Communication Papers of the of the 17th Conference on Computer Science and Intelligence Systems pp. 339–348
DOI: 10.15439/2022F93
ISSN 2300-5963 ACSIS, Vol. 32

From MSL to Dezyne: an Industrial Application Of QVTo

Leo van Schooten, Marco Alonso, Ronald Wiericx
Philips in Best, The Netherlands
Email: leo.van.schooten@philips.com

Mathijs Schuts
Philips in Best, The Netherlands
& Radboud University
in Nijmegen, The Netherlands

***Abstract*—At Philips Image Guided Therapy (IGT), we have developed a Domain Specific Language (DSL) that describes the behaviour of one of the subsystems of our interventional X-ray system. With the current implementation of our DSL we are able to generate C++ code that is integrated in our product software. As a next evolutionary step for our DSL, we would like to benefit from the features the Dezyne toolset offers, like C++ code generation and model checking. If all model checks pass, we know that the generated C++ code is free of certain issues. We present a model to model transformation developed in QVTo, that transforms our own DSL called the Movement Specification Language (MSL) to another DSL called Dezyne. To avoid confidentiality issues, we use a Lego robot example to explain the MSL.**

I. Introduction

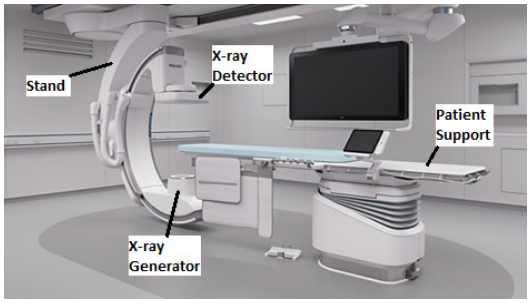

Fig. 1. Interventional X-ray system

AT Philips IGT, we develop and produce the interventional X-ray systems as shown in Figure 1. These systems are used for the diagnosis and treatment of mainly cardio and vascular diseases. One of the sub-systems is responsible for positioning the X-ray beam with respect to the patients body. A stand hanging on the ceiling holds an X-ray generator and an X-ray detector. To position the X-ray beam, the user of the system can initiate motorized movements of the stand using joysticks. Depending on the system state, these joystick requests are altered by a software component called the supervisory controller. Approximately six years ago we developed a Domain Specific Language (DSL) called: 'Movement Specification Language' (MSL). Language instances of the MSL called models describe the behaviour of the supervisory controller. To integrate a MSL model in our software, the instance is provided as input for a code generator that generates C++ code. The generated code can then be integrated in our software such that it behaves as described in the language instance.

Schuts et. al. [13] describe the evolution of the grammar of the MSL. In this paper, we describe the next evolutionary step we took to improve the MSL. We would like to replace the model to text C++ code generator by a model to model transformation. The target is a model in the Dezyne language [1]. Dezyne is a modelling language created by a Dutch company called Verum. Dezyne language instances can be checked for certain properties with the use of a model checker [2]. If all properties checks pass, C++ code can be generated with guaranteed equivalent behaviour as the checked model [2]. To be able to detect violations of certain properties in a MSL model and have C++ code that does not contain these violations we want to use the model checker and code generator provided by Dezyne.

The work described in this paper is executed by the first author of this paper during his internship at Philips. New in this paper, as compared to his master thesis [12], is that in this paper, we present and explain the QVTo instance needed for the model to model transformation.

The paper is organized as follows. We start with Section II about related work followed by Section III where we briefly describe the architecture of all the related languages and how they interact with each other. In Section IV, we introduce the case and grammar. The case will be a running example throughout this paper. Next in Section V we explain Dezyne and how the MSL example instance of Section IV maps to a Dezyne instance. Section VI describes how we have automated the transformation from MSL to Dezyne. Then we present the results in Section VII. We conclude our paper in Section VIII.

II. Related Work

The DSL that is developed at Philips IGT and described in this paper is informally described in [13] and formally described in [12]. Furthermore the Dezyne toolset is the successor of another toolset developed by Verum called ASD [3]. At Philips ASD has been successfully applied and evaluated as a valuable tool for software development. ASD, like Dezyne, is a lightweight formal software development tool. The toolset

[1]https://verum.com/discover-dezyne/

can be utilized to define and generate C++ code from finite state-machine models. Furthermore, the toolset is capable of detecting defects in the finite state-machines by means of model checking. Case studies of applying ASD at Philips IGT can be found in [5] and [11]. In both cases, the software contained less defects than when developing the software using conventional techniques of software development. Furthermore, both cases conclude that the overall development process was more efficient as time was spared because less tests had to be developed as some defects could be detected using the model checker of ASD.

