Industrial Experiences with the Evolution of a DSL

Mathijs Schuts
mathijs.schuts@philips.com
Philips
Best, The Netherlands
Radboud University
Nijmegen, The Netherlands

Marco Alonso
marco.alonso@philips.com
Philips
Best, The Netherlands

Jozef Hooman
jozef.hooman@tno.nl
TNO
Eindhoven, The Netherlands
Radboud University
Nijmegen, The Netherlands

Abstract

At Philips IGT, we develop and produce interventional X-ray systems. For a controller in these systems, we have an approximately five years old domain specific language. Like general programming languages, domains specific languages also evolve. These languages co-evolve together with their domain. The language used at IGT was initially created for one system instance. Because of our positive experiences with the language, we want to evolve the language to support a family of systems. In this paper, we report on our experiences with the modifications we made to the original language. We made these changes preserving the behavior of the existing system instance. To prevent confidentiality issues, we use a Lego robot in our examples.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Keywords: Domain Specific Modelling, Maintenance of DSM Languages, Evolution of DSM Languages, Industrial Application

ACM Reference Format:
Mathijs Schuts, Marco Alonso, and Jozef Hooman. 2021. Industrial Experiences with the Evolution of a DSL. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM '21), October 18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486603.3486774

1 Introduction

Model-based techniques use models to describe several aspects of a system, from its architecture to its behaviour. These techniques are usually applied to raise the level of abstraction, improve communication between multi-disciplinary teams and, ultimately, to increase productivity. An approach to model-based techniques is Model Driven Engineering (MDE) [13]. MDE is a software methodology focused on creating and exploiting models, its goal is to use models to describe particular uses cases (i.e. domains) of the system and to use those same models as the basis for the actual implementation.

Models are often created using Domain Specific Languages (DSLs) [12]. DSLs trade generality for conciseness, they are used to describe domain requirements and behaviour in a concise way by providing notations and constructs tailored to the domain. Their concepts and relations are known to the domain experts allowing them to contribute directly to the development process. The limited expressiveness and the reduction of programming expertise needed to create the models correspond to gains in productivity and reduced maintenance costs.

DSLs are created by developers and domain experts to capture all the knowledge about the domain in the DSL, while the execution semantics are covered by the DSL compiler/interpreter. Consequently, domain experts understand, modify, and even develop DSL-based models. The use of DSLs supports software evolution. Working at a high level of abstraction preserves the user from unsafe or inconsistent program modifications with respect to the original design. It makes the code of developed applications more clear; simplifying the documentation and reducing the need of updating when the application evolves. As the domain makes the DSL self-documented, it shifts the paradigm from document-based development to model-based development.

However, the DSL is strictly coupled to the domain and its requirements/capabilities at the time in which the DSL is written. If the domain requirements and/or capabilities change, then the DSL could become inadequate to deal with the changed domain. When the domain requires new features, we may need to develop a new DSL or adapt the existing one to the new features. To develop a new DSL, even if small, implies a great effort because the development of the supporting environment is a challenging task. To adapt the existing DSL to the changed domain by reusing part of the original DSL implementation requires a particular DSL design. In this work, we focus on the evolution of the DSL while preserving its existing capabilities, expressiveness and execution semantics.

In this paper, we report about the evolution of a DSL at Philips IGT, a business unit of the Philips company. Philips IGT develops and produces interventional X-ray systems, see

DSM '21, October 18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9106-1/21/10.
https://doi.org/10.1145/3486603.3486774

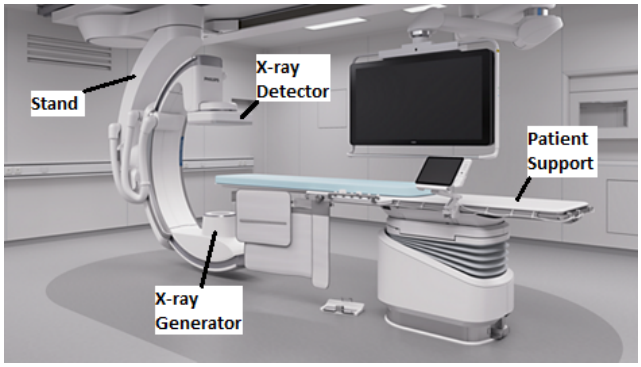

Figure 1. Interventional X-ray system

Figure 1. These systems can be used for many medical treatments. The systems have a sub-system which is responsible for positioning the X-ray beam with respect to the patient. The user of the system can initiate movements by joystick requests to change the region of interest of which the X-ray pictures are taken. This sub-system has a controller which decides how the movement requests need to be handled. The controller can reduce the speed of a movement when system comes too close to the patient or stop the movement when the system comes even closer to the patient.

