DOI 10.1007/s00165-016-0359-1

Formal Aspects of Computing (2016) 28: 181–206

Formal Aspects of Computing

Formalizing and testing the consistency of DSL transformations

Sarmen Keshishzadeh[1] and Arjan J. Mooij[2]

[1] Department of Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[2] Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

Abstract. A domain specific language (DSL) focuses on the essential concepts in a specific problem domain, and abstracts from low-level implementation details. The development of DSLs usually centers around the meta-model, grammar and code generator, possibly extended with transformations to analysis models. Typically, little attention is given to the formal semantics of the language, whereas this is essential for reasoning about DSL models, and for assessing the correctness of the generated code and analysis models. We argue that the semantics of a DSL should be defined explicitly and independently of any code generator, to avoid all kinds of complexities from low-level implementation details. As the generated analysis models must reflect some of these implementation details, we propose to formalize them separately. To assess the correctness and consistency of the generated code and analysis models in a practical way, we use conformance testing. We extensively illustrate this general approach using specific formalizations for an industrial DSL on collision prevention. We do not aim for a generic semantic model for any DSL, but this specific DSL indicates the potential of a modular semantics to facilitate reuse among DSLs.

Keywords: Domain specific language (DSL), Semantics, Code generation, Conformance testing

1. Introduction

Well-known programming languages such as C++ and Java are general purpose languages (GPLs) that are universally applicable. On the other hand, domain specific languages (DSLs) [DKV00, Voe13] focus on a specific class of related systems. By trading generality for expressiveness in a limited domain, DSLs offer substantial gains in ease of use compared with GPLs in their domain of application [MHS05]. Modern implementation technologies like EMF [SBP+08] and Xtext [Xte14] seem to boost the applicability of DSLs, as witnessed by reports like [NCB+12, VLH+13]. Figure 1 depicts the typical use of DSLs: a system specification is first formalized using a DSL model, which is afterwards used to generate implementation code and analysis models. The development of a DSL is based on the following separation of domain content [Voe13]:

- *Language* providing an abstraction for describing the variability in a specific class of systems.
- *Target platform* providing a technical environment in which the class of systems must be run.
- *Code generator* transforming a specification expressed in the language into code for the target platform.

Correspondence and offprint requests to: S. Keshishzadeh, E-mail: s.keshishzadeh@tue.nl

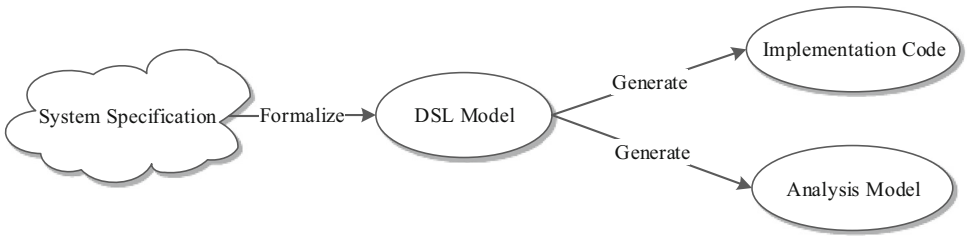


Fig. 1. Domain specific language and transformations

The language component is based on the essential concepts of a specific domain, and abstracts from low-level implementation details. The development of DSLs usually centers around defining the syntax of the language component (in terms of a meta-model or grammar depending on the tools being used) and a code generator, possibly extended with transformations to analysis models. Typically, little attention is given to the formal semantics of the language, and to the implementation details introduced by a particular code generator that are also relevant for some of the analysis models.

Contribution In this article we focus on the significance of developing a formal semantics for a DSL and the impact of some low-level implementation details on the generated artifacts. We propose practical solutions for three questions:

1. What is the precise meaning of the language elements?
2. How to validate the correctness of each generated artifact?
3. How to validate the consistency between generated artifacts?

The first question is about the semantics of the DSL: "It is great that DSLs focus on domain concepts, but what do we model exactly?" In the literature, many approaches can be found for giving semantics to a language [ABE11, SWR+12]. In practice, the semantics of a DSL is usually defined implicitly by implementing a code generator. As one would anyhow implement a code generator, such an implicit semantics comes for free. However, this semantics requires understanding the semantics of the target language and the low-level implementation details introduced by the code generator. This contradicts the purpose of using DSLs. To avoid such a low-level description, we describe the semantics independently of code generators.

The second question is about the correctness of each generated artifact: "Generating implementation code and analysis models is valuable, but how do we know that these artifacts are any good?" Checking the correctness of generated artifacts consists of two parts: whether they are syntactically correct, and whether they conform to the semantics of the DSL. Available tools for GPLs allow us to check for syntactic correctness, so we focus on the second aspect. Various authors [ABE12, EhE08] have argued that the correctness of transformations should be proved. Proving the correctness of transformations is a rigorous approach for gaining confidence in the generated artifacts. However, experience reports such as [Ler09] suggest that for a realistic transformation this approach can be very costly in terms of time and the required expertise. Thus, proving transformations should only be considered for well-established languages and transformations. On the other hand, for a young DSL, code generators are improved regularly, and hence proving their correctness may not be effective. In this setting, lightweight methods that can detect mistakes in the generated artifacts are valuable. We use conformance testing and derive test cases from the formal semantics.

The third question is about the consistency between generated code and analysis models: "Generating all kinds of models is useful for analyzing different aspects of a DSL model, but how do we know that the analysis results are valid for the generated code?" A DSL provides an abstraction that is aligned with the way domain experts reason about problems of the domain. However, a code generator produces implementations that are at a lower level of abstraction than DSL models. In addition to implementing the semantics, the generated code contains implementation details that make the code executable on the target platform. Such implementation details may

have a major impact on various aspects of the generated implementation, e.g., the run-time performance. When performance analysis models are generated from DSL models, the generated models are only useful if they follow the relevant implementation details that were introduced by a specific code generator. As the implementation details are bound to a specific code generator, we formalize them independently of the semantics. We use them to validate the implementation code and impacted analysis models.

We extensively illustrate this general approach using specific formalizations for an industrial DSL on collision prevention. We do not aim for a generic semantic model for any DSL, but this specific DSL indicates the potential of a modular semantics to facilitate reuse among DSLs. This article extends our earlier work presented in [KeM14] by assessing the correctness of analysis models (mentioned as future work in [KeM14]); the third question was not addressed at all in [KeM14].

Overview Our general approach to addressing the questions of Sect. 1 is discussed in Sect. 2. In Sect. 3 we introduce an industrial component and informally describe a DSL for specifying its behavior. The syntax of our industrial DSL is described formally in Sect. 4. In Sect. 5 we focus on formalizing the semantics of the DSL in a modular way. In Sect. 6 we formally describe the implementation details introduced by a code generator and their impact on the computations performed in a DSL model. Sections 5 and 6 form the basis of the conformance testing as discussed in Sect. 7. In Sects. 8 and 9 we provide details about the way we applied our approach in practice and the results obtained from the formalization and testing. In Sect. 10 we argue that studying formal semantics in a modular way can improve reusability between languages. In Sect. 11 we discuss related work. Section 12 contains some concluding remarks and suggestions for future research.

2. General approach

Transformations from a DSL to implementation code and analysis models are very valuable in industrial practice. However, the development of such transformations requires a deep understanding of the DSL, the target language, and the target platform. Moreover, the transformations should correctly realize the semantics of the DSL. As reported in [BrG13], such software engineering tasks are very error prone, and adding redundancy is an effective way to decrease the rate of mistakes in these tasks. In this section we describe the steps we have taken to address the questions from Sect. 1. In particular, reviewing and testing are the redundant mechanisms that we introduce to validate the generated artifacts for a DSL.

2.1. Semantics versus implementation details

A DSL is defined in terms of the essential concepts in a domain, and it abstracts from implementation details. A formal semantics for the DSL gives a precise meaning to the language constructs and it is not concerned with the concrete way that the semantics will be realized and executed on the target platform. A code generator, on the other hand, is a model transformation that realizes the semantics and introduces additional implementation details that are specific for a target platform.

Such implementation details include implementation decisions (such as internal interfaces within the software design, internal data structures and computation strategies), but they can also follow directly from the target platform (such as the used frameworks and technical details of the prescribed interfaces). In addition, there may be all kinds of target language specifics, coding standards, monitoring facilities, etc. Such implementation details are needed, but they clutter the view on the core semantics; so we prefer a formal semantics independent of code generators.

Some implementation details, however, have a major impact on certain aspects of the generated code, such as the run-time performance. Since generated analysis models are used to study various aspects of the generated code, they should be aligned with both the semantics and the relevant implementation details. In our approach, we explicitly distinguish the semantics and some of the relevant implementation details of the generated code. We formalize them separately, and use them to check the consistency of the various generated artifacts.

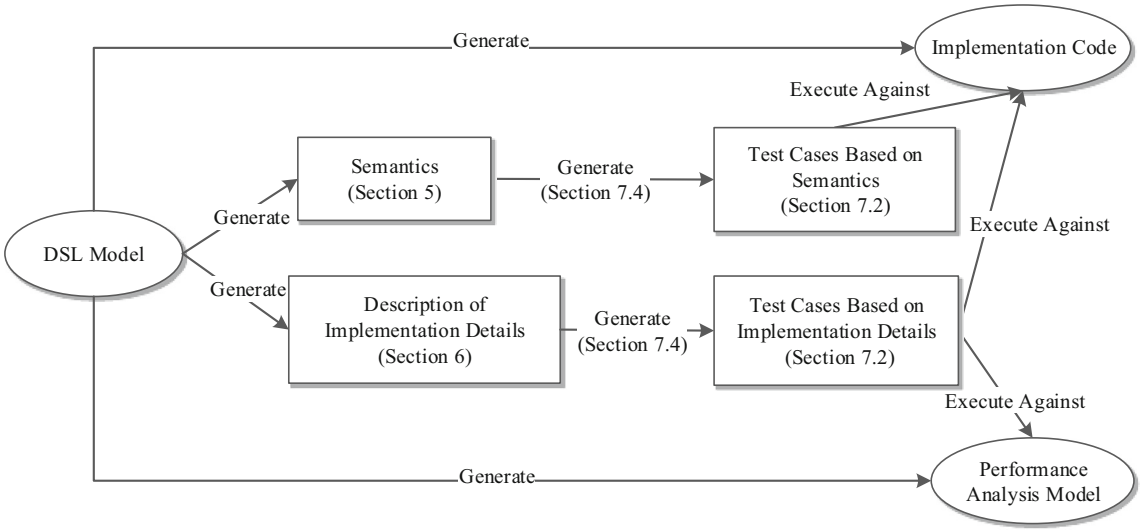


Fig. 2. Overview of the approach

Formalizing the semantics and the implementation details typically triggers discussions among the language designers and the developers of the model transformations. As a result, the developers have to review the choices they have made and make their assumptions explicit. This reviewing mechanism adds a level of redundancy; it can reveal subtle differences in interpretation of the semantics and the implementation details that could otherwise remain undetected. It may also be useful for end-users to be aware of some implementation details, for example, because slightly rewriting a DSL model may have an impact on the run-time performance.