The Dezyne toolset consists of a modelling language, a code generator and capabilities for model checking. In [2] the Dezyne language features are described and the transformation from the Dezyne modelling language to another modelling language called mCRL2 [6] is described. The mCRL2 models outputted by the transformation are used by the mCRL2 toolset [4] for model checking the behaviour of the Dezyne models.

Model to model transformations have been successfully applied in several occasions in the industry. For example, in [8] the authors describe a model to model transformation that was implemented at Cern[2] using the Asf+Sdf toolset [17]. The authors define a model to model transformation from a state machine DSL developed at Cern to the mCRL2 modelling language. This allowed the DSL instances from Cern to be model checked using the mCRL2 toolset and thus finding defects earlier in the development process. Furthermore, at Philips IGT a model to model transformation has successfully been applied in order to refactor state machine models from the Rhapsody [7] language to Dezyne [14]. This model to model transformation has been positively evaluated by comparing two approaches, manual transforming the models in the Rhapsody language to Dezyne and automatically transforming the Rhapsody models to Dezyne. The authors take into account the time it takes to fully implement and test a model to model transformation and the time it takes to do the transformation manually. The conclusion of the paper is that automating the transformation is a less time consuming process than doing the transformation manually.

As model transformations and, both toolsets from Verum: ASD and Dezyne have successfully been applied and positively evaluated at Philips we would like to continue on this path and therefore develop a model to model transformation that generates Dezyne language instances.

Finally, the QVTo language is an extension to the QVT language, the QVT language is described in [9] with the addition of operational mappings which is called QVTo. The extension QVTo adds new common imperative language constructs to the QVT language such as loops and conditions. Furthermore, the authors describe scenarios on which QVT can be utilized and which not. The authors discuss that the language can be seen as a general purpose model transformation language and suitable for a variety of model transformation problems. In addition they describe that the QVT language is less suitable for data transformation problems. Finally, in [16] a foundation is laid to describe QVTo transformations in a mathematical format for documentation purposes. As there are cases where the QVTo language has been evaluated for different purposes we present a practical application of QVTo in this paper.

[2]https://home.cern/

III. Model to Model Transformation Architecture

In order to be able to benefit from the advantages of Dezyne, we use Query/View/Transformation operational (QVTo) [9], defined by the Object Management Group (OMG) [10], as a language for describing a model to model transformation.

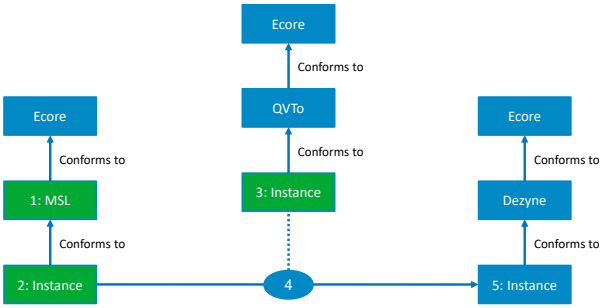

Fig. 2. Model to model transformation overview

Figure 2 provides an overview of the different languages we created and used. We make use of the Eclipse Modeling Framework (EMF) [1]. On the meta-meta-level we have Ecore, part of EMF, which is essentially a language to describe languages. The languages we use are described in Ecore. In Figure 2, all green boxes are language concepts that we developed and all blue boxes existed or are generated.

Our work consists of the following steps:

Step 1 In this step, we created a grammar for the MSL. The specifics of this language are described in [13].

Step 2 Software engineers create and alter an MSL instance to describe the behaviour of the supervisory controller.

Step 3 In this paper, we focus on the QVTo language instance we created.

Step 4 This step automatically generates Dezyne models. For the generation, a MSL instance and a QVTo instance are taken as input.

Step 5 We now have a generated Dezyne instances. Properties of the instances are tested using a model checker. If all checks pass, we generate C++ code with Dezyne and are done. If, however, there are failing properties, we have to go back to **Step 2** and change the language instance after which **Steps 4 & 5** have to be executed again.

IV. Movement Specification Language

First in order to understand the paper we will introduce the Movement Specification Language (MSL) as far as needed.

Because of confidentiality, we describe the MSL using a Lego rover example. The described example is inspired by an earlier paper we wrote about the evolution of the MSL's grammar [13].

We start this section with an introduction of the Lego Mars rover. Next we explain what Mars looks like. Then we describe how the Lego Mars rover is controlled. Last we provide an MSL example instance.

A. Rover

The Lego Mars rover has to accomplish missions on Mars Its mission is to search for water.