Approximately five years ago, we created the so-called Azurion DSL to describe the behavioral rules for the controller. The user of the language describes how movement requests are handled depending on the state of the system.

The Azurion DSL was created for a single system type. At creation, the language included a pre-defined set of movements, system states (e.g. motor defect) and actions (e.g. stop movement) as part of its grammar. These language features are hard-coded in the generator to be able to generate source code in our target language C++.

Because of the positive experiences with the Azurion DSL at Philips IGT, the aim is to extend the DSL to support a family of products. The language features need to be extended. Extending language features means: 1) extend the expressiveness of the language, 2) updating the code generator, and 3) using the new features in the language instances. Our experience is that it works best if DSL experts perform steps 1 and 2. DSL users are only concerned with performing step 3. We co-evolved our language in such a way that, e.g., movements, system states and actions are no longer fixed and predefined, but can be defined by the user in the language instance.

Due to the nature of our systems, it is paramount that we do not change the behavior of existing released products. Our goal is to evolve the language to support a family of systems without changing the generated artifacts and hence preserving its verified behavior.

We took an incremental approach for the DSL evolution. Every evolutionary change started with storing the current state of the generated artifacts. After which we made a change to the grammar, the code generator and the instance of the language. Lastly, we generated the artifacts again and checked if they were equal to the stored state. If the artifacts were different, the generator or instance needed to be changed until the artifacts were equal.

For maintainability and reuse, we splitted the new grammar over multiple languages. The lower-level languages we created could be reused in languages for different domains.

This paper is organized as follows. Related work is presented in Section 2. In Section 3 the DSL is introduced with the help of a Lego robot example. Next we describe the changes we made to the language to support a family of systems and how we splitted up the grammar in Section 4. Section 5 describes the industrial embedding of our language. The conclusion is in Section 6.

2 Related Work

Many papers have been written about the initial creation of DSLs in industry. For instance, over 20 industrial applications of DSLs are described in [10]. They observed that DSLs are beneficial for design guidance and early error prevention or detection. In addition, they report that DSLs increase productivity due to the raised level of abstraction. As any piece of software, also DSLs evolve over time. The challenge of the evolution of software, with a focus on the co-evolution of meta-models and models, were already identified in [6]. Tratt discusses the evolution of a simple state machine DSL, concentrating on the consequences of adding more functionality and making it more robust [11].

A large case study can be found in [8]; they study the impact of a large number of revisions of four modeling languages provided by the open source Graphical Modeling Framework (GMF). This framework is maintained by more than ten developers in several countries and its evolution is well-documented which makes it very suitable for studies on co-evolution. The authors of [9] describe the problem of a large number of DSL ecosystems at the company ASML. The largest ecosystem consists of 22 EMF-based DSLs, 95 QVT model transformations, and 5500 unit-test models supporting development of these transformations. Co-evolution is more complex than a single DSL because of the reuse of concepts between DSLs and implicit relations. The paper describes a high-level architecture for co-evolving models.

Different ways in which metamodels may evolve and the relations with modeling artifacts are identified in [5]. The paper also discusses the ingredients for a comprehensive solution, including a possible implementation. The authors of [3] disusses the analysis of dependencies between modifications which is an important challenge when aiming at complete automation of the coupled evolution of meta-models and models. To automate co-evolution, the paper [2] proposes

a higher-order model transformation based on a representation of the metamodel evolution. The approach of [4] automatically generates suggestions to repair inconsistencies based on consistent change propagation. A survey of a large number of approaches for meta-model co-evolution can be found in [7].

3 Case

In this section, we explain the DSL as far as needed to understand the paper. The DSL is used at Philips IGT. Because of confidentiality reasons, we illustrate our DSL using a robot example with a comparable domain. Note that we did tailor the grammar to fit our example.

We start this section with a description of the robot. Then we explain the characteristics of the playing field. We describe how the robot processes movement requests from a remote controller. Next we introduce the DSL with an example instance. Followed by an analysis of the evolvability of our DSL. We end the section with a comparison between the Lego DSL and the Azurion DSL.

3.1 Robot

For the case, we assume that we drop a robot on a playing field. Its mission is to explore the playing field and search water in colored lakes. When it has found a lake, it will sent the color of the lake back to the remote operator.