2.2. Conformance testing

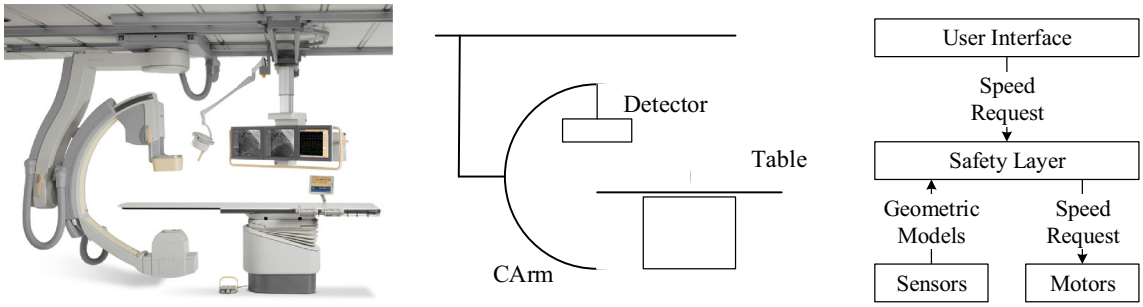
Model transformations generate artifacts that are typically expressed in different modeling and programming languages. We consider generated artifacts that are executable, and treat them as black-boxes that interact with their environment via their external interfaces. To validate their consistency, we use conformance testing. Testing adds a level of redundancy to the approach of Fig. 1 and it can detect inconsistencies between the artifacts.

To apply conformance testing, we need to define a formal relation between DSL models and generated artifacts. We follow the approach taken by several testing theories [Gau95, Tre08]. We abstract from the internal structure of the generated artifacts, and assume that the behavior of each generated artifact can be described by a formal model. This assumption is called the testing hypothesis; it only suggests the existence of a model in a particular formalism and does not imply that the formal model of an artifact is known.

Overview of the approach Our approach for validating the consistency of the generated artifacts is based on the following steps:

1. formalize the semantics of the DSL (Sect. 5) and the relevant implementation details (Sect. 6);
2. define conformance relations based on these formalizations (Sect. 7.2);
3. generate test cases based on the conformance relations (Sect. 7.4);

Figure 2 summarizes these steps for our case study, where the DSL models are used to generate implementation code and performance analysis models. Our approach gives a better understanding of the semantics of the DSL and formally describes the implementation details that are introduced by the code generator and that are relevant for analysis models. Moreover, we gain more confidence in the validity and consistency of the generated artifacts.



(a) Interventional X-ray **(b)** Geometry **(c)** Architecture

Fig. 3. Industrial case

3. Industrial case study

In this section, we describe an industrial component that we use as a running example throughout the article. First, we briefly describe the target platform (Sect. 3.1). Then, we give an informal introduction to a DSL that we use for specifying the behavior of the component (Sect. 3.2). Finally, some challenges regarding the DSL and its artifacts are discussed (Sect. 3.3).

3.1. Platform

Philips Healthcare produces interventional X-ray scanners (Fig. 3a) which are used to perform minimally invasive cardiac, vascular, and neurological medical procedures. These systems consist of several moving objects as sketched in Fig. 3b. For example, the Table can be moved horizontally, the Detector can be moved vertically, and the CArm can be rotated around its center.

The software architecture of these systems includes a dedicated safety layer to prevent collisions between these heavy objects; see Fig. 3c. User speed requests for object movements should pass the safety layer. The safety layer receives the sensor data as so-called 'geometric models', which store the shortest distance between each pair of objects. This data is used for making decisions about user requests and determining the speed requests that should be applied to the motors.

3.2. Prototype DSL

In collaboration with Philips Healthcare, we have developed a prototype DSL [MHA14] to describe the behavior of the safety layer. This DSL focuses on the description of collision prevention rules. Figure 4 depicts a model in the collision prevention DSL.[1] We call this language CPDSL throughout the article.

A CPDSL model starts with declaring the physical objects in the system. For instance, the object declarations in Fig. 4 corresponds to the system depicted in Fig. 3b with three objects, namely, Table, CArm and Detector. The shapes of the objects are not specified in the DSL. A CPDSL model also declares the geometric models that store the sensor data. For example, the declarations of Fig. 4 imply that the sensor data is stored in models with the name Actuals and LookAhead.

The behavior of the safety layer is described in terms of restrictions in the DSL. Each restriction has a name and consists of an activation condition, a deactivation condition, and a set of effects. The example of Fig. 4 consists of two restrictions, namely, `ApproachingTableAndCArm` and `ApproachingTableAndDetector`. The activation and deactivation conditions of a restriction specify when the restriction is active. As a syntactic shorthand, the activation or deactivation conditions can be omitted from a restriction; in such cases the omitted condition is assumed to be true. The effects of a restriction describe speed limitations for specific object movements and are only applicable when the restriction is active.

[1] For confidentiality reasons, numbers and details have been changed in this example.

```
// --- Context Declarations -------
object Table
object CArm
object Detector

model Actuals
model LookAhead

// --- Restrictions -------
restriction ApproachingTableAndCArm
  activation
    Distance[Actuals](Table, CArm) < 35 mm + 15 cm
  effects
    absolute limit CArm[Rotation]
      at ((Distance[Actuals](Table, CArm) - 35 mm) / 15 cm) * 10 dgps
    absolute limit Table[Translation]
      at Distance[Actuals](Table, CArm) ^ 0.75

restriction ApproachingTableAndDetector
  activation
    Distance[LookAhead](Table, Detector) < 35 mm + 15 cm
 && Distance[LookAhead](Table, Detector) <
                                  Distance[Actuals](Table, Detector)
  deactivation
    Distance[LookAhead](Table, Detector) > 35 mm + 20 cm
  effects
    relative limit Detector[Translation]
      at ((Distance[LookAhead](Table, Detector) - 35 mm) / 15 cm)
```

Fig. 4. Snapshot of a CPDSL model

3.3. Challenge

By giving several demonstrations of this DSL, we have observed that most of its features are understood quickly. Some advanced features (such as when both the activation and deactivation conditions hold; see Sect. 5.3), however, always need quite a bit of explanation. Detailed discussions about the DSL have revealed that some features can be interpreted in subtly different ways. These observations motivate the need for a precise description of the DSL semantics.

For this collision prevention DSL, we have developed multiple model transformations, namely, for early validation (such as verification of safety properties [KMM13] and performance analysis [BRM+13]), and the generation and run-time monitoring of implementation code [MHA13]. Although generating implementation code was valuable in practice and analysis models were useful for reasoning about CPDSL models, we were getting increasingly concerned about how to determine that the corresponding transformations are consistent.

In this article we focus on the following two artifacts generated from the collision prevention DSL:

- implementation code [MHA13] in the general-purpose programming language C++;
- analysis model for performance [BRM+13] in the lightweight modeling language POOSL [TFG+07].

The implementation code implements the full semantics of the DSL, whereas the POOSL model only considers the parts relevant for performance. In particular, the POOSL model focuses on computing time-consuming operations and does not actually compute speed requests that should be applied to the motors.

From a functional point-of-view, the geometric models store the shortest distances between each pair of objects. Figure 4 illustrates that activation conditions, deactivation conditions, and effects can refer to the distance values stored in the geometric models. As mentioned in [BRM+13], computing these distance values are by far the most time-consuming operations in a CPDSL model. The code generator from [MHA13] aims to reduce the number of performed distance queries in the generated implementations while still adhering to the semantics of the DSL. Informally, the code generator introduces the following implementation details in the generated code:

D1 the activation and deactivation conditions are evaluated from left to right using lazy evaluation.

D2 the effects of a restriction are only computed if the restriction is active.

These implementation details are specific to the code generator of [MHA13]. Thus, we do not consider them as part of the semantics of the DSL. However, D1 and D2 are relevant for validating the generated performance

analysis models. Additionally, it may also be useful for end-users to be aware of some of these implementation details, for example, because rewriting a CPDSL model in a different way may have an impact on the run-time performance, although the performed functionality remains the same. For instance, a consequence of D1 is that swapping the two operands of a conjunction or disjunction may influence the performance.

4. Syntax

A DSL model describes the behavior of a component in terms of concepts that are essential in its respective domain. As a first step in understanding such a model, we focus on its underlying syntax. This is common practice when developing DSLs. Studying the syntax of a DSL gives us consensus about the level of abstraction in DSL models, the fundamental concepts of the domain, and the relationships between them.

In this section we first introduce the domain concepts that appear in the abstract syntax of the collision prevention DSL and that are essential for describing the internal logic of the safety layer. For each domain concept, we refer to the example of Fig. 4 to show its concrete representation in the language. At some places we provide parts of the grammar. After introducing the domain concepts, we describe the structure of a CPDSL model.

Object, geometric model A CPDSL model starts with declaring a set of objects and a set of geometric models; see the objects and geometric models declared by `object` and `model` in Fig. 4. We use the notations *Obj* and *Mod* to denote the set of declared objects and geometric models for a given CPDSL model, respectively. For example, for the CPDSL model of Fig. 4 we have $Obj = \{Table, CArm, Detector\}$ and $Mod = \{Actuals, LookAhead\}$.

Each geometric model $mod \in Mod$ consists of the distances between each pair of objects. In the CPDSL syntax, $Distance[mod](obj_1, obj_2)$ denotes the distance between objects obj_1 and obj_2 stored in geometric model mod.

For a given CPDSL model, we assume the set of declared objects *Obj* and the set of declared geometric models *Mod* as context and do not mention them as subscripts in the rest of the formalization.

Expression, condition In the CPDSL syntax, expressions and conditions are defined by the following grammars:

$$E ::= rn \mid Distance[mod](obj_1, obj_2) \mid E \ \textbf{arithop} \ E \mid \textbf{unaryop} \ E. \tag{1}$$

$$C ::= false \mid true \mid \neg C \mid C \vee C \mid C \wedge C \mid E \ \textbf{compop} \ E. \tag{2}$$

where $\textbf{arithop} \in \{+, -, *, /, mod, \hat{}, max, min\}$, $\textbf{unaryop} \in \{+, -\}$, $\textbf{compop} \in \{==, !=, \geq, >, \leq, <\}$, and $rn \in \mathbb{R}$, $mod \in Mod$, $obj_1, obj_2 \in Obj$. We use the notations *ES* and *CS* to denote the sets of all possible expressions and conditions constructed by grammar (1) and (2), respectively.

The constant rn in an expression can be annotated by measurement units (e.g., `cm`, `dgps`); we assume that all constants are transformed to a default unit and do not consider this any further.

Effect There are two types of effects, namely, absolute and relative speed limits. We use the data type *LimType* to describe the type of effects:

$$LimType := \{Abs, Rel\}.$$

Effects have an impact on a specific object and a specific type of movements, namely, rotation or translation. We use the data type *MovType* to describe the movement types:

$$MovType := \{Rotation, Translation\}.$$

Note that the *LimType* and *MovType* are fixed sets that are not specified in the CPDSL model.

The notation *Eff* denotes the set of all possible effects for *Obj* and *Mod*. An effect is a tuple $(lt, om, e) \in Eff$ where:

- $lt \in LimType$ is a limit type;
- $om \in \mathcal{P}(Obj \times MovType)$ is a finite set of object movements;

- $e \in ES$ is an expression constructed by grammar (1).

For a set A, the powerset notation $\mathcal{P}(A)$ denotes the set of all subsets of A.

Restriction From Fig. 4, one can see that each restriction consists of three parts. We use *Restr* to denote the set of all possible restrictions for *Obj* and *Mod*. A restriction is a tuple $(act, deact, eff) \in Restr$ where:

- $act \in CS$ is an activation condition constructed by grammar (2);
- $deact \in CS$ is a deactivation condition constructed by grammar (2);
- $eff \in \mathcal{P}(Eff)$ is a finite set of effects over the sets *Obj* and *Mod*.

Finally, the general structure of a CPDSL model is as follows:

CPDSL model A CPDSL model is a tuple $DM = \langle Obj, Mod, R \rangle$ where:

- *Obj* is a finite set of objects;
- *Mod* is a finite set of geometric models;
- $R \in \mathcal{P}(Restr)$ is a finite set of restrictions over the sets *Obj* and *Mod*.