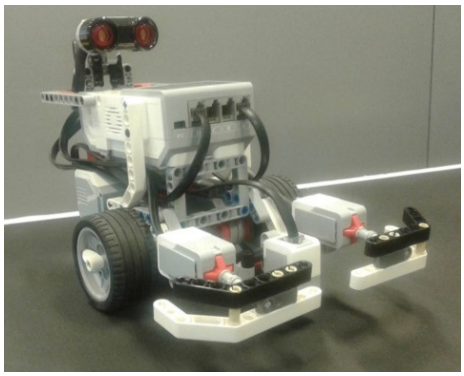

Fig. 3. Lego Mars rover

The Lego Mars rover is depicted in Figure 3. The rover has two big wheels at the front that are powered by two individual motors and the rover has a wheel at the back. Two bumpers at the front can detect if the rover has collided with an object. Between the bumpers there is a colour sensor that is used to look at the surface of Mars. At the back and on top of the rover an ultrasonic sensor looks forward.

The Lego Mars rover is remotely controlled. With joysticks the following movements can be requested: moving forward, moving backward, turn left and turn right. For all movements, the analog deflection of the joystick indicates the desired movement speed.

B. Playing Field

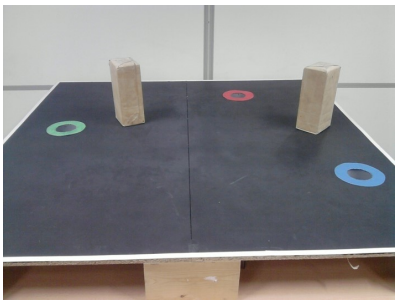

Fig. 4. Mars

For Mars we use Figure 4. Mars has the following characteristics:

- Mars has a square and flat surface.
- The surface of Mars is black and the edge has a white line.
- On the surface there are coloured lakes.
- There can be rocks on Mars.

The colour sensor is used to detect the edges of Mars. It is used to prevent driving the rover over the edge. The colour sensors can also be used to detect the colour of the lakes. Rocks can be seen at a distance using the ultrasonic sensor. Collisions with rocks are detected by the bumpers.

C. Controller

The controller of the Lego Mars rover is responsible for altering movement request that come from the remote control. The controller could alter a request based on the following examples:

1) *State:* The rover can be in a certain state for example, when the bumpers are active or when a sensor is defect the rover can reduce the speed of movement or even stop.
2) *Position:* The rover can deduce its position on the playing field. Depending on the on the position the rover can either reduce or increase its speed. For example, it can detect when it is near a lake or on the surface of Mars.
3) *Sensor value:* Sensor values of the rover can provide input that indicate that the rover should stop or reduce its speed. For example, when the collision sensor detects the rover is nearing a collision the rover should reduce its speed.

Note that: if none of the inputs trigger a reason to alter the movement request then the movement request will be executed unchanged.

D. Movement Specification Language

The DSL we describe in this section is used to describe the behaviour of our controller. We created a DSL we call MSL using the Xtext [1] Eclipse plug-in. First a few requirements are specified for the controller followed by an example of an MSL instance that implements these requirements. After that we argue that some of these requirements might lead to conflicting behaviour and how this is reflected in the MSL instance.

First consider the following requirements for our MSL instance:

1) Given the Lego Mars rover is driving on the surface of Mars then the rover should move with at most *max* speed.
2) Given the Lego Mars rover is driving near a coloured lake then the rover should move with at most a *reduced* speed.
3) Given the bumpers detect a collision then the rover should disable the ultra sonic sensor.
4) Given the bumpers detect a collision then the rover should stop immediately.

```
1 const DrivingMovements = Forward +
      Backward + TurnLeft + TurnRight
2 const ForwardMovements = Forward +
      TurnLeft + TurnRight
3
4 WHILE Inside LAKE_AREA DO MaximumSpeed
      safe APPLIES TO ForwardMovements
5 WHILE Inside MARS_AREA DO MaximumSpeed
      max APPLIES TO DrivingMovements
6 WHILE inState BUMPERS_ACTIVE In
      ULTRA_SONIC_SENSOR is disabled
      APPLIES TO allMovements
7 In ULTRA_SONIC_SENSOR, BUMPERS_SENSOR
      is enabled APPLIES TO allMovements
8 WHILE inState BUMPERS_ACTIVE DO
      QuickStop APPLIES TO allMovements
9 WHILE inState ULTRASONIC_ACTIVE DO
      NormalStop APPLIES TO allMovements
```

Listing 1. New proposal

5) Given the ultra sonic sensors detect a rock then the rover should gradually reduce speed and eventually stop.