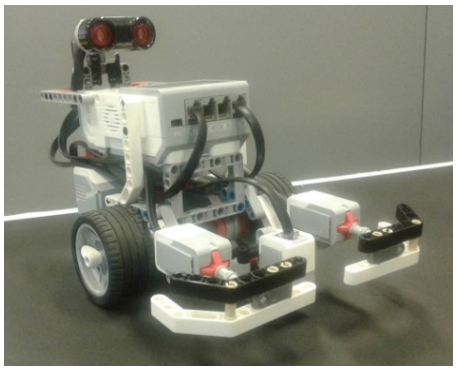

Figure 2. Robot

Figure 2 depicts the robot we use to explore the playing field. The robot has the following hardware mounted to its chassis. It has three wheels: one at the back in the middle and two wheels on each side which are individually driven by a motor. At the front, it has two bumpers that can detect if it crashed onto an object. Also at the front is a downwards facing color sensor. An ultrasonic sensor is mounted on the top.

For driving the robot, we have a remote control. There are joysticks for the following movement requests:

- Analog deflection for moving forward at the desired speed.
- Analog deflection for moving backward at the desired speed.
- Sharp left turn. Left wheel moves backward. Right wheel moves forward.
- Sharp right turn. Right wheel moves backward. Left wheel moves forward.
- Wide left turn. Right wheel moves forward. Left wheel does not move.
- Wide right turn. Left wheel moves forward. Right wheel does not move.

In this paper, we discuss the controller that is responsible for decision making. The implementation for receiving remote control messages, reading the sensors and moving the motors is out of scope.

3.2 Playing Field

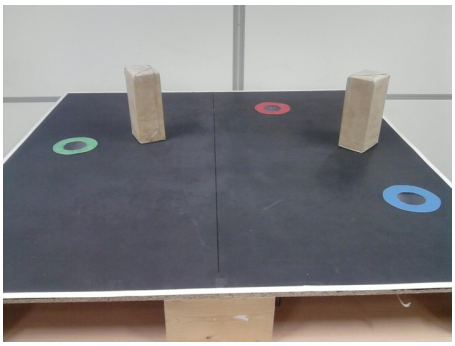

Figure 3. Playing field

Figure 3 is a top view of the playing field with a robot on it. The playing field has the following characteristics:

- The playing field is square and flat. The surface of the playing field is black and the edge has a white line.
- The playing field has lakes with colored edges.
- There are bricks.

The color sensor at the front of the robot can send the color of the lake back to the remote operator when it has found one. In addition, the color sensor is used for edge detection; to make sure that the robot will not fall of the playing field. To detect the bricks, the robot has two types of sensors. The distance to a brick is determined with the reading of the ultrasonic sensor. The ultrasonic sensor needs calibration before use. The bumpers can detect a collision with a brick.

The robot uses a coordinate system to know its place on the playing field. It distinguishes the following zones:

- 5 cm before the edge of the playing field. The edge is the white line, see Figure 3.
- On the edge.
- Over the edge.

The coordinates of the zones are calibrated and stored on the robot. During start-up, the robot checks if the calibrations are present.

3.3 Movement Requests

Movement requests from the joysticks are processed by the robot. Depending the state of the robot, the position of the robot or the sensor readings of the robot, the controller will alter movement requests. We will give an example for each of the three reasons a movement request can be altered:

State. When a certain state is true (for instance, a sensor is not calibrated or there is a faulty sensor), the movement can be altered to for instance a slower speed or not start at all.

Position. Zone near the edge of the playing field, the robot can reduce the speed to a lower value.

Sensor value. The robot has sensors for detecting a collision. When detecting a collision, all movements can be stopped.

Note that when none of these three reasons are present, the robot will execute the given movement request with the requested speed.

3.4 Language

Our language and generators are created using the Xtext and Xtend Eclipse plug-ins [1].

Listing 1 is an example instance of the Lego DSL. It is used as a running example throughout the paper.

Next we explain what the example instance describes. All words in bold and blue are keywords i.e. part of the grammar. Lines 1 till 3 define a condition called *ReducedPerformanceCondition*. This condition is false when the position and ultrasonic sensor are calibrated. Note that **Position calibrated** and **Ultrasonic sensor calibrated** are part of the language. A language instance can have more conditions. One condition can be used in another condition.

As explained in Section 3.3, a movement request can be altered by the controller. Lines 4–15 provide four examples. The first one describes that all movements will be executed with safe speed when the *ReducedPerformanceCondition* condition is true. This line implements the following behaviour for the controller. If the requested movement speed is faster than safe speed, then the movement will move with safe speed. Alternatively, if the requested movement speed is slower than safe speed, then the movement will move with the requested speed. When no movement is requested, the controller will not execute this rule. The word between "[" and "]" is an identifier which is used in the generated code and needs to be unique.