5. Semantics

Studying the syntax of a DSL gives insight into the concepts of the respective domain. In practice, the meaning of the domain concepts used in a DSL is defined implicitly and in terms of a transformation to a programming language. Such a low-level description of the semantics is entangled with details of the target language. To avoid the complexities of such a low-level description, we specify the semantics independently of code generators. We also advocate the use of a modular approach for formalizing the semantics; see the formalized modules in Sects. 5.3 and 5.4. Section 10 contains further discussions in this direction. Additionally, in Sect. 7 we use the formal semantics for validating the semantic correctness of the generated artifacts.

Models in the collision prevention DSL determine how the speed request to the motors is computed given a speed request from the user and geometric models from the sensors; see Fig. 3c. To understand such a model, we first focus on the external interfaces of the component described by the DSL (Sect. 5.1). We also discuss the way two basic types of constructs, namely, expressions and conditions, are interpreted in the language (Sect. 5.2). Afterwards, we split the semantics of the collision prevention DSL in two modules. The first module is a symbolic transition system that determines the active restrictions (Sect. 5.3). The second module is a set of functions that compute the output speed requests (Sect. 5.4). Afterwards, we discuss the relation between these modules and introduce the notion of trace (Sect. 5.5).

5.1. External interfaces

The external interfaces of the safety layer in the architecture of Fig. 3c use two concepts: speed request and geometric models. In what follows, we precisely describe the meaning of these domain concepts. We use the set of objects and geometric models declared in a CPDSL model to define these concepts.

Each speed request contains for each object and movement type a 3D vector. We use the data type *SpReq* to describe speed requests:

$$SpReq := Obj \times MovType \rightarrow \mathbb{R}^3.$$

Note that the DSL specifies speed restrictions based on the geometric models only, and hence there is no language construct for speed requests.

In Sect. 4 we discussed geometric models and introduced the syntax for referring to the distance values they contain. We interpret each geometric model $mod \in Mod$ as a distance function $[\![mod]\!] : Obj \times Obj \rightarrow \mathbb{R}_0^+$ such that (for all $obj_1, obj_2 \in Obj$):

- $[\![mod]\!](obj_1, obj_1) = 0$;
- $[\![mod]\!](obj_1, obj_2) = [\![mod]\!](obj_2, obj_1)$.

We use *Dist* to denote the set of distance functions for *Obj*. As depicted in Fig. 3b, the objects have shapes. Each distance value refers to the shortest distance between a pair of objects, i.e., the distance between the closest pair of points from these objects. The triangle inequality does not apply in terms of the 3D objects, i.e., $[\![mod]\!](obj_1, obj_2) + [\![mod]\!](obj_2, obj_3) \geq [\![mod]\!](obj_1, obj_3)$ is not necessarily valid for $obj_1, obj_2, obj_3 \in Obj$ and $mod \in Mod$. By extending this interpretation to the set of geometric models, we describe the set *Mod* as a function of type $GeoVal := Mod \rightarrow Dist$, i.e., each geometric model in *Mod* is interpreted as a distance function.

Communications between the safety layer and its interfacing components take place at fixed intervals of time; see Fig. 3c. Starting from the initial state, in each round of execution the safety layer collects the current speed requests from the user and geometric models from the sensors and computes output speed requests that are sent to the motors; we formalize this notion of execution for the safety layer in Sect. 5.5.

5.2. Expressions and conditions

In Sect. 4 we discussed the syntax of expressions and conditions in the collision prevention DSL; we also introduced their use for specifying effects and restrictions. In what follows, we formalize the semantics of expressions and conditions. This will be a basis for Sect. 5.3 where we discuss the semantics of effects and restrictions.

We interpret an expression $e \in ES$ constructed by grammar (1) as a function $[\![e]\!] : GeoVal \rightarrow \mathbb{R}$. This interpretation is defined inductively as follows ($g \in GeoVal$ is the current value of geometric models):

- $[\![rn]\!](g) = rn$
- $[\![Distance[mod](obj_1, obj_2)]\!](g) = g(mod)(obj_1, obj_2)$
- $[\![e_1 \textbf{ arithop } e_2]\!](g) = [\![e_1]\!](g) \textbf{ arithop } [\![e_2]\!](g)$
- $[\![\textbf{unaryop } e]\!](g) = \textbf{unaryop } [\![e]\!](g)$

where $rn \in \mathbb{R}$, $mod \in Mod$, $obj_1, obj_2 \in Obj$ and $e, e_1, e_2 \in ES$ are expressions constructed by grammar (1).

Similarly, we interpret a condition $c \in CS$ constructed by grammar (2) as a function $[\![c]\!] : GeoVal \rightarrow Bool$. This semantics is defined inductively as follows ($g \in GeoVal$ is the current value of geometric models):

- $[\![false]\!](g) = false$
- $[\![true]\!](g) = true$
- $[\![\neg c]\!](g) = \neg[\![c]\!](g)$
- $[\![c_1 \vee c_2]\!](g) = [\![c_1]\!](g) \vee [\![c_2]\!](g)$
- $[\![c_1 \wedge c_2]\!](g) = [\![c_1]\!](g) \wedge [\![c_2]\!](g)$
- $[\![e_1 \textbf{ compop } e_2]\!](g) = [\![e_1]\!](g) \textbf{ compop } [\![e_2]\!](g)$

where $c, c_1, c_2 \in CS$ are conditions constructed by grammar (2) and $e_1, e_2 \in ES$ are expressions constructed by grammar (1).

5.3. State-based semantic module for determining active restrictions

We start our formalization of the DSL semantics by introducing a semantic module that determines the active restrictions. This module considers only one of the inputs to the safety layer, namely, the geometric models. We first consider a single restriction and afterwards extend it to a set of restrictions.

A single restriction $r = (act_r, deact_r, eff_r)$ can be in two states: active or passive. Initially, each restriction r is passive. The current state of r depends on its previous state and the current value of the geometric models. Let $p \in Bool$ be the previous state of r, and $g \in GeoVal$ the current geometric models. The current state of r is active if:

- the activation condition evaluates to true (i.e., $[\![act_r]\!](g)$), or
- it was active in the previous state and the deactivation condition evaluates to false (i.e., $p \wedge \neg[\![deact_r]\!](g)$).

It follows from the discussion above that if both the activation and the deactivation conditions evaluate to true, the current state is active.

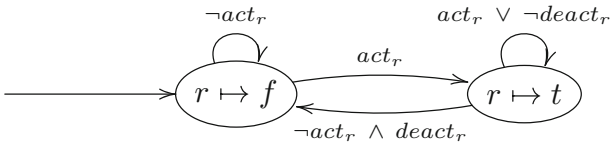


Fig. 5. A symbolic transition system for $R = \{r\}$

Figure 5 illustrates these rules as a symbolic transition system; state labels $r \mapsto f$ and $r \mapsto t$ denote "r is passive" and "r is active", respectively. The transition relation can be formalized by the following propositional formula:

$$CurrAct_r := act_r \vee (pb \wedge \neg deact_r)$$

where pb symbolically represent the previous state of r. In Fig. 5, the initial state is depicted by an incoming arrow. It is assumed that r is passive in the initial state.

The idea of Fig. 5 for one restriction can be extended to the set of restrictions R in $DM = \langle Obj, Mod, R\rangle$. This induces an augmented symbolic transition system. Each state of this transition system is a function of type $R \to Bool$ which describes the active/passive restrictions. Similar to Fig. 5, the initial state is $q : R \to Bool$ such that $q(r) = false$ for $r \in R$, i.e., independent of the CPDSL model, we choose all restrictions to be passive in the initial state. Each transition is defined based on $CurrAct_r$ for all $r \in R$ and symbolically represents the geometric models values that enable the move from the source state to the target state.

Figure 6 depicts the symbolic transition system for $R = \{r_1, r_2\}$ where $r_1 = (act_1, deact_1, eff_1)$ and $r_2 = (act_2, deact_2, eff_2)$. Each state is represented by a function that specifies the state of r_1 and r_2. Propositional formulas c_i for $1 \le i \le 16$ are specified based on $CurrAct_{r_1}$ and $CurrAct_{r_2}$.

Formally, the set of restrictions R in $DM = \langle Obj, Mod, R\rangle$ induces an augmented symbolic transition system $ASTS_R = \langle Q, q, T\rangle$ such that:

- $Q := R \to Bool$ is the set of states;
- $q \in Q$ is the initial state and $q(r) = false$ for all $r \in R$;
- $T \subseteq Q \times CS \times Q$ is a transition relation such that:

$$T = \{(q_1, c, q_2) \mid c = \bigwedge_{r \in R} ((\neg q_2(r) \vee CurrAct_r[pb := q_1(r)]) \wedge (\neg CurrAct_r[pb := q_1(r)] \vee q_2(r)))\}$$

where $CurrAct_r[pb := q_1(r)]$ replaces the symbol pb in $CurrAct_r$ with the state of restriction r in the source, i.e., $q_1(r)$. Note that condition c is equivalent to $\bigwedge_{r \in R}(q_2(r) \Leftrightarrow CurrAct_r[pb := q_1(r)])$; we have rewritten c to make it derivable from grammar (2).

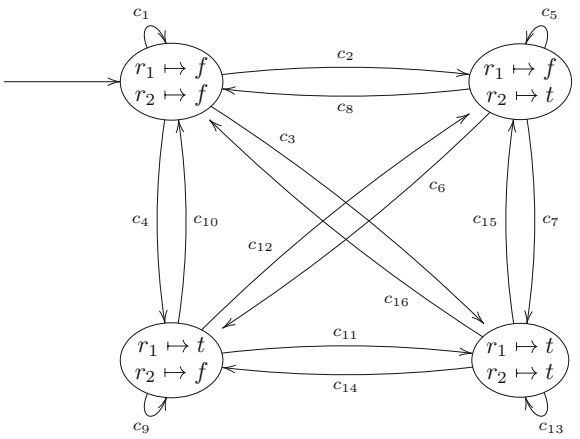
In the construction of the semantic model, we choose each restriction $r \in R$ to be passive in the initial state. Note that unreachable states and unsatisfiable transitions are not discarded by this description.

The transition system of DM is deterministic, i.e., the outgoing transitions of each state are labeled with disjoint conditions. Moreover, for each state $q_1 \in Q$ the set T considers all combinations of active restrictions that are conceivable from q_1. Thus, for each q_1 and $g \in GeoVal$ there is exactly one transition (q_1, c, q_2) such that $[\![c]\!](g)$ is valid.

5.4. Function-based semantic module for computing output speed requests

Output speed request computation is the second module of the semantics. This module considers both inputs of the safety layer, namely, the geometric models and the speed requests. In addition, it uses the state of the restrictions determined by the first module.

Speed requests from the user are 3D vectors. To determine the output speed vectors, we first compute the length of the vector that should be applied to the motors. Afterwards, we determine the direction and compute the output vector. We first describe the effect of a single restriction on output computation and afterwards extend it to a set of restrictions.