Requirements 1 and 2 are requirements for the movement speed of the Lego Mars rover. On the flat surface of Lego Mars the rover is allowed to move with maximum speed as its collision prevention sensors are capable of preventing potential harm. However, the rover should move with at most a reduced safe speed whenever it is near a coloured lake as it is incapable of detecting when it could potentially ride into the water and thus get stuck in the water. Furthermore, requirement 3 ensures that the rover only relies on the bumper sensors when it is in a collision that the ultrasonic sensor is not able to detect. For example, when the obstacle's height is below the vision of the ultrasonic sensor. Then, the bumpers detect a collision and the ultrasonic sensor does not. Finally, requirements 4-5 are requirements that prevent the rover from colliding into obstacles. For requirement 4, the rover should stop immediately as it is already colliding with an obstacle when the bumpers detect a collision. However, when the ultrasonic sensor detects a collision, the rover has still some room to move forward. Hence, it may stop in a more gradual manner.

In Listing 1 an example MSL instance of the Lego rover is shown. It is used as a running example throughout the paper. The example instance in Listing 1 describes requirements 1-5. All the words in bold and blue are keywords i.e. part of the MSL's grammar. As explained in Section IV-C, a movement request can be altered by the controller. Lines 4-9 provide six examples of how the movement requests are altered.

First we will discuss some language concepts that help understand how the requirements are implemented. Consider line 4, the line contains a *condition*, *action* and a *set of movements*. The *condition* is denoted after the **WHILE** keyword, the *action* is denoted after the **DO** keyword and the applicable *set of movements* is denoted after the **APPLIES TO** keywords. Conditions can be considered as logical propositions and thus can be evaluated to *true* or *false*. The *actions* in the MSL instance are references to handwritten C++ classes that implement a function that takes as input a movement request and outputs an altered movement request. A typical line in the MSL instance can be interpreted as: when the *condition* is *true*, execute the *action* for movement requests related to movements in the *set of movements*. Furthermore, there are also *actions* related to enabling or disabling certain sensor inputs. Examples of such *actions* are shown on lines 6-7. These lines describe when the rover should consider or ignore the input values from certain sensors.

Recall Mars as shown in Figure 4. Line 4 describes that when a movement is requested and the rover is inside one of the lake areas of Mars the rover will move with at most a reduced *'safe'* speed. This line is applicable to all the movement requests related to driving forward i.e. moving forward, turn left and turn right. This *set of movements* is also defined in the language instance itself as shown on line 2. The line defines a *set of movements* called `ForwardMovements` to be the union of three movements being: `Forward`, `TurnLeft` and `TurnRight`. In addition, line 5 describes similar behaviour to line 4 the difference is that the rover moves at maximum speed instead of a reduced speed. These two lines essentially cover requirements 1 and 2. Next, lines 6-9 implement the behaviour required for requirements 3-5. Line 6 specifies that when the bumper sensor is active, the input from the ultrasonic sensor is disabled for all of the available movements. Furthermore, in other cases the ultrasonic sensor should be enabled and this is specified by line 7. These two lines implement the behaviour of requirement 3. In addition line 8 covers requirement 4. Line 8 states that when the bumper sensor is reporting a collision the movement should stop immediately which is represented by *action* `QuickStop`. Finally, line 9 covers requirement 5 which can be interpreted in a similar fashion as line 8, the difference being that the *condition* depends on the ultrasonic sensor input and the *action* representing a gradual stop.

E. Verification properties MSL

With the example in Listing 1 we will outline some properties that we would like to have in our MSL instance.

Recall that in a MSL instance the maximum speed of a movement can be set depending on the system state. An undesirable scenario is an instance that can set conflicting speeds for one specific movement. For example, in the MSL a developer can specify two lines where one line specifies that a movement should move with safe reduced speed while another specifies that a movement should move with maximum speed. If the *conditions* of those two lines exclude one another there is no problem i.e. the conditions can never both evaluate to *true*. However, if the two *conditions* do not exclude each other there might be cases where the speed is first set to the maximum value and then to a reduced value or vice versa. We would like to prevent such specifications as this might lead to undesirable behaviour i.e. the rover damaging itself. Hence, a

MSL instance should be free of ambiguous specifications for speed values. From now on we will refer to the property as *no conflicting speeds*.