The second rule defines that when the collision sensor is active all movements except for the *BackwardMove* are quickly stopped with stop release. The latter means that the movement can continue when the joystick is released first.

```
1  Condition ReducedPerformanceCondition
       is
2    (NOT Position calibrated) OR
3    (NOT Ultrasonic sensor calibrated)
4  [RedPerf] WHILE
       ReducedPerformanceCondition
5    DO Maximum speed is safe
6    APPLIES TO all movements
7  [CollCond] WHILE Collision sensor
       active
8    DO QStop with stop release
9    APPLIES TO all movements except
         BackwardMove
10 [InEdgeZone] WHILE Inside Before edge
       of playing field zone
11   DO Maximum speed is safe
12   APPLIES TO SharpLeft + SharpRight +
         WideLeft + WideRight
13 [InEdgeZone] WHILE Inside In edge of
       playing field zone
14   DO NormalStop
15   APPLIES TO ForwardMove
16 Movement ForwardMove description:
17   Main Action defined as Forward move
18 End of movement ForwardMove
       description
19 Movement BackwardMove description:
20   Main Action defined as Backward
         move
21 End of movement BackwardMove
       description
22 Movement SharpLeft description:
23   Main Action defined as Sharp left
         turn
24 End of movement SharpLeft description
25 Movement SharpRight description:
26   Main Action defined as Sharp right
         turn
27 End of movement SharpRight
       description
28 Movement WideLeft description:
29   Main Action defined as Wide left
         turn
30 End of movement WideLeft description
31 Movement WideRight description:
32   Main Action defined as Wide right
         turn
33 End of movement WideRight description
```

Listing 1. DSL example statements

Note that *BackwardMove* is an instance of **Backward move**. Hence, for each movement instance (like *BackwardMove*), a movement definition (specified in the language) is required.

The third rule makes the controller alter the *SharpLeft*, *SharpRight*, *WideLeft* and *WideRight* movements to safe speed when the robot approaches the edge of the playing field.

The last rule stops the *ForwardMove* when the robot is on the edge of the playing field.

The movements used in the above rules are defined on lines 16 to 33. These lines define six movements, all in the same way. We only explain the first movement. The movement name can be chosen by the user of the language. In this example *ForwardMove*. The movement name is mapped to **Forward move** which is a keyword. Because of **Forward move**, the code generator knows what to generate when *ForwardMove* is used.

From the language instance, the artifacts in C++ source files are generated. The code generator generates all behavior as specified by the language instance. Depending on the system capabilities (for instance, some movements might not be present because a customer chooses not to buy certain options) some code will never be executed.

3.5 Supporting a Family of Systems

Figure 4. Another Robot

The language we created was intended for one robot, see Figure 2. To support a family of systems, we also need to deal with robots with different hardware configurations. Figure 4 is an example of a new robot with different hardware. Compared with the robot of Figure 2, this robot has a sensor at the back to make sure it will not fall of the playing field when moving backwards. This sensor value is present in the new hardware instance and is not present in the old hardware instance of the robot. There are other differences, such as two addition color sensors at the front and a measuring arm. To deal with the large number of sensors the robot has two controllers.

In the next section, we describe the changes we made to support a family of systems instead of a single system instance. Some features of the presented language are not variable. They can only be added by changing the grammar and the generator of the language.

We use Listing 1 to decide what needs to co-evolve with the domain:

- **Movement:** Movements are part of grammar. For instance, **Backward move** is a keyword on line 20.
- Zones, system states and actions are also part of the grammar:
  - **Position: Before edge of playing field zone** on line 10 is a keyword.
  - **System state:** Line 2 uses the **Position calibrated** keyword.
  - **Action: NormalStop** is a keyword, see line 14
- **Configuration:** Because we need to support a family of systems, it might be that depending on the used chassis of the robot a specific movement is available in e.g. two instances of the system, but we want to have different behaviour for one of the two system instances. With the language presented in Listing 1, we cannot support such a case.

Summarizing, observe that the grammar and the generator have to be adapted to deal with new movements, zones, actions and system states, while ideally only the instance needs to be updated.

3.6 Validity of Example

Here we compare the presented Lego DSL with the Azurion DSL used for the real system from Figure 1.

The keywords with capital letters (e.g. **NOT**, **WHILE** , **APPLIES TO**) are exactly the same in both languages. The actions **Maximum speed is safe**, **QStop with stop release** and **NormalStop** are a subset of the actions present in the Azurion DSL. As mentioned, it was not allowed to disclose the actual positions, system states or movements. The zones **Before edge of playing field zone** and **In edge of playing field zone** are not used at Philips IGT. However, one can imagine that the system has some notion about where its parts are such that collisions between stand and

patient support can be avoided. The same is the case for people in the room. In the Lego DSL we have **Collision sensor active** for this as part of the grammar. The actual DSL also has keywords for states related to the calibration of sensors. The movements **Forward move**, **Backward move** , **Sharp left turn**, **Sharp right turn**, **Wide left turn** and **Wide right turn** are not part of the Azurion DSL. The Azurion DSL has movement keywords in the grammar for the movements the stand and patient support can make.