$c_1 := \neg act_1 \wedge \neg act_2$
$c_2 := \neg act_1 \wedge act_2$
$c_3 := act_1 \wedge act_2$
$c_4 := act_1 \wedge \neg act_2$
$c_5 := \neg act_1 \wedge (act_2 \vee \neg deact_2)$
$c_6 := act_1 \wedge (\neg act_2 \wedge deact_2)$
$c_7 := act_1 \wedge (act_2 \vee \neg deact_2)$
$c_8 := \neg act_1 \wedge (\neg act_2 \wedge deact_2)$
$c_9 := (act_1 \vee \neg deact_1) \wedge \neg act_2$
$c_{10} := (\neg act_1 \wedge deact_1) \wedge \neg act_2$
$c_{11} := (act_1 \vee \neg deact_1) \wedge act_2$
$c_{12} := (\neg act_1 \wedge deact_1) \wedge act_2$
$c_{13} := (act_1 \vee \neg deact_1) \wedge (act_2 \vee \neg deact_2)$
$c_{14} := (act_1 \vee \neg deact_1) \wedge (\neg act_2 \wedge deact_2)$
$c_{15} := (\neg act_1 \wedge deact_1) \wedge (act_2 \vee \neg deact_2)$
$c_{16} := (\neg act_1 \wedge deact_1) \wedge (\neg act_2 \wedge deact_2)$

Fig. 6. A symbolic transition system for $R = \{r_1, r_2\}$

The effects of a restriction $r = (act_r, deact_r, \mathit{eff}_r)$ is only considered in output computation when r is active. Thus, we describe the influence of eff_r in the context of $b \in \mathit{Bool}$ which denotes the current state of r. When r is active, eff_r specifies speed limitations for specific object movements. The function $[\![\mathit{eff}_r]\!]_b : \mathit{Obj} \times \mathit{MovType} \times \mathit{LimType} \times \mathit{GeoVal} \to \mathcal{P}(\mathbb{R})$ specifies the active limits of r for each object movement, limit type, and the current geometric models value:

$$[\![\mathit{eff}_r]\!]_b(obj, m, \ell, g) = \{v \mid b \wedge \exists om \in \mathcal{P}(\mathit{Obj} \times \mathit{MovType}), e \in \mathit{ES}. \\ (\ell, om, e) \in \mathit{eff}_r \wedge (obj, m) \in om \wedge [\![e]\!](g) = v\}.$$

When r is active, for each object movement the most restrictive relative and absolute limits are considered as the effects of r. Formally, we interpret r as a partial function $[\![r]\!]_b : \mathit{Obj} \times \mathit{MovType} \times \mathit{LimType} \times \mathit{GeoVal} \nrightarrow \mathbb{R}$ such that:

$$[\![r]\!]_b(obj, m, \ell, g) = \min([\![\mathit{eff}_r]\!]_b(obj, m, \ell, g)) \quad \text{if } [\![\mathit{eff}_r]\!]_b(obj, m, \ell, g) \neq \emptyset$$

Now we discuss the effects of the set of restrictions R. For each object movement, multiple active restrictions from R can specify absolute and relative limits. The most restrictive limits are considered for each object movement, i.e., the minimum of the absolute limits and the minimum of the relative limits. Formally, the set of restrictions R is interpreted in the context of $B : R \to \mathit{Bool}$, which denotes the current state of the restrictions. The set R is interpreted as $[\![R]\!]_B : \mathit{Obj} \times \mathit{MovType} \times \mathit{LimType} \times \mathit{GeoVal} \to \overline{\mathbb{R}}$. The notation $\overline{\mathbb{R}}$ denotes the two point compactification of real numbers: $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$.

$$[\![R]\!]_B(obj, m, \ell, g) = \begin{cases} \min\{[\![r]\!]_{B(r)}(obj, m, \ell, g) \mid r \in R\} & \text{if } (\exists\, r \in R.\, (obj, m, \ell, g) \in dom([\![r]\!]_b)) \\ \mathit{DefLim}(\ell) & \text{if } \neg(\exists\, r \in R.\, (obj, m, \ell, g) \in dom([\![r]\!]_b)) \end{cases}$$

where $dom([\![r]\!]_b)$ extracts the domain of r. If R does not specify a limit, an appropriate default value specified by $\mathit{DefLim} : \mathit{LimType} \to \overline{\mathbb{R}}$ is returned, i.e., $\mathit{DefLim}(Abs) = +\infty$ and $\mathit{DefLim}(Rel) = 1$. An absolute limit specifies a maximum speed that may be requested to the motors and a relative limit indicates the maximum percentage of the requested speed that may be applied to the motors. Thus, we choose $+\infty$ and 1 as default values for Abs and Rel, respectively.

The effects of R are real numbers, whereas (user/output) speed requests are 3D vectors. The length of the speed requests to the motors are computed based on the most restrictive absolute and relative limits specified by R. The function $[\![\mathit{outSpeed}_R]\!]_B : \mathit{Obj} \times \mathit{MovType} \times \mathit{GeoVal} \times \mathit{SpReq} \to \mathbb{R}$ describes this computation for the set

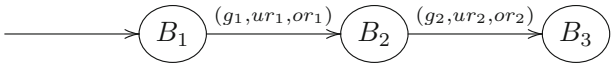


Fig. 7. An example run for $(ASTS_R, output_R)$

of restrictions R in the context of B as follows:

$$[\![outSpeed_R]\!]_B(obj, m, g, ur) = \min\{[\![R]\!]_B(obj, m, Abs, g), \\ [\![R]\!]_B(obj, m, Rel, g) \times norm(ur(obj, m))\}$$

where $ur \in SpReq$ is the current speed request from the user and $norm$ is defined as $norm((x, y, z)) = \sqrt{x^2 + y^2 + z^2}$. In the definition above, "min" refers to the definition of the minimum function for $\overline{\mathbb{R}}$. Since the relative limits and norm of vectors are real numbers, the computed speed is a real number (not $-\infty$ or $+\infty$).

Now, we can proceed by computing the 3D speed vectors that should be applied to the motors. We describe this computation by $[\![output_R]\!]_B : GeoVal \times SpReq \to SpReq$ as follows:

$$[\![output_R]\!]_B(g, ur)(obj, m) = \begin{cases} \frac{ur(obj,m)}{norm(ur(obj,m))} \times [\![outSpeed_R]\!]_B(obj, m, g, ur) & \text{if } ur(obj, m) \neq (0, 0, 0) \\ (0, 0, 0) & \text{otherwise.} \end{cases}$$

It follows from the definition that the direction of the output is determined by computing a unit vector from the user request. The norm of the output is determined by the value computed by the CPDSL model, i.e., $outSpeed_R$.

5.5. Semantics of a DSL model

Finally, the semantics of a CPDSL model combines two modules: one that determines the active restrictions and one that computes the output speed requests:

Definition 1 (*Semantics of a model*) Let $DM = \langle Obj, Mod, R\rangle$ be a CPDSL model. The semantics of DM is the pair $(ASTS_R, output_R)$.

In the following definition, we formalize the notion of execution for a CPDSL model and show the way the two modules of the semantics are linked. These definitions are guided by the manner in which the safety layer communicates with its interfacing components (Fig. 3c). The safety layer reads the latest geometric models and the latest speed requests from the user simultaneously at fixed intervals of time, and then produces a speed request to the motors.

Definition 2 [*Trace in* $(ASTS_R, output_R)$] Let $DM = \langle Obj, Mod, R\rangle$ be a CPDSL model, $c \in CS$, $g \in GeoVal$, and $ur, or \in SpReq$. The set of traces for a state $p \in Q$ of $ASTS_R$ is denoted by $Traces_I(p)$ and is the minimal set satisfying:

1. empty trace $\epsilon \in Traces_I(p)$;
2. non-empty trace $(g, ur, or)\sigma \in Traces_I(p)$, if there exists $p' \in Q$ such that $[\![c]\!](g)$ is valid and $(p, c, p') \in T$, $or = [\![output_R]\!]_{p'}(g, ur)$, and $\sigma \in Traces_I(p')$.

The set of traces for $(ASTS_R, output_R)$ is $Traces_I(DM) = Traces_I(q)$ where $q \in Q$ is the initial state of $ASTS_R$.

Based on the definition of $ASTS_R$ (see Sect. 5.3), for a state $p \in Q$ and $g \in GeoVal$, there is exactly one transition (p, c, p') such that $[\![c]\!](g)$ is valid. Thus, each non-empty trace $(g_1, ur_1, or_1) \dots (g_n, ur_n, or_n)$ corresponds to a unique run $q(g_1, ur_1, or_1)q_1 \dots q_{n-1}(g_n, ur_n, or_n)q_n$ in $(ASTS_R, output_R)$. The set of runs for $(ASTS_R, output_R)$ is denoted by $Run_I(DM)$. A run registers a sequence of states and tuples of (g_i, ur_i, or_i) where g_i is the concrete values of geometric models that enable the transition from q_{i-1} to q_i, ur_i is a speed request from the user, and or_i is the computed output speed request. The trace corresponding to a run does not register the encountered states.

Figure 7 depicts a run of $(ASTS_R, output_R)$ where the state labels $B_i : R \to Bool$ describe the active restrictions in three different states. It is assumed that B_1, B_2 and B_3 are distinct functions.

6. Formalizing implementation details

In Sect. 2.1 we distinguished between the DSL abstraction and the implementation details introduced by a code generator. We discussed that implementation details are specific to a code generator but they may be relevant for multiple artifacts generated from the DSL. To have an explicit description of the relevant implementation details, we propose to specify them independently of the semantics. Moreover, we use the formalized implementation details to validate the consistency of the impacted artifacts.

To illustrate the significance of the implementation details, we consider the code generator from [MHA13] for the collision prevention DSL. In Sect. 3.3 we discussed that computing distance values are by far the most time-consuming operations of a CPDSL model. Moreover, we informally described the implementation details introduced by the code generator for minimizing the number of distance queries. In this section, we formalize the impact of the implementation details on time-consuming operations of a CPDSL model in terms of two modules. The first module is again the symbolic transition system from Sect. 5.3. The second module is a set of functions that compute the set of distance queries required for evaluating restrictions (Sect. 6.1). Afterwards, we combine these modules to describe the time-consuming operations of a CPDSL model, and introduce the notion of trace (Sect. 6.2).

6.1. Module for time-consuming operations of restrictions

In this section, we apply the semantics of the DSL and the implementation details described in D1 and D2 (see Sect. 3.3) to mathematically specify the set of distance queries required for evaluating the restrictions of a model DM.

The semantics described in Sect. 5 shows that distance queries are required for computing:

- the set of active restrictions (i.e., computing the current state of $ASTS_R$ as discussed in Sect. 5.3), and
- the effects of active restrictions (i.e., computing $[\![r]\!]_b$ for an active restriction as discussed in Sect. 5.4).

The speed request from the user does not influence the computation of active restrictions and their effects. Moreover, $[\![output_R]\!]_B$ is computed based on the current state and $[\![r]\!]_b$ for all active restrictions $r \in R$ and hence no further distance queries are required to compute the output speed requests.

We first focus on computing the set of distance queries required for evaluating a given expression ($e \in ES$) and condition ($c \in CS$) and then compute the set of distance queries required for evaluating restrictions. In what follows, we use the data type $DistQuery$ to describe distance queries:

$$DistQuery := Mod \times Obj \times Obj.$$

6.1.1. Evaluating expressions and conditions

We consider grammar (1) for expressions (Sect. 4) and specify $ExprQuery\colon ES \to \mathcal{P}(DistQuery)$ to calculate the set of distance queries required for evaluating an expression. The computation rules for $ExprQuery$ are as follows:

1. $ExprQuery(rn) = \emptyset$
2. $ExprQuery(Distance[mod](obj_1, obj_2)) = \{(mod, obj_1, obj_2)\}$
3. $ExprQuery(e_1 \textbf{ arithop } e_2) = ExprQuery(e_1) \cup ExprQuery(e_2)$
4. $ExprQuery(\textbf{unaryop } e) = ExprQuery(e)$

where $rn \in \mathbb{R}$, $e, e_1, e_2 \in ES$, $mod \in Mod$, $obj_1, obj_2 \in Obj$.