The next property we would like our MSL model to have is that the specification is unable to simultaneously enable and disable the same sensor input. Such a specification might also lead to undesirable behaviour as it will or will not ignore certain sensor input while it is required to do so. This might occur when a developer specifies two lines where one line enables the input of a sensor and the other disables the input of a sensor and similar to property *no conflicting speeds* both conditions do not exclude each other. In addition to the *no conflicting speeds* property we would also like a MSL instance to be free of ambiguous specification where a sensor input can simultaneously be turned on and off. From now on we will refer to this property as *no ambiguous sensor status*.

V. Dezyne

In order to understand the translation we will as far as required describe the Dezyne modelling language, its features and give a small example. After that we will provide an outline to the implemented model to model transformation from the MSL to Dezyne. Dezyne language instances are called models and from now on we will refer to Dezyne languages instances as Dezyne models.

A. Dezyne toolset

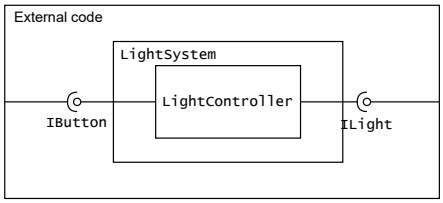

Fig. 5. System model of light system example

We start with an introduction to the Dezyne modelling language and the features of the Dezyne toolset. The Dezyne modelling language is a DSL with the purpose of designing state machine models. Dezyne models define three main concepts: interfaces, components and systems. We will explain the different types of Dezyne models by means of an example. Consider a simple light switch system where a light can be turned on and turned off by a button press. In addition the light has a requirement that whenever the light is turned on it is not allowed to turn the light on again and whenever the light is turned off it is not allowed to turn the light off again. The structure of the light switch system is depicted in Figure 5. The outside rectangle represents the external code while the inside rectangles represent a Dezyne system and component model `LightSystem` and `LightController` respectively. An arc at the end of a connection means that the Dezyne interface is required by the rectangle that it is connected to and a circle at the end of a connection means that the Dezyne interface is provided by the rectangle that it is connected to. We will now continue to zoom into these Dezyne concepts.

```
1  interface IButton {
2   in void buttonPress();
3   behaviour {
4     on buttonPress: { }
5   }
6  }
```

Listing 2. Example Dezyne interface model `IButton`

```
1  interface ILight {
2    in void turnOn();
3    in void turnOff();
4    enum State {On, Off};
5
6    behaviour {
7      State state = State.Off;
8      [state.On] on turnOff:{state = State.Off;}
9      [state.Off] on turnOff: illegal;
10     [state.On] on turnOn: illegal;
11     [state.Off] on turnOn:{state = State.On;}
12   }
13 }
```

Listing 3. Example Dezyne interface model `ILight`

1) Dezyne interface models: First, interfaces in Dezyne models describe a specification of deterministic stateful behaviour. The description of behaviour in the interface models is implemented by Dezyne component models or external C++ code. The interface models can define input events. Input events can be used as input for the implementation of a Dezyne interface. Interfaces can also define output events which occur as a consequence of certain behaviour and can be input for other implementations of the same interface. However, output events are not utilized by our Dezyne translation. Hence, we will further omit them. Furthermore, interface models can also define when certain behaviour is *illegal* meaning that the implementation is not allowed to trigger certain input events in certain states. For example, consider the interface model defined in Listing 2 for the light switch system. The model defines on line 2 one input event that corresponds to the button being pressed. The second interface model called `ILight` defined in Listing 3 defines two input events and a state type to keep track of the system's state i.e. whether the light is turned on or turned off. Furthermore, the interface specifies for each state what is allowed and what is not allowed. For example, on line 8 the interface specifies that whenever the interface is in the `On` state a `turnOff` event is allowed, while on line 9 the interface specifies that the `turnOff` event is not allowed by specifying it with the `illegal` keyword.

```
1  component LightController {
2    provides IButton provided;
3    requires ILight required;
4    behaviour {
5      ILight.State state = ILight.State.Off;
6      on provided.buttonPress(): {
7        if (state == ILight.State.Off) {
8          required.turnOn();
9          state = ILight.State.On;
10       } else if (state == ILight.State.On) {
11         required.turnOff();
12         state = ILight.State.Off;
13       }
14     }
15   }
16 }
```