Figure 1 is an example hardware configuration of the Azurion system. The customer can choose different stands and patient supports. Depending on the hardware configuration a customer has chosen, the current states, sensors and movements might be different.

4 Evolution

In this section, we describe how we evolved the language. First we describe how the language co-evolved with the domain. Then the internal structure of the language has changed. Last we describe how we divided the grammar over multiple languages to improve maintainability and to allow the re-use of languages.

4.1 Grammar Changes of Language

In this section, we describe how our language co-evolved with the domain.

Position. In Listing 1 on line 10 **Before edge of playing field zone** was part of the grammar. In the evolved language called Movement Specification Language (MSL), zones can be added by the user of the language without changing the grammar or generator. Listing 2 shows that zones are user defined in the MSL.

```
1 msl UserDefinedZones{
2   Zones {
3     UserDefined
          BEFORE_EDGE_OF_PLAYING_FIELD_ZONE
           [function:
          "isBeforeEdgeOfPlayingFieldZone()"]
4     UserDefined
          ON_EDGE_OF_PLAYING_FIELD    [
          function: "isOnEdgeOfPlayingField()"]
5   }
6 }
```

Listing 2. DSL example user defined zones

The user defined name *BEFORE_EDGE_OF_PLAYING_FIELD-_ZONE* is mapped by the generator to a manually implemented function with the *isBeforeEdgeOfPlayingFieldZone* name. The assumption is that these functions have a boolean return value. During building the source code, the C++ compiler will statically check if this is indeed the case. We use "{" and "}" because multiple groups can be created.

System state. System states in the old language were part of the grammar, see e.g. Listing 1 on line 2. In the evolved language, system states can be defined by the user. In Listing 3, we use the same notation as in Listing 2. Here we also have the assumption that the defined functions have a boolean return value.

```
1 msl UserDefinedSystemStates{
2   SystemStates {
3     UserDefined
          POSITION_IS_CALIBRATED  [
          function: "isPositionCalibrated()"]
4     UserDefined
          ULTRASONIC_SENSOR_IS_CALIBRATED
           [function:
          "isUltrasonicSensorCalibrated()"]
5     UserDefined
          COLLISION_SENSOR_IS_ACTIVE  [
          function: "isCollisionSensorActive()"]
6   }
7 }
```

Listing 3. DSL example user defined system states

Action. Analogously, Listing 4 is an example of an user defined action. The action is mapped to a manually implemented class that implements the desired behaviour. As mentioned, the class is implemented in C++. The C++ linker will produce an error when the class does not exist. In line 3, we have a string that is used for documentation purposes. This notation can be used for everything that is defined by the user.

```
1 msl UserDefinedActions{
2   Actions {
3     @user.doc("This documentation describes what
          the NormalStop action does.")
4     UserDefined NormalStop [class:
          "NormalStop"]
5     UserDefined
          QuickStopWithStopRelease  [
          class: "QuickStopWithStopRelease"]
6     UserDefined MaximumSpeed [class:
          "MaximumSpeed"]
7   }
8 }
```

Listing 4. DSL example user defined actions

4.2 Supporting a Family of Systems

To support a family of systems, we introduced a capability specification language (CSL). The CSL is used to specify systems, configurations and their capabilities.

A CSL model defines physical objects. The chassis type of our example robot is a physical object. For the Azurion system, one can think of stands or table supports. Physical objects have capabilities. A capability is the ability of the system to do something. In our case, we define movements as capabilities.

A CSL model also defines configurations. A configuration is a system type instance. For instance, the combination of stands and table supports that a customer can chose for the Azurion system. A configuration refers to a physical object and specifies which capabilities are available for that configuration.

Below is a detailed description of the evolution of the language for movements and configurations.

Movement. Listing 1 has movements that are part of the grammar, see for instance line 20. To support a family of systems, we need to be able to easily add new movements. In Listing 5, we can add user defined movements per chassis. We start by defining a name for the chassis type, in this case *ChassisA*. Next we add movements to the chassis. We add a movement with the identifier *ForwardMove*, and define its maximum and safe speeds. How behaviour is added to a movement will be described in the next section.

```
1 csl Movements
2 {
3   Chassis ChassisA
4   {
5     ForwardMove  maxSpeed : 500 mm/
          sec safeSpeed : 100 mm/sec
6     BackMove  maxSpeed : 200 mm/sec
          safeSpeed : 100 mm/sec
7     ...
8   }
9 }
```

Listing 5. DSL example movement

Configuration. From the different chassis, we can compose a system configuration, see Listing 6. The movements that are used in the **ChassisConfiguration** are imported on Line 1. These imported movements are defined in Listing 5. A system configuration inherits the movements from a chassis, in this example *ChassisA*. When required, a system configuration can override the safe speed from the chassis definition. This is done on lines 8 and 15. The default values are used otherwise.