The production rules of grammar 2 (Sect. 4) imply that conditions can also refer to the distance values of geometric models. The implementation detail described in D1 (Sect. 3.3) is introduced by the code generator to reduce the number of distance queries required for evaluating conditions. We formalize the impact of D1 using the function $CondQuery\colon CS \times GeoVal \to \mathcal{P}(DistQuery)$ which given the current value of geometric models computes the set of distance queries required for evaluating a condition:

1. $CondQuery(false, g) = CondQuery(true, g) = \emptyset$
2. $CondQuery(\neg c, g) = CondQuery(c, g)$

3. $CondQuery(c_1 \vee c_2, g) = \begin{cases} CondQuery(c_1, g) & \text{if } [\![c_1]\!](g) \\ CondQuery(c_1, g) \cup CondQuery(c_2, g) & \text{if } \neg[\![c_1]\!](g) \end{cases}$

4. $CondQuery(c_1 \wedge c_2, g) = \begin{cases} CondQuery(c_1, g) & \text{if } \neg[\![c_1]\!](g) \\ CondQuery(c_1, g) \cup CondQuery(c_2, g) & \text{if } [\![c_1]\!](g) \end{cases}$

5. $CondQuery(e_1 \textbf{ compop } e_2, g) = ExprQuery(e_1) \cup ExprQuery(e_2)$

where $c, c_1, c_2 \in CS$, $g \in GeoVal$, $e_1, e_2 \in ES$. Rule 3 and 4 show the effect of D1 on the computation of distance queries (i.e., lazy evaluation for $c_1 \vee c_2$ and $c_1 \wedge c_2$).

6.1.2. Evaluating restrictions

Let $r = (act_r, deact_r, eff_r) \in R$ be a restriction. Evaluating r requires:

- determining the current state of r (evaluating $CurrAct_r$ based on the previous state of r and the current geometric models value);
- calculating the effects of r if r is currently active.

In Sect. 6.1.1 we formalized the impact of D1 on evaluating conditions. We use this formalization to describe distance queries required for evaluating active restrictions. Our description should also reflect the impact of D2 on distance computations. The set of distance queries required for evaluating restriction r can be specified by $[\![Query_r]\!]_{ps} : GeoVal \to \mathcal{P}(DistQuery)$ where $ps \in Bool$ is the previous state of r:

$$[\![Query_r]\!]_{ps}(g) = \begin{cases} CondQuery(CurrAct_r[pb := ps], g) \cup & \text{if } [\![CurrAct_r[pb := ps]]\!](g) \\ (\bigcup_{(lt,om,e) \in eff_r} ExprQuery(e)) & \\ CondQuery(CurrAct_r[pb := ps], g) & \text{if } \neg[\![CurrAct_r[pb := ps]]\!](g). \end{cases}$$

The definition of $Query_r$ shows that D2 is applied to avoid additional distance queries for passive restrictions.

Distance computations for r can be extended to the set of restrictions R in a CPDSL model. We describe the set of distance queries required for evaluating R by the function $[\![Query_R]\!]_{PS} : GeoVal \to \mathcal{P}(DistQuery)$ where $PS : R \to Bool$ specifies the previous state of the restrictions.

$$[\![Query_R]\!]_{PS}(g) = \bigcup_{r \in R} [\![Query_r]\!]_{PS(r)}(g)$$

The function $Query_R$ computes the set of distance queries required for evaluating restrictions and hence in each state and for each geometric models value duplicate distance queries are considered only once.

6.2. Time-consuming operations performed by a DSL model

The time-consuming operations performed by a CPDSL model can be described by two modules: one that determines the active restrictions and one that computes the distance queries required for evaluating the restrictions:

Definition 3 (*Distance queries of a* CPDSL *model*) Let $DM = \langle Obj, Mod, R \rangle$ be a CPDSL model. The set of distance queries required for realizing the semantics of DM with respect to the implementation details D1 and D2 introduced by the code generator can be described by the pair $(ASTS_R, Query_R)$.

Definitions 1 and 3 describe two different aspects of $DM = \langle Obj, Mod, R \rangle$. The pair $(ASTS_R, output_R)$ specifies the relation between the inputs and the computed speed requests to the motors, whereas $(ASTS_R, Query_R)$ specifies the relation between the inputs and the required set of distance queries.

Similar to Definition 2, we define the notion of trace for $(ASTS_R, Query_R)$ to formalize the way the modules $ASTS_R$ and $Query_R$ are linked:

Definition 4 [*Trace in* $(ASTS_R, Query_R)$] Let $DM = \langle Obj, Mod, R \rangle$ be a CPDSL model, $c \in CS, g \in GeoVal, ur \in SpReq$ and $dq \in \mathcal{P}(DistQuery)$. The set of traces for a state $p \in Q$ of $ASTS_R$ is denoted by $Traces_P(p)$ and is the minimal set satisfying:

1. empty trace $\epsilon \in \mathit{Traces}_P(p)$;
2. non-empty trace $(g, ur, dq)\sigma \in \mathit{Traces}_P(p)$, if there exists $p' \in Q$ such that $[\![c]\!](g)$ is valid and $(p, c, p') \in T$, $dq = [\![\mathit{Query}_R]\!]_p(g)$, and $\sigma \in \mathit{Traces}_P(p')$.

The set of traces for $(\mathit{ASTS}_R, \mathit{Query}_R)$ is $\mathit{Traces}_P(DM) = \mathit{Traces}_P(q)$ where $q \in Q$ is the initial state of ASTS_R.

For each $p \in Q$ of ASTS_R and $g \in \mathit{GeoVal}$, there is exactly one transition $(p, c, p') \in T$ such that $[\![c]\!](g)$ is valid (see the definition of ASTS_R in Sect. 5.3). Thus, each non-empty trace $(g_1, ur_1, dq_1)\ldots(g_n, ur_n, dq_n)$ corresponds to a unique run $q(g_1, ur_1, dq_1)q_1 \ldots q_{n-1}(g_n, ur_n, dq_n)q_n$ in $(\mathit{ASTS}_R, \mathit{Query}_R)$. The set of runs for $(\mathit{ASTS}_R, \mathit{Query}_R)$ is denoted by $\mathit{Run}_P(DM)$.

Traces from Traces_I and Traces_P differ in the last element they specify in each step; see Definitions 2 and 4. Traces from Traces_I are sequences of (g_i, ur_i, or_i) where or_i is the output speed request computed for inputs g_i, ur_i in step i. On the other hand, traces from Traces_P are sequences of (g_i, ur_i, dq_i) where dq_i is the set of distance queries required for computing the output request in step i.

7. Conformance testing

In this section we define what it means for a generated artifact in Fig. 1 to comply to a CPDSL model DM. Inspired by model-based testing approaches, in Sect. 7.1 we introduce a formalism for describing the behavior of artifacts. In Sect. 7.2 we define the conformance of an artifact to a CPDSL model and formalize a notion of consistency between two artifacts. In Sect. 7.3 we define the notions of test case and test case execution for validating the correctness and consistency of the artifacts. Finally, in Sect. 7.4 we address the test selection problem in our testing approach.

In what follows, we refer to finite sequences over a given set A by $\mathit{seq}(A)$ such that:

$$\mathit{seq}(A) = \{f : \mathbb{N} \nrightarrow A \mid \exists\, n \in \mathbb{N} \cdot \mathit{dom}(f) = \{1, \ldots, n\}\}.$$

We write $[a, b, c]$ to denote the sequence $\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$.

7.1. Testing hypothesis

In our testing approach, an artifact is treated as a black-box that interacts with its environment via its interfaces. To define conformance relations between DSL models (formal models) and generated artifacts (physical entities), we need objects that describe the behavior of the artifacts and are suitable for formal reasoning. Similar to several model-based testing approaches [Gau95, Tre08], we assume that the behavior of an artifact can be described with a particular formalism (testing hypothesis). Testing hypothesis only requires the existence of a formalism for describing the behavior of artifacts and it does not imply that the formal model of an artifact is available.

Let *Obj* be a set of objects and *Mod* be a set of geometric models. We assume that any implementation or performance analysis model of the safety layer can be modeled as $\mathit{artifact} = (\mathit{STS}, \mathit{compute})$. In this model, $\mathit{STS} = \langle Q, q, T\rangle$ is a symbolic transition system where:

- Q is a set of states;
- $q \in Q$ is the initial state;
- $T \subseteq Q \times \mathit{CS} \times Q$ is a transition relation.

The function $[\![\mathit{compute}]\!]_h : \mathit{GeoVal} \times \mathit{SpReq} \to \mathit{Output}$ considers the execution history $h \in \mathit{seq}(\mathit{GeoVal})$, and calculates an output. The artifact is treated as a black-box and the concrete way the computations are performed is not relevant. The output set is $\mathit{Output} = \mathit{SpReq}$ if *artifact* computes speed requests (*artifact* is an implementation) and $\mathit{Output} = \mathcal{P}(\mathit{DistQuery})$ if *artifact* computes sets of distance queries (*artifact* is a performance model).

For *Obj* and *Mod*, the set of all formal models for implementations and performance models of the safety layer are denoted by *Imp* and *Perf*, respectively. We assume that artifacts are deterministic, e.g., for each state, geometric models, and user speed request an implementation $\mathit{imp} \in \mathit{Imp}$ produces exactly one speed request.

For a state $p \in Q$ of $imp = (STS, compute) \in Imp$ the set of traces is denoted by $Traces_I(p)$ and is the minimal set satisfying:

1. empty trace $\epsilon \in Traces_I(p)$
2. non-empty trace $(g, ur, or)\sigma \in Traces_I(p)$, if there exists $p' \in Q$ such that $[\![c]\!](g)$ is valid and $(p, c, p') \in T$, $or = [\![compute]\!]_h(g, ur)$, $\sigma \in Traces_I(p')$ and h is the sequence of geometric models from $(g, ur, or)\sigma$.

The set of traces for $imp = (STS, compute)$ is $Traces_I(imp) = Traces_I(q)$ where $q \in Q$ is the initial state of STS.

The set of traces for a performance model $perf \in Perf$ is defined similarly to traces of $imp \in Imp$. Each non-empty trace of $perf$ registers a sequence of (g, ur, dq) where $dq \in \mathcal{P}(DistQuery)$ is the required set of distance queries. The notation $Traces_P(perf)$ denotes the set of traces for $perf \in Perf$.

7.2. Conformance relations

The formalisms of Sect. 7.1 allow us to define formal relations to describe the compliance of an artifact to a CPDSL model. From Sects. 5 and 6, one can see that a CPDSL model specifies:

- Exactly one output request for any state, geometric models value, and user request.
- Exactly one set of distance queries for any state, geometric models value, and user request.

In many conformance testing theories, an implementation is considered to be compliant to a specification if for any experiment derived from the specification, the implementation produces an output that is foreseen by the specification. In other words, the output produced by the implementation should belong to the set of outputs allowed by the specification.

In our setting, specifications (CPDSL models) compute output speed requests and required distance queries in a deterministic way and hence trace equivalence is a natural way of describing conformance. Thus, an artifact conforms to a CPDSL model if there is no way of telling them apart through observation of finite traces of arbitrary length.

Definition 5 Let $DM = \langle Obj, Mod, R \rangle$ be a CPDSL model:

1. An artifact described by $imp \in Imp$ conforms to DM if and only if $Traces_I(DM) = Traces_I(imp)$.
2. An artifact described by $perf \in Perf$ conforms to DM with respect to the implementation details D1 and D2 if and only if $Traces_P(DM) = Traces_P(perf)$.

Definition 5 allows us to validate implementations and performance models with respect to CPDSL models. To assess the consistency between an implementation and a performance model generated from a CPDSL model, we need to define what it means for these two artifacts to be consistent.

The formalisms of the testing hypothesis show that implementations and performance models differ in their external interfaces, i.e., an implementation computes speed requests, whereas a performance model computes sets of required distance queries. However, the implementation also performs distance queries to compute output speed requests. Hence, it is possible to create an interface for implementations that produces the set of performed distance queries. In other words, with the new interface an implementation can be modeled as $imp \in Perf$, i.e., we can treat the implementation as a black-box that can be described by $(STS, compute)$ where $[\![compute]\!]_h : GeoVal \times SpReq \to \mathcal{P}(DistQuery)$.