Listing 4. Example Dezyne Component model `LightController`

2) Dezyne component models: The second type of models in Dezyne are component models. These component models implement the actual behaviour described by the interface models. Component models can either require or provide an interface model. The models need to adhere to the state

behaviour that is described by their provided and required interfaces. When a component requires an interface it means that the component can trigger the inputs of the interface. Furthermore, when a component provides the interface it means that the component provides the implementation of the interface i.e. the component defines what the behaviour is if certain inputs are triggered. The component model defined in Listing 4 implements the behaviour for the required interface i.e. interface `ILight`. On lines 6-14 the component model defines the behaviour of the input event `buttonPress`. The implementation triggers the input events on the required `ILight` interface according to the specification of the interface. The component ensures the behaviour of the interface by keeping track of the state and using the if else statement on lines 7 and 10 to evaluate the system state and call the correct input event on the provided interface.

```
1  component LightSystem {
2    provides IButton provided_port;
3    requires ILight required_port;
4
5    system {
6      LightController controller;
7      provided_port <=> controller.provided;
8      controller.required <=> required_port;
9    }
10 }
```

Listing 5. Example Dezyne system model `LightSystem`

3) Dezyne system models: The third type of Dezyne models are the system models. These models describe the composition of the components and interface models i.e. the models describe the structure depicted in Figure 5. Meaning that these models specify what interface is connected to what component and what interfaces are provided and required to the software that is integrating the Dezyne models. External C++ code is supposed to provide the implementation of turning the light on and off and trigger the input event on the `IButton` interface when the button is pressed. Furthermore, the Dezyne models implement the control behaviour of the light switch system. The Dezyne model that represents this structure is depicted in Listing 5. Observe that the `LightSystem` is also a component model that provides and requires interfaces. The required interfaces of the `LightSystem` will be provided by the external code. On line 6 the model states that the system consists of one component model being the `LightController`. On line 7 the model connects the implementation of the provided interface to that of the `LightController` and on line 8 the model connects the required interface of the `LightController` to the external code.

The semantics of the Dezyne models from Listings 2-5 implement the run to completion semantics [15]. The run to completion semantics imply that when a Dezyne component calls an input event on an interface the component gets blocked and awaits the result of the event that was called. These semantics are demonstrated in Figure 6. Observe that on the component the `buttonPress` event is called, as a consequence of the input event, component `LightController` calls the `turnOn` event on the required interface. Now observe that both the component and the initial caller on the component are blocked until the required interface returns.

The next feature Dezyne offers is formal verification of the behaviour that is expressed by a Dezyne model with the help of the mCRL2 model checker [2]. The general formal verification properties that Dezyne can verify are:

- Deadlock and livelock i.e. can the specification in a Dezyne model get stuck and thus no longer progress.
- Compliance i.e. does the component implement equivalent state behaviour as the provided interface description i.e. does the component not trigger illegal behaviour.
- Deterministic i.e. does a component or interface specification implement deterministic behaviour.

Finally, Dezyne models can be used as input for a C++ code generator that outputs code that is equivalent in behavior to the mCRL2 model. Whenever the model checker does not find any violations in a Dezyne model, C++ code can be generated and integrated in a system. In the case of the light switch system, the model checker found no violations which, implies that the implementation in component `LightController` is not able to trigger a `turnOn` event when the light is turned on or `turnOff` event when the light is turend off.

B. Translation outline

In this section we will describe the outline of the Dezyne models outputted by the model to model transformation. In Figure 7 an outline of all the Dezyne models generated for movement `TurnLeft` is shown. The rectangles describe component models and the connectors between the rectangles describe the interface models. An arc at the end of a connection means that the component requires the mentioned interface and a circle at the end of a connection means that the component provides the mentioned interface.

The main idea of the translation is that each distinct *action-movement* pair has its own component. Figure 7 shows the decomposition for the `TurnLeft` movement. Observe that the `TurnLeft` movement shows two components that have behaviour for an *action* i.e. component `TurnLeftUltraSonicSenor` and component `TurnLeftMaximumSpeed`. Component `TurnLeftUltraSonicSenor` implements the behaviour