Lines 19–21 define constants for sets of movements. Constants are variables that cannot be altered after their definition. Movements can be added to these constants, see for instance line 19. Line 20 and 21 define the same set of movements, but in a different way. Next to the **difference** keyword, we also added the **union** and **intersection** set operators.

```
1  import "Movements.csl"
2
3  csl Configurations
4  {
5    @gen.config("isConfigurationA()")
6    ChassisConfiguration ConfigurationA
         : ChassisA
7    {
8      ForwardMove  safeSpeed : 50 mm/
           sec
9      BackMove
10     ...
11   }
12   ChassisConfiguration ConfigurationB
         : ChassisA
13   {
14     ForwardMove
15     BackMove  safeSpeed : 50 mm/sec
16     ...
17   }
18
19   const STRAIGHT_MOVEMENTS =
         ForwardMove + BackMove
20   const TURN_MOVEMENTS_1 = SharpLeft
         + SharpRight + WideLeft +
         WideRight
21   const TURN_MOVEMENTS_2 = (
         allMovements)=>difference(
         STRAIGHT_MOVEMENTS)
22 }
```

Listing 6. DSL example configuration

The annotation on line 5 is used by the code generator. When code needs to be generated that is specific for *ConfigurationA*, the generator will use *isConfigurationA()*. The assumption is that *isConfigurationA()* is a manually implemented function with a boolean return value.

4.3 Putting It All Together

Behaviour. In this paragraph, we describe a replacement of Listing 1. In Listing 7, we present the behavioral part of the language that is ready to support the family of systems. At the top, the earlier described listings are imported.

The condition statement now uses the imported user defined system states on lines 8 & 9. The user defined system states are defined in Listing 3.

As explained in Section 3.3, a movement request can be altered by the controller. Listing 7 provides four example rules. The first rule uses the same *ReducedPerformanceCondition*. The *MaximumSpeed* is defined in Listing 4. The **safe** keyword

```
1  import "Capabilities.csl"
2  import "Configurations.csl"
3  import "UserDefinedSystemStates.msl"
4  import "UserDefinedActions.msl"
5  import "UserDefinedZones.msl"
6
7  Condition ReducedPerformanceCondition
       is
8    (NOT POSITION_IS_CALIBRATED) OR
9    (NOT
         ULTRASONIC_SENSOR_IS_CALIBRATED)
10
11 [RedPerf] WHILE
       ReducedPerformanceCondition
12   DO MaximumSpeed safe
13   APPLIES TO allMovements
14
15 [CollCond] WHILE
       COLLISION_SENSOR_IS_ACTIVE
16   DO QuickStopWithStopRelease
17   APPLIES TO (allMovements)=>
         difference(BackwardMove)
18
19 [InEdgeZone] WHILE Inside
       BEFORE_EDGE_OF_PLAYING_FIELD_ZONE
20   DO MaximumSpeed safe
21   APPLIES TO TURN_MOVEMENTS_1
22
23 [InEdgeZoneA] WHILE configuration
       ConfigurationA AND Inside
       ON_EDGE_OF_PLAYING_FIELD
24   DO NormalStop APPLIES TO
         ForwardMove
25 OTHERWISE [InEdgeZoneO]
26   DO QuickStop APPLIES TO ForwardMove
```

Listing 7. DSL example statements

alters the maximum speed for the involved movements to safe. The value for the safe speed is defined in Listing 6. The keyword **all movements** is changed to **allMovements** to prevent parser issues which we do not discuss here.

The *CollCond* rule uses the state *COLLISION_SENSOR_IS_ACTIVE* from Listing 3. In Listing 1 on line 9 **except** is used to exclude movements from the **all movements** keyword. In the modified language, we added support for set operations. The line in the new instance results in the same movements.

The *InEdgeZone* rule makes the controller alter the *SharpLeft*, *SharpRight*, *WideLeft* and *WideRight* movements to safe speed when the robot approaches the edge of the playing field. The involved movements are now packed in a constant *TURN_MOVEMENTS_1*. As described, *TURN_MOVEMENTS_2* would give the same behaviour. These constants can also be used in the previously described set operations. In the latter, the keyword **Before edge of playing field zone** is replaced by an user defined zone from Listing 2.