In this way, we can relate implementations to CPDSL models using the second conformance relation from Definition 5. An implementation (described by $imp \in Perf$) and a performance model (described by $perf \in Perf$) of DM are consistent if and only if the sets $Traces_P(imp)$ and $Traces_P(perf)$ are equivalent to $Traces_P(DM)$. To establish the consistency between imp and $perf$, the set $Traces_P(DM)$ should be tested on the artifacts.

7.3. Test cases

A test case for an artifact is an experiment where in each step we supply geometric models and a user speed request to the artifact and observe its output. Formally speaking, a test case for $artifact \in Imp \cup Perf$ is an element of $seq(GeoVal \times SpReq \times Output)$ where $Output$ is $SpReq$ or $\mathcal{P}(DistQuery)$.

In addition to the notion of test case, we should provide a mechanism for executing test cases from $seq(GeoVal \times SpReq \times Output)$ on artifacts such that a suitable verdict (**pass** or **fail**) is assigned after test case execution. A test case terminates with **pass** if in each step of the test case the output generated by the artifact corresponds to the expected output. On the other hand, any deviation from the expected outputs leads to **fail**.

Let $t \in seq(GeoVal \times SpReq \times Output)$ be a test case for $artifact = (STS, compute)$. Execution of t on $artifact$ can be described by $Exec : (Imp \cup Perf) \times seq(GeoVal \times SpReq \times Output) \to \{\textbf{pass}, \textbf{fail}\}$ such that:

$$Exec((STS, [\![compute]\!]_h), t) = \begin{cases} \textbf{pass} & \text{if } (t = []) \vee \\ & \quad (t = [(g_1, ur_1, o_1), \ldots, (g_n, ur_n, o_n)] \wedge \\ & \qquad (\forall k.1 \leq k \leq n.[\![compute]\!]_{[g_1,\ldots,g_k]}(g_k, ur_k) = o_k)) \\ \textbf{fail} & \text{otherwise.} \end{cases}$$

From Definition 5, it follows that to establish the conformance of an artifact to DM, the relevant set of traces ($Traces_I(DM)$ or $Traces_P(DM)$) should be executed on the artifact. Similarly, consistency checking between an implementation and performance model for DM requires executing $Traces_P(DM)$ on the artifacts.

7.4. Coverage criteria

In Sect. 7.3 we discussed about testing conformance of an artifact to a CPDSL model and testing consistency between the artifacts by executing test cases from $Traces_I(DM)$ and $Traces_P(DM)$. Such test cases are sequences of type $seq(GeoVal \times SpReq \times Output)$; the inputs are chosen from the infinite data types $GeoVal$ and $SpReq$. Since we cannot test with all traces from $Traces_I(DM)$ and $Traces_P(DM)$, we use coverage criteria to select a finite number of test cases based on specific test selection requirements. The intention is to define test requirements that examine various behaviors of the artifacts and that are expected to expose faults.

We define coverage criteria in two ways:

- reformulate existing criteria that are relevant to the modules $ASTS_R$, $output_R$, and $Query_R$;
- define new criteria that test certain DSL-specific aspects.

In what follows, we use $AT = \{I, P\}$ to denote the type of runs/traces. The function $Context_{DM} : (\bigcup_{j \in AT} Traces_j(DM)) \to (R \to Bool)$ is used to compute the state of restrictions (the current state of $ASTS_R$) after executing a trace $t \in \bigcup_{j \in AT} Traces_j(DM)$:

$$Context_{DM}(t) = \begin{cases} q_m & \text{if } t = (g_1, ur_1, o_1) \ldots (g_m, ur_m, o_m) \in Traces_j(DM) \wedge \\ & \quad q(g_1, ur_1, o_1) \ldots (g_m, ur_m, o_m) q_m \in Runs_j(DM) \\ q & \text{if } t = \epsilon. \end{cases}$$

In [AmO08] the authors define an exhaustive list of coverage criteria for source code, specifications, etc. These coverage criteria are typically defined based on common mathematical structures (e.g., transition systems, boolean expressions) and hence they can be reformulated for different contexts. For instance, for a CPDSL model DM the symbolic transition system $ASTS_R$ is a key element for computing speed requests and required distance queries. One can define a criterion that requires test cases to visit each state $p \in Q$ of $ASTS_R$. This is related to the idea of node coverage in [AmO08].

Definition 6 [*Coverage criterion 1 (CC1)*] Let $DM = \langle Obj, Mod, R \rangle$ be a CPDSL model and $j \in AT$. A test suite $T_j \subseteq Traces_j(DM)$ satisfies the criterion CC1 for DM if:

$$\forall p \in Q \; \exists [(g_1, ur_1, o_1), \ldots, (g_n, ur_n, o_n)] \in T_j, \quad 1 \leq i \leq n. \\ Context_{DM}((g_1, ur_1, o_1), \ldots, (g_i, ur_i, o_i)) = p.$$

The criterion CC1 does not enforce any requirements on user speed requests; in each state arbitrary user requests are considered for each object movement.

As an alternative, we can define coverage criteria based on activation and deactivation conditions of restrictions in a CPDSL model. As we discussed in Sects. 5 and 6, activation and deactivation conditions influence output computations by determining active restrictions. Thus, test cases that examine different truth assignments of conditions can potentially cause the artifacts under test to exhibit different behaviors. This can in turn increase

the likelihood of detecting mistakes in the artifacts. In [AmO08] the authors refer to this idea as logic coverage. They also discuss various coverage criteria to capture different possibilities for examining truth assignments of boolean expressions.

We follow the terminology used in [AmO08] and reformulate a logic coverage criterion called "combinatorial coverage" for the collision prevention DSL. We call a condition $cond \in CS$ (see grammar (2)) a clause if it does not contain any logical operators. In other words, clauses have the shape e_1 **compop** e_2. We use the notation Cl_{cond} to denote the set of clauses in $cond \in CS$. One way to examine different truth assignments of $cond$ is to enforce test requirements for clauses in Cl_{cond} to evaluate to each possible combination of truth assignments. For instance, the condition `Distance[m](o1,o2)<10 && Distance[m](o2,o3)>20` with two clauses has four possible truth assignments and hence combinatorial coverage requires test cases for examining these four truth assignments.

For a CPDSL model $DM = \langle Obj, Mod, R\rangle$, we would like to have test cases for examining truth assignments of the activation and deactivation conditions. Thus, we formulate combinatorial coverage at the level of DM. We use the notation $Cl_{DM} = \bigcup_{r\in R}(Cl_{act_r} \cup Cl_{deact_r})$ to denote the set of clauses in the activation and deactivation conditions. The following criterion describes the test requirements of combinatorial coverage for the conditions of a CPDSL model. This criterion can be useful for validating the computations of $Query_R$.

Definition 7 [*Coverage criterion 2* (*CC2*)] Let $DM = \langle Obj, Mod, R\rangle$ be a CPDSL model and $j \in AT$. A test suite $T_j \subseteq Traces_j(DM)$ satisfies the criterion CC2 for DM if:

$$\forall f : Cl_{DM} \to Bool\ \exists[(g_1, ur_1, o_1), \ldots, (g_n, ur_n, o_n)] \in T_j, \quad 1 \le i \le n \\ \forall c \in Cl_{DM}.\ [\![c]\!](g_i) = f(c).$$

Functions $f : Cl_{DM} \to Bool$ in Definition 7 are used to denote truth assignments of clauses in Cl_{DM}.

In addition to reformulating existing criteria for the DSL, new coverage criteria can also be formulated in terms of language-specific features. For instance, for the collision prevention DSL, it is relevant to test the influence of absolute and relative speed limits on output speed computations (see the definition of $outSpeed_R$ and $output_R$ in Sect. 5). Thus, we would like to have a set of test cases to test whether in each state the most restrictive absolute and relative limit determines the output request to the motors. To this end, we define a criterion to test the artifacts by enforcing two test requirements in each state for each object ($obj \in Obj$) and movement ($m \in MovType$):

- the output speed request should be determined by the relative limit for m of obj;
- the output speed request should be determined by the absolute limit for m of obj.

Hence, the specification of this criterion refers to $ASTS_R$ and $output_R$.

Definition 8 [*Coverage criterion 3* (*CC3*)] Let $DM = \langle Obj, Mod, R\rangle$ be a CPDSL model and $j \in AT$. A test suite $T_j \subseteq Traces_j(DM)$ satisfies the criterion CC3 for DM if:

$$\begin{aligned}
&\forall p \in Q, obj \in Obj, m \in MovType \\
&\quad \exists[(g_1, ur_1, o_1), \ldots, (g_k, ur_k, o_k)], [(g'_1, ur'_1, o'_1), \ldots, (g'_\ell, ur'_\ell, o'_\ell)] \in T_j \\
&\qquad \exists 1 \le s \le k, 1 \le t \le \ell\ .\ [\![outSpeed_R]\!]_p(obj, m, g_s, ur_s) = [\![R]\!]_p(obj, m, Abs, g_s) \wedge \\
&\qquad\quad [\![outSpeed_R]\!]_p(obj, m, g'_t, ur'_t) = [\![R]\!]_p(obj, m, Rel, g'_t) \times norm(ur'_t(obj, m)) \wedge \\
&\qquad\quad [\![R]\!]_p(obj, m, Abs, g_s) \neq [\![R]\!]_p(obj, m, Rel, g'_t) \times norm(ur'_t(obj, m)) \wedge \\
&\qquad\quad p = Context_{DM}((g_1, ur_1, o_1), \ldots, (g_s, ur_s, o_s)) \wedge \\
&\qquad\quad p = Context_{DM}((g'_1, ur'_1, o'_1), \ldots, (g'_t, ur'_t, o'_t)).
\end{aligned}$$

The formalization of CC3 indicates that for each state p, object obj, and movement m two traces are required such that:

- p is visited in step s of the first trace (Line 6) where the output speed for movement m of obj is determined by its absolute limit (Line 3);
- p is visited in step t of the second trace (Line 7) where the output speed for movement m of obj is determined by its relative limit (Line 4).

The outputs calculated based on absolute and relative limits for movement m of obj in state p must be different (Line 5).

Fig. 8. Performance model for the safety layer

The criteria from Definitions 6–8 may describe test requirements that cannot be satisfied for certain DSL models. For example, for a CPDSL model DM with `Distance[m](o1,o2)>10`, `Distance[m](o1,o2)<5` $\in Cl_{DM}$, it is not possible to satisfy test requirements of CC2 that enforce `Distance[m](o1,o2)>10` and `Distance[m](o1,o2)<5` to evaluate to true. Such test requirements are called infeasible. We assume that infeasible test requirements are eliminated in a practical way in test case generators and do not consider them as part of the definitions of the coverage criteria.

In this section, we first defined what it means for an artifact to be compliant to a DSL model. Afterwards, we formalized a relation to describe the consistency of two artifacts of a CPDSL model with respect to a set of relevant implementation details. The notion of test case is also formalized and a mechanism for assigning verdicts (**pass** or **fail**) after test case execution is introduced. Various coverage criteria are formalized to enable effective test case selection from an infinite set of candidate test cases. In Sects. 8 and 9, we discuss about applying these ingredients in practice for testing the generated artifacts and the results we have obtained in the case study.

8. Practical remarks

In this section we describe the environments that we have created for performing test experiments on the implementations generated by the code generator from [MHA13] (Sect. 8.1) and the performance analysis models of [BRM+13] (Sect. 8.2). The main challenge is to create a test environment based on the level of abstraction established in Sect. 7.3. An important decision is the interface that is used for test case execution. For our specific case, we see two possible options:

- use a test interface that consumes geometric models, and computes distances from them;
- use a test interface that consumes distance values.

We have chosen for the second option, because it simplifies the derivation of the test cases. Moreover, it allows the tests to focus on the required distance values instead of the internal logic of geometric models which is beyond the scope of the safety layer.

