posetOrder \Rightarrow a \mapsto c \in posetOrder$
EVENTS
Initialisation
  begin
     act1 : $posetBody := \varnothing$
     act2 : $posetOrder := \varnothing$
  end
Event   NewPartialOrder $\widehat{=}$
    any poset, order
    where
     grd1 : $poset \subseteq PosetElement$
     grd2 : $order \in poset \leftrightarrow poset$
     grd3 : $\forall x, y \cdot x \mapsto y \in order \Rightarrow x \neq y$
     grd4 : $\forall x, y \cdot x \mapsto y \in order \Rightarrow y \mapsto x \notin order$
     grd5 : $\forall a, b, c \cdot a \mapsto b \in order \land b \mapsto c \in order \Rightarrow a \mapsto c \in order$
    then
     act1 : $posetBody := poset$
     act2 : $posetOrder := order$
  end
Event   GetMaximalElement $\widehat{=}$
    any maximal
    where
     grd1 : $maximal \in posetBody$
     grd2 : $\forall x \cdot x \in posetBody \land x \neq maximal \Rightarrow maximal \mapsto x \notin posetOrder$
    then
         skip
  end
Event   RemoveElement $\widehat{=}$
    any element, elementRelations
    where
     grd1 : $element \in PosetElement$
     grd2 : $elementRelations = \{x, y \cdot x \mapsto y \in posetOrder \land (x = element \lor y = element) | x \mapsto y\}$
    then
     act1 : $posetBody := posetBody \setminus \{element\}$
     act2 : $posetOrder := posetOrder \setminus elementRelations$
  end
END
```

Appendix 2:Event-B specification generated for the semantic module Robotic Arm Parallel

See Figs. 14 and 15.

CONTEXT roboticarm_structure_context

SETS

Actions

CONSTANTS

$HandActions$

$ArmActions$

$TURN\_LEFT$

$TURN\_RIGHT$

$MOVE\_UP$

$MOVE\_DOWN$

$GRAB$

$RELEASE$

$ROTATE\_LEFT$

$ROTATE\_RIGHT$

AXIOMS

axm1 : $partition(Actions, ArmActions, HandActions)$

axm2 : $partition(ArmActions, \{TURN\_LEFT\}, \{MOVE\_UP\}, \{TURN\_RIGHT\}, \{MOVE\_DOWN\})$

axm3 : $partition(HandActions, \{GRAB\}, \{RELEASE\}, \{ROTATE\_LEFT\}, \{ROTATE\_RIGHT\})$

END

Fig. 14 Event-B context for the robotic arm DSL

Fig. 15 Event-B machine for the semantic module robotic arm parallel (page 1)

```
MACHINE  RoboticArmParallel_machine
SEES  roboticarm_structure_context
VARIABLES
      driver1_queue, driver2_queue, distributor_request_body
INVARIANTS
distributor_inv1 : distributor_request_body ∈ ℙ(Actions)
driver2_inv1 : driver2_queue ∈ ℕ ⇸ HandActions
driver1_inv1 : driver1_queue ∈ ℕ ⇸ ArmActions
EVENTS
Initialisation
  begin
     driver1_act1 : driver1_queue := ∅
     driver2_act1 : driver2_queue := ∅
     distributor_act1 : distributor_request_body := ∅
  end
Event  taskStm ≙
    any  task
    where
     distributor_grd1 : task ∈ ℙ(Actions)
     distributor_grd2 : distributor_request_body = ∅
    then
     distributor_act1 : distributor_request_body := task
  end
Event  handActionStm ≙
    any  driver2_index, action
    where
     distributor_grd1 : action ∈ distributor_request_body
     driver2_grd1 : action ∈ HandActions
     driver2_grd2 : driver2_index ∈ ℕ
     driver2_grd3 : driver2_queue ≠ ∅ ⇒ (∀i·i ∈ dom(driver2_queue) ⇒ driver2_index > i)
     driver2_grd4 : {driver2_index ↦ action} ∈ ℕ ⇸ HandActions
     driver2_grd5 : driver2_index ∉ dom(driver2_queue)
    then
     distributor_act1 : distributor_request_body := distributor_request_body \ {action}
     driver2_act1 : driver2_queue := driver2_queue ∪ {driver2_index ↦ action}
  end
Event  armActionStm ≙
    any  driver1_index, action
    where
     distributor_grd1 : action ∈ distributor_request_body
     driver1_grd1 : action ∈ ArmActions
     driver1_grd2 : driver1_index ∈ ℕ
     driver1_grd3 : driver1_queue ≠ ∅ ⇒ (∀i·i ∈ dom(driver1_queue) ⇒ driver1_index > i)
     driver1_grd4 : {driver1_index ↦ action} ∈ ℕ ⇸ ArmActions
     driver1_grd5 : driver1_index ∉ dom(driver1_queue)
    then
     distributor_act1 : distributor_request_body := distributor_request_body \ {action}
     driver1_act1 : driver1_queue := driver1_queue ∪ {driver1_index ↦ action}
  end
Event  executeArm ≙
    any  driver1_index, action
    where
     driver1_grd1 : driver1_index ↦ action ∈ driver1_queue
     driver1_grd2 : ∀i·i ∈ dom(driver1_queue) ⇒ driver1_index ≤ i
    then
     driver1_act1 : driver1_queue := driver1_queue \ {driver1_index ↦ action}
  end
Event  executeHand ≙
    any  driver2_index, action
    where
     driver2_grd1 : driver2_index ↦ action ∈ driver2_queue
     driver2_grd2 : ∀i·i ∈ dom(driver2_queue) ⇒ driver2_index ≤ i
    then
     driver2_act1 : driver2_queue := driver2_queue \ {driver2_index ↦ action}
  end
END
```

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, vol. 1. Cambridge Univ Press, Cambridge (2010)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-B. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(6), 447–466 (2010)
4. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to event-B. Fundam. Inform. **77**(1–2), 1–28 (2007)
5. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Amsterdam (1986)
6. Arbab, F.: Proper Protocol. Springer, Berlin (2016)
7. Berg, H., Møller-Pedersen, B.: Type-safe symmetric composition of metamodels using templates. Syst. Anal. Model. Theory Pract. **7744**, 160–178 (2013)
8. Blazy, S., Gervais, F., Laleau, R.: Reuse of specification patterns with the B method. In: ZB 2003: Formal Specification and Development in Z and B, volume 2651 of Lecture Notes in Computer Science, pp. 40–57. Springer, Berlin (2003)
9. Chen, K., Porter, J., Sztipanovits, J., Neema, S.: Compositional specification of behavioral semantics for domain-specific modeling languages. Int. J. Semant. Comput. **3**, 31–56 (2009)
10. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. European Conference on Model Driven Architecture—Foundations and Applications, pp. 115–129 (2005)
11. Dagand, P.-E., Baumann, A., Roscoe, T.: Filet-o-fish: practical and dependable domain-specific languages for OS development. In: Proceedings of the 5th Workshop on Programming Languages and Operating Systems, PLOS '09, pp. 5:1–5:5. ACM (2009)
12. Delaware, B., Cook, W.R., Batory, D.S.: Product lines of theorems. In: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, pp. 595–608 (2011)
13. Delaware, B. Keuchel, S., Schrijvers, T., Oliveira, B.C. d.S.: Modular monadic meta-theory. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'13, pp. 319–330 (2013)
14. Deursen, A.V., Klint, P., Visser, J.: domain-specific languages: an annotated bibliography. ACM Sigplan Not. **35**, 26–36 (2000)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Amsterdam (1994)
16. Helin, J., Kellomäki, P., Mikkonen, T.: Patterns of collective behavior in Ocsid. In: Taibi, T. (eds.) Design Pattern Formalization Techniques. IGI Publishing, Hershey, pp. 73–93
17. Hoang, T.S., Fürst, A., Abrial, J.: Event-b patterns and their tool support. Softw. Syst. Model. **12**(2), 229–244 (2013)
18. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Upper Saddle River (1985)
19. Kellomäki, P.: A formal basis for aspect-oriented specification with superposition. In: The FOAL Workshop on Foundations of Aspect-Oriented Languages, pp. 27–32 (2002)
20. Kellomäki, P.: Composing distributed systems from reusable aspects of behavior. In: Distributed Computing Systems Workshops, IEEE Press, pp. 481–486 (2002)
21. Kellomäki, P., Mikkonen, T.: Design templates for collective behavior. In: ECOOP 14th European Conference on Object-Oriented Programming, pp. 277–295 (2000)
22. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP, Springer-Verlag LNCS, pp. 220–242 (1997)
23. Kienzle, J., Al Abed, W., Klein, J.: Aspect-oriented multi-view modeling. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD '09, pp. 87–98 (2009)
24. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages using Metamodels. Addison-Wesley, Boston (2008)
25. Kmieć, P.: The Unofficial LEGO Technic Builder's Guide. No Starch Press, San Francisco (2013)
26. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising event-B models with B-motion studio. In: Formal Methods for Industrial Critical Systems, FMICS 2009, pp. 202–204 (2009)
27. Levin, M.Y., Pierce, B.C.: Tinkertype: a language for playing with formal systems. J. Funct. Program. **13**(2), 295–316 (2003)
28. Mannadiar, R., Vangheluwe, H.: Domain-specific Engineering of Domain-specific Languages. In: Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM '10, pp. 11:1–11:6. ACM (2010)
29. Meyers, B., Cicchetti, A., Guerra, E., de Lara, J.: Composing textual modelling languages in practice. In: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012, Innsbruck, Austria, pp. 31–36 (2012)
30. Mosses, P.: Theory and practice of action semantics. In: Penczek, W., Zalas, A. (eds.) Mathematical Foundations of Computer Science 1996, vol. 1113 of Lecture Notes in Computer Science, pp. 37–61. Springer, Berlin (1996)
31. Mosses, P.: Modular structural operational semantics. J. Log. Algebr. Progr. **60–61**, 195–228 (2004)
32. Musser, D., Stepanov, A.A.: Generic programming. In: Gianni, P.M. (ed.) Symbolic and Algebraic Computation: ISSAC 88, pp. 13–25. Springer, Berlin (1988)
33. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley, London (1992)
34. Nipkow, T., Klein, G.: Concrete Semantics. Springer, Berlin (2014)
35. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, February 2015. Version 1.2. (2015)
36. Pedro, L.: A systematic language engineering approach for prototyping domain specific modelling languages. Ph.D. Dissertation (2009)