The *InEdgeZoneA* rule stops the *ForwardMove* movement when the robot is on the edge of the playing field. The main difference compared to Listing 1 is that with the support for a family of systems, it can be the case that one specific system instance requires different behaviour than the other system instances. In Listing 6, we can define a system configuration that can be used in the behavioural rules. We also added an **OTHERWISE** keyword to define the behaviour for all other system instances. The code generator will use the annotation *isConfigurationA()* (see line 5) from Listing 6 for this and generates an *else* for the otherwise. Hence, *ConfigurationA* will perform a *NormalStop* and all other system instances a *QuickStop* for the *ForwardMove* movement when the robot is on the edge of the playing field.

4.4 Divide Language into Multiple Languages

Before we started refactoring the language, we had one large grammar and code generator. To improve maintainability and support the re-use of languages and code generators, we divided the languages. Figure 5 gives an overview of our languages and how they relate to each other.

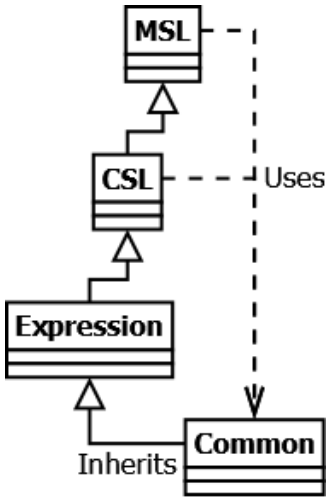

Figure 5. Relation of languages

Below we describe these languages in more detail.

Common. The common language is the root of all our languages. It defines the infrastructure concepts for all the

other language, e.g. it is used to import files into other files. Examples of importing files are in Listings 6 & 7.

Annotations are used to add documentation that can be made visible in the editor by a tooltip. In Listing 4 on line 3, we provided an example. When hovering with the mouse over line 24 of Listing 7, the documentation will be displayed. In addition, annotations are used to instruct the code generator, see Listing 6 line 5 for an example.

Expression. The Expression language provides features for numerical and boolean operations. It also provides support for operations on sets. This language inherits the Common language.

CSL. The Capability Specification Language (CSL) adds support for capabilities (e.g. movements) and configurations. Listing 5 and Listing 6 are instances of the CSL.

As can be seen in Figure 5, the CSL inherits from the Expression language and the CSL uses the Common language. In Listing 6 on line 5, there is an example of using the Common language in the CSL. This line provides guidance for the code generator to map *ConfigurationA* to function *isConfigurationA()*. In addition, on lines 19, 20 and 21 set operations of the Expression language are used to define constants.

MSL. Listing 7 is an example of the Movement Specification Language (MSL). The MSL inherits the CSL, and it makes use of the Common and the Expression languages.

An example of the usage of the importing mechanism of the Common language are the lines 1–5 of Listing 7. And lines 8 & 9 give an example of the use of the boolean operations of the Expression language inside the MSL. Constants defined in the CSL can used in the MSL, see for instance the definition of *TURN_MOVEMENTS_1* in Listing 6 line 20 and its use in Listing 7 line 21.

5 Industrial Embedding

Released products are installed in hospitals and need to be supported for more than a decade. Hence, our code needs to support these systems for a very long time. We use trunk-based development. For every product release, we create a branch in our code archive. Our code archive consists of several components. One of these components includes the grammar and code for the artifacts generator. From this component, we generate the plug-ins needed for the editor that is used by the users of the language. When a user starts the editor, it will automatically load the plug-ins from the archive. With this mechanism, the user always uses the version of the Azurion DSL that corresponds to that version of the archive. In addition, the version of the Azurion DSL is always aligned with the other code in the archive. The users of our language are approximately 40 engineers that use the language to create new functionality, solve field problem or add support for new hardware configurations. Before a change, a user takes the latest version of the archive for the product release he or she is working on. The user starts the editor that loads the current version of the plug-ins, starts changing the instance, and generates code when done editing. Azurion DSL users do not change or use the component that holds the grammar and code generator. The first two authors of this paper made the changes to the grammar and code generator described in this paper. They also update the plug-ins that are used by the users.