Afterwards, we discuss the procedure that we use for generating test cases based on the test selection criteria (Sect. 8.3). Test data selection from infinite domains is addressed by symbolically representing the test requirements specified by the coverage criteria and by using solvers to find concrete values that satisfy these requirements.

8.1. Implementation code of the safety layer

Our tests on the implementation code are based on the code generator from [MHA13]. In addition, we have developed test stubs for the external interfaces. In particular, we have developed two test environments to observe:

- the computed speed requests (testing conformance to DSL models);
- the required distance queries (validating consistency with performance models).

In both environments, we have replaced the component in the real system that computes distance values by a stub that returns the distance values provided by the generated test cases. This stub repeatedly provides new input values to the generated code. For each environment, we have developed a specific stub that produces the required type of output, i.e., the computed speed requests or the set of performed distance queries.

8.2. Performance model of the safety layer

Our tests on the performance analysis model are based on the model generator from [BRM+13]. For a given CPDSL model, the generated performance model consists of the following components (see Fig. 8):

- *Scenarios* describes a set of execution scenarios;
- *DSL Computations* describes the distance queries required for evaluating the restrictions;
- *Query Execution* models the way distance queries are performed in the system.

The component *Scenarios* contains execution scenarios. Each step of a scenario mimics an interaction of the safety layer with the sensors. In each step, the *Scenarios* component determines the fragments of the DSL model (i.e., clauses from the conditions and the effect expressions) that need to be evaluated based on some probability distributions. Then, it sends stimuli to *DSL Computations* to trigger the evaluation of the required fragments. The *DSL Computations* component receives the stimuli from *Scenarios* and evaluates the corresponding clauses/expressions of the DSL model. Required distance queries are delegated to *Query Execution*. The component *Query Execution* consists of subcomponents that model the behavior of the cache and a sampling method that estimates the execution time for distance queries that are not available in the cache. In [BRM+13] the authors perform simulations on this model for different scenarios. Using the obtained execution times, they estimate the execution time of the safety layer described by the DSL model.

For a given CPDSL model, the component *DSL Computations* is generated automatically using a transformation to POOSL and the other components are constructed independent of DSL models. Our approach aims to validate the *DSL Computations* component of the performance model. Hence we replace:

- the *Query Execution* component by a stub that reports the performed queries;
- the *Scenarios* component by a stub that indicates which fragments of the DSL model should be evaluated based on the test cases.

Our approach validates only the set of time-consuming operations described by the performance models from [BRM+13]. The authors of [BRM+13] have also measured the execution time of these operations, and compared the performance predictions from the analysis models with real observations on the implementation code.

8.3. Test case generation

Let $DM = \langle Obj, Mod, R \rangle$ be a CPDSL model. To generate test cases satisfying a coverage criterion CC for an artifact described by (STS, $compute$), we automatically extract:

- $output_R$ (the function module from Sect. 5) if $compute\colon GeoVal \times SpReq \to SpReq$;
- $Query_R$ (the function module from Sect. 6) if $compute\colon GeoVal \times SpReq \to \mathcal{P}(DistQuery)$.

For actual CPDSL models, $ASTS_R$ contains a large number of transitions. Therefore, we do not explicitly generate $ASTS_R$. For each coverage criterion, the corresponding test case generator uses the activation and deactivation conditions of the restrictions to explore $ASTS_R$ in an on-the-fly manner.

The computations in $ASTS_R$, $output_R$, and $Query_R$ are influenced by values from the infinite domains *SpReq* and *Dist*. Our test case generation procedures treat data items from these domains symbolically. For a given coverage criterion CC, we start exploring $ASTS_R$ from the initial state. In each state, we symbolically build a condition in terms of geometric model values and user speed requests that refer to the requirements of CC. We use a solver to find a satisfiable assignment for the constructed condition. If the condition is satisfiable, we compute the next state and the expected output. Otherwise, the requirement is not satisfiable. We repeat the same procedure until all requirements for CC are satisfied or known to be infeasible. Generating minimal sets of test cases for each criterion is outside the scope of this article.

Using a state-of-the-art SMT solver, e.g., Z3 [DeB08], as the back-end solver is an option. However, most SMT solvers provide limited support for solving non-linear expressions. In our realistic CPDSL models, exponentiation is used to specify smooth brake patterns. Thus, we have decided to use Mathematica [Math] as the back-end solver.

9. Case study results

By formalizing the semantics and the relevant implementation details, and by testing the generated artifacts from [MHA13] and [BRM+13] for a realistic CPDSL model, we have encountered five main issues. These give an indication of what can go wrong in model transformations. In order to test the artifacts, we have implemented (non-optimized) test case generators based on the coverage criteria CC1 and CC3. For the considered realistic DSL model (consisting of five objects, six geometric models, and nine restrictions), we have generated approximately 10,000 test cases for each artifact based on CC1 and CC3. Test case generation takes approximately one hour per artifact. Generated test cases were executed in less than 5 min.

The discovered issues show that formalizing the semantics and implementation details, and conformance testing based on them, provide effective ways for reasoning about the DSL and validating the generated artifacts. The redundancy introduced by these mechanisms can reduce the potential for mistakes.

9.1. Mathematically undefined operations

The syntax of the collision prevention DSL allows arithmetic operations inside conditions and effects. Some operations may not be mathematically defined, for example, because of a division by zero, or a square root of a negative number. Thus we should provide a semantics for interpreting undefined operations. For the collision prevention DSL, this issue can be encountered with respect to conditions and expressions. We assume that conditions return the value *false* for undefined cases, although actual DSL models did not include conditions with undefined operations. In the case of effect expressions, we see two interpretations:

- "ignore the effect" this is a general solution. It requires *Expr* in Sect. 5.2 to be a partial function. In turn, the semantics in Sect. 5.4 should be extended in order to take the partiality into account;
- "assume the effect to have value 0 for undefined cases" this is a specific solution for the collision prevention DSL which conservatively attempts to stop the objects affected by the undefined effect. It only requires a change in the way that elements of *ES* are interpreted.

We have defined the semantics for both interpretations. The code generator from [MHA13] uses the second interpretation. In our experiments the test cases based on the first interpretation fail, whereas the test cases based on the second interpretation pass. This was discovered by test cases satisfying criterion *CC1*; perhaps random testing could also identify this difference.

Time consuming operations of a CPDSL model depend on the distance queries required for determining the state of the restrictions and for evaluating the effect expressions for active restrictions. Thus, the formalization of Sect. 6.1 is affected by the interpretation of undefined conditions. The two interpretations of expressions (mentioned above) evaluate effect expressions for active restrictions and only differ in the final value they produce after evaluation. Hence, the interpretation of undefined expressions does not affect the formalization of the implementation details.

9.2. Semantics of restrictions

From the example of Fig. 4 one can see that a restriction may not specify any effects for a limit type $lt \in LimType$ of a movement $m \in MovType$ for $obj \in Obj$. The semantics from Sect. 5.4 resolves limits that are not specified by any $r \in R$ in the context of the set of all restrictions R by returning default values specified by *DefLim*. However, the code generator from [MHA13] resolves unspecified limits in the context of a single restriction r by returning default values from *DefLim*; hence, restrictions are interpreted as total functions.

This makes a difference for certain CPDSL models. For instance, let $DM = \langle \{obj\}, Mod, \{r_1, r_2\} \rangle$ be a DSL model where:

- $r_1 = (act_1, deact_1, eff_1)$ and $r_2 = (act_2, deact_2, eff_2)$;
- r_1 does not specify a relative limit for rotation of obj;
- r_2 has an effect (Rel, om, e) such that $(obj, Rot) \in om$ and $[\![e]\!](g) > 1$ for $g \in GeoVal$.

As discussed in Sect. 5, a relative limit specifies the maximum percentage of the length of the user speed request

that may be applied to the motors. The code generator from [MHA13] interprets r_1 as a total function that returns 1 (the default value) for the unspecified relative limit for rotation of obj. Since the value 1 is more restrictive than $[\![e]\!](g)$, the relative limit described by DM is 1 for g. On the other hand, the semantics of Sect. 5.4 interprets r_1 as a partial function without any effect on the relative limit and hence $[\![e]\!](g)$ is the relative limit described by DM for g. This is the issue of restriction masking as discussed in [KMM13].

Actual CPDSL models did not include effects above 1 for relative limits. However, we discovered this mismatch by formalizing the semantics. This issue affects the computation of effect expressions in the restrictions and it is only relevant for the semantics. One can also formulate a criterion to detect this mismatch by testing.

9.3. Computation accuracies

Some test cases on the implementation fail because of very small differences in the computed speed request to the motors. This is due to differences in computation accuracies between the test case generator and the generated code. To address this issue in a practical way, we have used an acceptance threshold for comparing expected and actual outputs. We consider the expected and actual speed requests $(e_x, e_y, e_z), (a_x, a_y, a_z) \in \mathbb{R}^3$ for an object to be equal if $\mid e_x - a_x \mid \leq 1 \wedge \mid e_y - a_y \mid \leq 1 \wedge \mid e_z - a_z \mid \leq 1$. This solution may not apply when individual rounding errors lead to large propagation errors.

9.4. Symmetric distance queries

According to the semantics of Sect. 5.1, distance functions are symmetric. In other words, replacing a distance query of the shape `Distance[m](o1,o2)` with `Distance[m](o2,o1)` in a CPDSL model does not influence the output speed request that is applied to the motors. On the other hand, the formalization of Sect. 6.1 implies that `Distance[m](o1,o2)` and `Distance[m](o2,o1)` are distinct distance queries. Thus, although the order of the objects in symmetric queries does not influence output speed requests, it can affect the number of queries that the safety layer performs.

We discovered this discrepancy by writing the formalization of Sect. 6. It is useful to make such assumptions explicit and to make users aware of their impact. As an alternative, we could ensure that the generated artifacts unify such symmetric distance queries based on some fixed ordering on the objects.

9.5. Lazy evaluation for conditions

With respect to the implementation detail described in D1, we have discovered a few issues. While writing the formalization of Sect. 6, we realized that D1 can be interpreted in two subtly different ways. D1 could be applied to the individual activation (act_r) and deactivation ($deact_r$) conditions, or to the combined condition $CurrAct_r := act_r \vee (pb \wedge \neg deact_r)$. For the performance, the combined condition can be beneficial due to the lazy evaluation rule of disjunction, but it turns out that the prototype generators from [MHA13, BRM+13] only apply it to the individual conditions.

By reviewing the implementation of lazy evaluation in the prototype generators from [MHA13, BRM+13], we have also noticed that the generator from [BRM+13] for performance models only considers D1 for conjunctions, which is adequate for the DSL models considered. The code generator from [MHA13] considers both conjunctions and disjunctions.

By providing CPDSL models that include disjunctions in the activation/deactivation conditions, these issues can also be detected by test cases generated based on the coverage criterion CC2.

10. Afterthought: towards a modular semantics for DSLs

The formal semantics of a DSL can also be used by language developers as a means to characterize the main features of the language and their expressiveness. During the formalization of the semantics of the collision prevention DSL, we have observed that its semantics can naturally be split in two modules, focusing on the state-based and function-based aspects, respectively. We formalized the connection of these two aspects in the

definition of trace (see Definition 2) where the label of the current state affects the output computation. This modularity helped us to consider variants of the language by changing the way the history of execution affects output computation. In particular, the simpler version of this DSL as discussed in [KMM13] is stateless, and hence could be described using only a (modified) function-based module.

Although DSLs usually focus on a narrow domain, they typically share some general semantic concepts. Ideally, this would enable the reuse of semantic modules (not necessarily their syntax). A similar direction is sketched in [RVM+12]. The authors consider a set of primary modules with well-defined semantics shared among different DSLs as analysis DSLs (e.g., expressions language module). Formalizing the semantics of DSLs can facilitate the reusability of more complex semantic models.