37. Pedro, L., Amaral, V., Buchs, D.: Foundations for a domain specific modeling language prototyping environment: a compositional approach. In: Proceedings of the 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM), pp. 20–27 (2008)
38. Pedro, L., Risoldi, M., Buchs, D., Barroca, B., Amaral, V.: Composing visual syntax for domain specific languages. In: Human-Computer Interaction. Novel Interaction Methods and Techniques, 13th International Conference, HCI International 2009, San Diego, Proceedings, Part II, pp. 889–898 (2009)
39. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Progr. **60–61**, 17–139 (2004)
40. Ratiu, D., Voelter, M., Molotnikov, Z., Schaetz, B.: Implementing modular domain specific languages and analyses. In: Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation, pp 35–40 (2012)
41. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebr. Progr. **79**(6), 397–434 (2010)
42. Schäfer, C. Kuhn, T., Trapp, M.: A pattern-based approach to DSL development. In: Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, and VMIL'11, SPLASH '11 Workshops, pp. 39–46. ACM (2011)
43. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Model

Driven Architecture- Foundations and Applications, vol. 4530 of Lecture Notes in Computer Science, pp. 157–171. Springer, Berlin (2007)
44. Silva, R.: Supporting Development of Event-B Models. Ph.D. thesis, University of Southampton (2012)
45. Silva, R., Butler, M.: Supporting reuse of event-B developments through generic instantiation. In: Breitman, K., Cavalcanti, A. (eds.) 11th International Conference on Formal Engineering Methods, ICFEM, vol. 5885 of Lecture Notes in Computer Science, pp. 466–484. Springer, Berlin (2009)
46. Silva, R., Butler, M.: Shared event composition/decomposition in event-B. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects (FMCO), pp. 122–141. Springer, Berlin (2010)
47. Simko, G.: Formal semantic specification of domain-specific modeling languages for cyber-physical systems. Ph.d. dissertation. Chapter 6: Reusable Semantic Units for Formalizing the Denotational Semantics of CPS Modeling Languages, pp. 59–75. Vanderbilt University (2014)
48. Snook, C., Fritz, F., Illisaov, A.: An EMF framework for event-B. In: Workshop on Tool Building in Formal Methods—ABZ Conference (2010)
49. Stappers, F.P.M., Weber, S., Reniers, M.A., Andova, S., Nagy, I.: Formalizing a domain specific language using SOS: an industrial case study. In: Software Language Engineering—4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers, pp. 223–242 (2011)
50. Tikhonova, U.: Engineering the dynamic semantics of domain specific languages. Ph.d. dissertation
51. Tikhonova, U., Manders, M., Boudewijns, R.: Visualization of formal specifications for understanding and debugging an industrial DSL. In: Human Oriented Formal Methods (HOFM), STAF Workshops, pp. 179–195 (2016)
52. Tikhonova, U., Manders, M., van den Brand, M.G.J., Andova, S., Verhoeff, T.: Applying model transformation and event-b for specifying an industrial DSL. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation, pp. 41–50 (2013)
53. Tikhonova, U., Willemse, T.: Designing and describing QVTo model transformations. In: ICSOFT-EA 2015—Proceedings of the 10th International Conference on Software Engineering and Applications, Colmar, Alsace, France, pp. 401–406 (2015)
54. van Amstel, M., van den Brand, M., Engelen, L.: An exercise in iterative domain-specific language design. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), IWPSE-EVOL '10, pp. 48–57. ACM, New York (2010)
55. Vergu, V. A., Neron, P., Visser, E.: DynSem: a DSL for dynamic semantics specification. In: 26th International Conference on Rewriting Techniques and Applications, RTA, pp. 365–378 (2015)
56. Watt, D. A., Muffy, T.: Programming language syntax and semantics. Prentice Hall International Series in Computer Science (1991)

Ulyana Tikhonova is finalizing her Ph.D. study at the Model-Driven Software Engineering group of Eindhoven University of Technology. Her research project was carried out in close collaboration with software engineers of ASML who develop and apply DSLs for controlling lithography machines. Before joining TU/e Ulyana graduated from St. Petersburg State Polytechnical University and had been working in a number of companies as a software developer. Her research area is software engineering, with the focus on model-driven engineering, domain-specific languages, and human-oriented formal methods. Ulyana is motivated to transfer research results from the academic world to the everyday practice of software engineers working in industry.