We made the changes described in Section 4 incrementally over the course of half an year. Before every change, we stored the generated artifacts on a separate location such that they could not be overwritten by a newly generated version. Next we changed the grammar and adapted the code generator for the changed grammar. Then we changed the instance to be compliant with the new grammar. The editor pointed us to the errors in the current instance. Last, we generated the artifacts and compare them with the stored artifacts using a diff tool. When equal, we had some confidence that we did not break anything. When unequal, the generator or instance were adapted to make them equal. In addition, to increase the confidence we also run all pre-delivery test cases. The changes were driven by requests of users and the current ideas we had on improving the DSL. There was not a predefined plan at the start. We took the following steps:

1. The zone definitions and movement definitions were taken out of the grammar and generator. With this change, the users could adapt and create zones and movements. After this change, the zone definitions and movement definitions were still part of the MSL.
2. In this step, we took out expressions from MSL. We created an Expression language and refactored the MSL to use the Expression language.
3. We moved movement definitions to a separate file and created the mechanism to import this file in an MSL instance that describes the behaviour of the movements.
4. Next we added support for constant variables in the file with movement descriptions.
5. In this step, we added max and safe speed values support as part of the movement definitions.
6. We took out the fixed action definitions from the grammar and made them user defined, analogue to the zones. We also introduced annotations for generator guidance.
7. Then we did the same for system states definition. After this step, these have to be specified by the user in a separate file.
8. We added configuration support. Configurations can be defined in a separate file. These can be imported and used in the MSL.
9. Last we refactored out the CSL. The grammar for movements, constant variables and configuration, was placed in the CSL and the MSL was adapted to use the

CSL. In this step, we also created the Common language. The import and annotation features were taken out of the MSL and placed in the Common language. The CSL and MSL were changed to use the Common language.

Splitting up the languages has enabled their reusability in other (related) domains. Common, Expression and CSL are now the basis of other languages used for priority of movement definitions and testing specifications.

Domain experts no longer need DSL experts to define new configuration with new movements, states and zones. We now have a clear distinction between the users of the DSL and maintainers of the DSL.

6 Concluding Remarks

We have co-evolved a five year old DSL together with its domain. The DSL was created to describe the behaviour of a controller used in an interventional X-ray system. Initially, this DSL was created for a single hardware configuration of the system.

We incrementally changed the language to support a family of systems, supporting multiple hardware configurations. In the old language, the hardware capabilites of the single hardware configuration were fixed in the grammar. To support a family of systems, the fixed language features (such as system states and zones) were removed and replaced in the grammar to be user defined. Regardless on how future system instances look like, the required concepts can be user defined now.

In addition to the grammar improvements, we also enhanced the architecture by splitting up the grammar over multiple languages. This architecture allows for the re-use of lower-level languages for DSLs that cover other domains or parts of our systems.

With the described evolution, we are confident that our DSL will have a bright future in our industrial setting.

Limitations. After evolutionary step in which the grammar changed, the language instances were changed manually. We could do this because of the limited amount of instances and the size of these instances. Our approach of having a grammar using Xtext and a generated meta-model does not really suit automated model-to-model transformations.

Changing to an approach that start with a meta-model would mean a full reimplementation of the language, which would definitely be more work than the evolutionary steps we took as described in this paper.

We could only gain confidence in our evolutionary steps by comparison of the stored and newly generated artifacts. There was no existing test suite available for testing the code generator we could use.

Future work. After the work presented in this paper, we want to improve early feedback to the user of the Azurion DSL. We think that with static checks on the model, we can provide useful feedback to the user of the language before the artifacts are generated.

We also would like to extend the CSL to describe more capabilities of the system and extend its use in the product code.

References

[1] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd, 2013.

[2] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, 2008.

[3] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, pages 35–51. Springer Berlin Heidelberg, 2009.

[4] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, 2016.

[5] D. Di Ruscio, L. Iovino, and A. Pierantonio. What is needed for managing co-evolution in MDE? In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, IWMCP '11, pages 30–38. ACM, 2011.

[6] J.-M. Favre. Meta-model and model co-evolution within the 3D software space. In *Int. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA) at ICSM 2003*, 2003.

[7] R. Hebig, D. E. Khelladi, and R. Bendraou. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, 2017.

[8] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language evolution in practice: The history of GMF. In M. van den Brand, D. Gašević, and J. Gray, editors, *Software Language Engineering*, pages 3–22, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] J. Mengerink, R. Schiffelers, A. Serebrenik, and M. van den Brand. DSL/model co-evolution in industrial EMF-based MDSE ecosystems. In T. Mayerhofer, A. Pierantonio, B. Schätz, and D. Tamzalit, editors, *Models and Evolution*, CEUR workshop proceedings, pages 2–7, 2016.

[10] J.-P. Tolvanen and S. Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *International Conference on Software Product Lines*, pages 198–209. Springer, 2005.

[11] L. Tratt. Evolving a DSL implementation. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007*, pages 425–441. Springer Berlin Heidelberg, 2008.

[12] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[13] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2013.