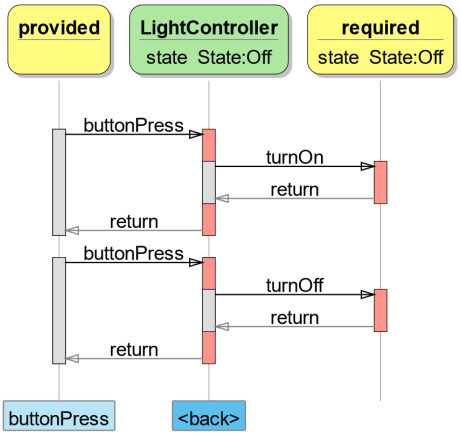

Fig. 6. Semantics Example model

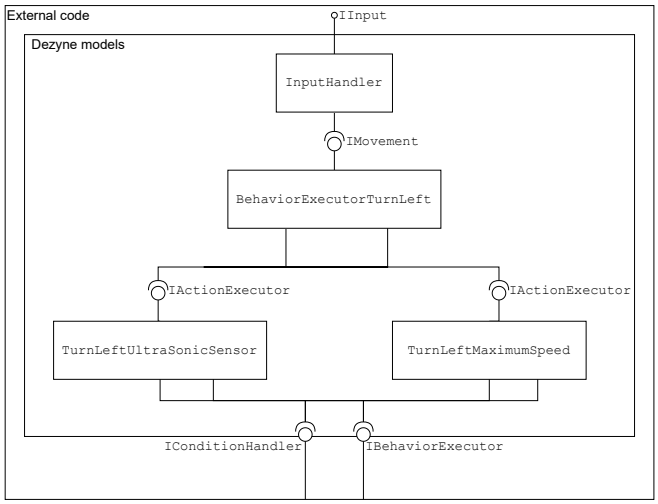

Fig. 7. Translation outline

for enabling and disabling the ultrasonic sensor and component `TurnLeftMaximumSpeed` implements the behaviour for setting the maximum speed of the movement. Recall, from the running example in Listing 1 that for movement `TurnLeft`, a `QuickStop` *action* and a `NormalStop` *action* are also defined. Our translation would also generate component models for these two actions. However, we have omitted these from Figure 7. Furthermore, the other components `BehaviorExecutorTurnLeft` and `InputHandler`, process the input coming from manually written components in the system. Movement requests are provided as input through the `IInput` interface. Depending on the request the `InputHandler` will forward the movement request to a component that maintains the behaviour for a specific movement. For example, in the case of a movement request for the `TurnLeft` movement `InputHandler` will forward the request to component `BehaviourExecutorTurnLeft` which will then forward the input to the relevant *action* components i.e. `TurnLeftUltraSonicSenor` and `TurnLeftMaximumSpeed`.

VI. QVTo

In this section, we will describe how we utilized QVTo to implement our model to model transformation. First we will provide a Dezyne model that shows how we want a generated model to look like using the model to model transformation. Second we will describe the QVTo language constructs and see how the translation is implemented.

A. Dezyne target model

```
1  component TurnLeftUltraSonicSensor {
2      requires IConditionHandler iConditionHandler;
3      requires IBehaviourExecutor iBehaviourExecutor;
4      provides IActionExecutor iActionExecutor;
5      behaviour {
6          enum State {enabled, notSet, disabled};
7          State state = State.notSet;
8          on iActionExecutor.executeBehaviours(mc): {
9              state = State.notSet;
10             bool bumpersActive = iConditionHandler.BumpersActive();
11             if (bumpersActive) {
12                 iBehaviourExecutor.Disabled(mc, UltraSonicSensor);
13                 if (!(state == State.disabled  state == State.notSet)) illegal;
14                 state = State.disabled;
15             }
16             if (true) {
17                 iBehaviourExecutor.Enabled(mc, UltraSonicSensor);
18                 if (!(state == State.enabled  state == State.notSet)) illegal;
19                 state = State.enabled;
20             }
21         }
22     }
23 }
```

Listing 6. Component specification `TurnLefUltraSonicSensort`

The Dezyne component models that are generated all have a similar structure. Listing 6 shows an example of a Dezyne model generated by our model to model transformation. The model implements the behaviour of turning the ultrasonic sensor on and off for movement `TurnLeft` as described in our running example in Listing 1. Observe that the model requires two interfaces being `IConditionHandler` and `IBehaviourExecutor`, and it provides the `IActionExecutor` interface. The required interfaces are used to evaluate *conditions* and to execute *actions*. The provided interface is there to provide the input to the Dezyne model. In the case of the model in Listing 6 the input consists of movement requests for movement `TurnLeft`. Furthermore, observe that line 11 implements the *condition* on line 6 of Listing 1. In addition line 16 implements the absent condition on line 6 of Listing 1. Hence, it is simply translated as *true*. Finally in order to verify that the *no ambiguous sensor status* property holds, the Dezyne model keeps track of the state of the sensor and triggers an *illegal* event when conflicting values for the sensor state are set. This is implemented using the if-statements on lines 13 and 18.

B. QVTo language

We will introduce QVTo as far as needed to understand this paper. The first main concept QVTo defines are the source and the target language i.e. the language that is used as input and the language that the QVTo transformation should output.