The separation in semantic modules can help us by selecting appropriate analysis tools for DSLs with known semantics. Since the function-based aspect is dominant in the collision prevention DSL, the use of solvers is very effective. For example, in [KMM13] we have used solvers for verifying properties such as deadlock freedom. Similarly, in Sect. 8.3 we have used solvers for generating test cases. Studying semantics in a modular way has the potential to further reduce the required effort of developing DSLs and their tool infrastructure such as code generators and analysis techniques. In [RVM+12] the authors use analysis tools to perform certain checks on the identified primary modules (e.g., completeness of a set of boolean conditions).

By identifying semantic modules in DSLs, language designers can obtain more insight into the features of languages. Moreover, reuse of semantic modules together with appropriate tool-support for analysis of DSLs can improve the feasibility of DSL approaches in industry.

11. Related work

Various authors have formalized the semantics of DSLs to enable formal verification or to build simulators:

- A method for prototyping visual interpreters and debugging facilities for DSLs is proposed in [SaW08]. The approach is illustrated by a DSL for Petri net models. They extend the meta model of the language with the concept of configuration and use query/view/transformation (QVT) relations to describe the semantics.
- In [SWR+12] the authors formalize the semantics of an industrial DSL. First, the concrete syntax of the language is projected onto an abstract and compositional language consisting of process terms. Then, structural operational semantics (SOS) is used to assign semantics to the obtained process terms. The SOS rules are used for state space generation and model validation.
- A translational approach for prototyping the semantics of a DSL named SLCO is studied in [ABE11]. SLCO is used in a setting with a number of transformations, e.g., to implementation code, or to restricted SLCO models with equivalent observable behavior. The semantics of the DSL is captured by a transformation to an intermediate language called CS. They also introduce a straightforward transformation from CS to labeled transition systems and analyze them by existing tools. The correctness of the transformations is assessed by comparing the underlying labeled transition systems of the source and target models.
- In [RDV09] the authors propose an approach for formalizing DSLs using Maude. The approach is described using a DSL for production systems. Maude is used to describe the meta-model of the language and DSL models. The semantics of the DSL is described using rewrite rules. They use the available tools for Maude to perform simulation, reachability analysis, and model checking.

The DSLs studied in the mentioned works provide domain-specific abstractions for transition systems. In contrast, the dominating aspect of the collision prevention DSL is the function module. Moreover, we have used the semantics to test the generated artifacts.

For establishing the correctness of generated code for a particular DSL model, also [Voe13] focuses on testing. However, he focuses on manually writing unit tests, either at the level of the implementation language, or at the level of the DSL.

A testing approach based on models in the prototype verification system (PVS) is studied in [DMN+15]. The authors have constructed a PVS model for a software component used for trajectory generation. The model is used to prove theorems about the desired behavior of the component and to generate implementation code. Random testing and input space partitioning is used to generate test cases for evaluating the implementation. For each

test case, the computations performed in the implementation is compared against the corresponding symbolic evaluations of the formal model. In contrast, our approach is centred around domain-specific models. We provide formalizations for the semantics and the implementation details that are relevant for multiple transformations. These formalizations are used for assessing the consistency of the generated artifacts.

As an alternative approach, various authors have proposed to assess the correctness of transformations using a test set that consists of multiple models in the source language:

- In [MHD+15], the authors validate transformations from B specifications to C and LLVM implementations. First, grammar-based testing is used to check whether syntactically correct code is generated for a test set that consists of various B specifications. Afterwards, test cases are derived from the B specifications to evaluate the compliance of the generated implementations to their respective B models.
- In [FMO+10], for a model transformation it is tested whether, given source models satisfying the precondition, the transformation produces target models satisfying the postcondition. It is assumed that the meta-models of the source and target languages, and the precondition and postcondition of the transformation are encoded in a constructive logic. A set of predefined criteria is used to generate source models as test data.

Instead of testing the code generator, we test the generated code/analysis model for a particular DSL model. Our approach can be extended to test the transformations with a set of DSL models that are generated based on certain criteria.

12. Conclusions and future work

Our approach for formalizing and testing the consistency of DSL transformations is based on explicitly distinguishing and formalizing:

- semantics: specific for a DSL, but independent of any code generator;
- implementation details: specific for a particular code generator, and relevant for some analysis models.

This difference is particularly important in a context where code generators may change over time, or multiple code generators (for different platforms) are present. A code generator may introduce a wide range of implementation details to realize the semantics of the DSL, e.g., coding standards, internal data structures. We focus on formalizing implementation details that are relevant for the specific analysis models generated from the DSL.

To validate DSL transformations in a practical way, we rely on conformance testing for checking:

- correctness of each generated artifact with respect to the semantics of the DSL;
- consistency between the generated artifacts with respect to the relevant implementation details.

We extensively illustrate our general approach using specific formalizations for an industrial DSL on collision prevention. We have made the following observations on this application of our approach:

- Discovery of inconsistencies: the formalization and testing reveals a surprising number of subtle inconsistencies, which were not discovered before. Our approach helped to detect them, and solve them.
- Selection of analysis tools: the modular semantics explains the effectiveness of solvers and state-based analysis techniques for particular parts of this DSL.

In this article we have focused on testing the generated code and analysis models for a given DSL model. This could be extended to testing the model transformations themselves, based on a test set that consists of several DSL models. This is related to the idea of compiler testing [BaS82] and would be relevant as a regression test when generators are optimized or extended.

Introducing DSLs in industrial practice involves a shift in the software development process. On the one hand, a DSL provides an accessible way for domain experts to describe their systems using a domain-specific abstraction. On the other hand, developing and maintaining a DSL and its code generators and validating the correctness of the generated artifacts requires substantial efforts. Studying the semantics of DSLs in a modular way would enable the language designers to reuse existing modules in different languages. This can potentially reduce the effort required for developing analysis techniques for DSLs.

Acknowledgments

This research was supported by the Dutch national COMMIT program under the Allegio project, and by the European ARTEMIS program under the Crystal project. The authors like to thank the referees for their useful comments on earlier versions of this article.

References

[ABE11] Andova S, van den Brand MGJ, Engelen L (2011) Prototyping the semantics of a DSL using ASF+SDF: link to formal verification of DSL models. In: Proceedings of AMMSE'11, EPTCS, vol 56, pp 65–79

[ABE12] Andova S, van den Brand MGJ, Engelen L (2012) Reusable and correct endogenous model transformations. In: Proceedings of ICMT'12. LNCS, vol 7307. Springer, New York, pp 72–88

[AmO08] Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press, Cambridge

[BaS82] Bazzichi F, Spadafora I (1982) An automatic generator for compiler testing. IEEE Trans Softw Eng 4:343–353

[DeB08] De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings of TACAS'08. LNCS, vol 4963. Springer, New York, pp 337–340

[DMN+15] Dutle AM, Munoz CA, Narkawicz AJ, Butler RW (2015) Software validation via model animation. In: Proceedings of TAP'15. LNCS, vol 9154. Springer, New York, pp 92–108

[EhE08] Ehrig H, Ermel C (2008) Semantical correctness and completeness of model transformations using graph and rule transformation. In: Proceedings of ICGT'08. LNCS, vol 5214. Springer, New York, pp 194–210

[FMO+10] Fiorentini C, Momigliano A, Ornaghi M, Poernomo I (2010) A constructive approach to testing model transformations. In: Proceedings of ICMT'10. LNCS, vol 6142. Springer, New York, pp 77–92

[Gau95] Gaudel MC (1995) Testing can be formal, too. In: Proceedings of TAPSOFT'95. LNCS, vol 915. Springer, New York, pp 82–96

[KMM13] Keshishzadeh S, Mooij AJ, Mousavi M (2013) Early fault detection in DSLs using smt solving and automated debugging. In: Proceedings of SEFM'13. LNCS, vol 8137, pp 182–196

[KeM14] Keshishzadeh S, Mooij AJ (2014) Formalizing DSL semantics for reasoning and conformance testing. In: Proceedings of SEFM'14. Springer, New York, pp 81–95

[Ler09] Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 52(7):107–115

[Math] Wolfram Research Inc., Mathematica 10.0.1.0. http://www.wolfram.com. Accessed Aug 2015

[MHA13] Mooij AJ, Hooman J, Albers R (2013) Gaining industrial confidence for the introduction of domain-specific languages. In: Proceedings of IEESD'13. IEEE, pp 662–667

[MHA14] Mooij AJ, Hooman J, Albers R (2014) Early fault detection using design models for collision prevention in medical equipment. In: Proceedings of FHIES'13. LNCS, vol 8315. Springer, New York, pp 170–187

[MHD+15] Moreira AM, Hentz C, Déharbe D, de Matos ECB, Neto JBS, de Medeiros Jr V (2015) Verifying code generation tools for the B-method using tests: a case study. In: Proceedings of TAP'15. LNCS, vol 9154. Springer, New York, pp 76–91

[MHS05] Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. Comput Surv ACM 37:316–344

[NCB+12] Nagy I, Cleophas LG, van den Brand M, Engelen L, Raulea L, Mithun EXL (2012) VPDSL: a DSL for software in the loop simulations covering material flow. In: Proceedings of ICECCS'12. IEEE, pp 318–327

[RVM+12] Ratiu D, Voelter M, Molotnikov Z, Schaetz B (2012) Implementing modular domain specific languages and analyses. In: Proceedings of workshop on MoDeVVa'12. ACM, New York, pp 35–40

[RDV09] Rivera JE, Durán F, Vallecillo A (2009) Formal specification and analysis of domain specific models using maude, simulation: transactions of the society for modeling and simulation international. Sage Publications, USA, pp 778–792

[SaW08] Sadilek DA, Wachsmuth G (2008) Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: Proceedings of ECMDA-FA'08. LNCS, vol 5095. Springer, New York, pp 63–78

[SBP+08] Steinberg D, Budinsky F, Paternostro M, Merks E (2008) Eclipse modeling framework. Pearson Education, London

[SWR+12] Stappers FPM, Weber S, Reniers MA, Andova S, Nagy I (2012) Formalizing a domain specific language using sos: an industrial case study. In: Proceedings of SLE'11. LNCS, vol 6940. Springer, New York, pp 223–242

[TFG+07] Theelen BD, Florescu O, Geilen MCW, Huang J, van der Putten PHA, Voeten JPM (2007) Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the international conference on formal methods and models for codesign. IEEE, pp 139–148

[Tre08] Tretmans J (2008) Model based testing with labelled transition systems. In: Formal methods and testing. LNCS, vol 4949. Springer, New York, pp 1–38

[BrG13] van den Brand M, Groote JF (2013) Software engineering: redundancy is key. Sci Comput Program Elsevier 97:75–81

[BRM+13] van den Berg F, Remke A, Mooij AJ, Haverkort B (2013) Performance evaluation for collision prevention based on a domain specific language. In: Proceedings of EPEW'13, vol 8168. Springer, New York, pp 276–287

[DKV00] van Deursen A, Klint P, Visser J (2000) Domain-specific languages: an annotated bibliography. In: SIGPLAN notices, vol 35. ACM, New York, pp 26–36

[VLH+13] Verriet J, Liang HL, Hamberg R, van Wijngaarden B (2013) Model-driven development of logistic systems using domain-specific tooling. In: Proceedings of CSD&M. Springer, New York, pp 165–176
[Voe13] Voelter M (2013) DSL engineering, version 1.0. http://dslbook.org. Accessed Aug 2015
[Xte14] Xtext (2014) Version 2.7. http://www.eclipse.org/Xtext/. Accessed Aug 2015

Received 26 January 2015
Accepted in revised form 29 January 2016 by Dimitra Giannakopoulou, Gwen Salaün, and Michael Butler
Published online 15 March 2016