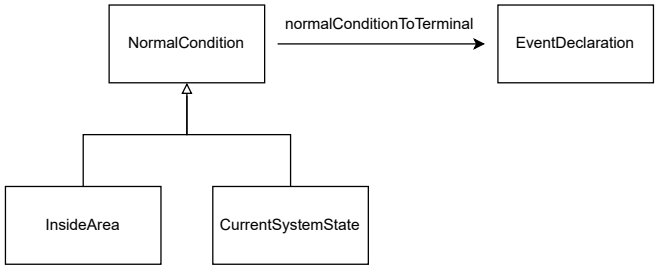

Fig. 8. Description of mappings

The second QVTo concept that we will discuss are mappings. Mappings transform a language element from the source language to a language element in the target language. Mappings are one of the main driving concepts in the QVTo language. For example, consider the *conditions* of the MSL. As mentioned earlier our MSL language conforms to the Ecore language on the meta model level. The *conditions*

```
1  mapping msl::NormalCondition::normalConditionToTerminal(...),
2  disjuncts
3  msl::InsideArea::InsideAreaToTerminal,
4  msl::CurrentSystemState::systemStateToTerminal: dzn::EventDeclaration {
5
6  }
7
8  mapping msl::InsideArea::InsideAreaToTerminal(...) : dzn::EventDeclaration {
9      name := "inside" + self.object3D.name;
10     dir := dzn::EventDirection::IN;
11     typeName := new BoolType();
12 }
13
14 mapping msl::CurrentSystemState::systemStateToTerminal(...) : dzn::
       EventDeclaration {
15     name := "inState" + self.currentState.name;
16     dir := dzn::EventDirection::IN;
17     typeName := new BoolType();
18 }
```

Listing 7. Disjunct mapping normalConditionToTerminal

```
1  query isMovementInSet(movementName : String, set : OrderedSet(csl::AMovement))
       : Boolean {
2      return set->selectOne(s | s.name = movementName) != null;
3  }
```

Listing 8. Query for movement set

are described using an Ecore class called 'NormalCondition'. This class has two subtypes called 'InsideArea' and 'CurrentSystemState'. This class decomposition is shown in Figure 8, the composition can be interpreted similar to super types and subtypes in object-oriented programming languages. QVTo mappings allow us to convert object instances of the NormalCondition type to object instances of the target Dezyne language.

Listing 7 shows an example of a mapping in QVTo. This mapping defines how objects of the 'NormalCondition' type should be transformed to 'EventDeclaration' type of Dezyne i.e. it defines events for the evaluation of system states on the `ICondtionHandler` interface. On line 1 a mapping for the 'NormalCondition' type is defined. Note that the mapping first states the input type i.e. 'NormalCondition'. Then observe on lines 2-4 that the mapping defines mappings for its subtypes 'InsideArea' and 'CurrentSystemState'. Finally, on line 4 the mapping also defines the output type of the mapping which is a type of the target language. In this case, the output type is an 'EventDeclaration'. The mappings for the subtypes are defined on lines 8-18. Typically a mapping states what values class member variables of the target class should have. For example on lines 9-11 it is shown that the EventDeclaration that is the result of the mapping 'InsideAreaToTerminal' states what name the event should have, what the direction of the event is i.e. ingoing or outgoing and finally what the type of the event is i.e. boolean.

The next language concept in QVTo are queries. Queries are typically used to retrieve information from elements in the source language. Furthermore, queries can also be utilized to return static elements of the target language. An example of a query is shown in Listing 8, the query determines if a certain movement is in a certain *set of movements*. Observe that the query is implemented similar to lambda expressions in modern object-oriented programming languages.

Furthermore, QVTo defines the notion of intermediate classes. Elements from the source language can be mapped

```
1  intermediate class BehaviourStatement {
2      name:String;
3      condition:intermediateCondition;
4      action:msl::GeneralDoAction;
5      setOfMovements:MovementSet;
6  };
```

Listing 9. Intermediate class

```
1  mapping msl::GeneralExecutionStatement::ExecutionStatementToBehaviourStatement
       (...) : BehaviourStatement {
2    name := self.name;
3    if(self.conditions != null) {
4      condition := new intermediateBoolExprCondition(self.conditions);
5    } else {
6      condition := new intermediateNoCondition();
7    };
8    action := self.actionGen;
9    setOfMovements := self.setOfMovements.xmap SetOfMovementsToSet(...);
10 };
```

Listing 10. Intermediate class mapping

to intermediate classes before they are mapped to elements of the target language. This can be beneficial as not always elements from the source language can be one to one mapped to elements in the target language. Listing 9 shows an example of a intermediate class, it describes an abstraction of a behaviour in the MSL language. Recall from the instance in Listing 1 that a line in the MSL can describe a behaviour that consists of a *condition*, *action* and an applicable *set of movements*. This intermediate class is meant to be used to extract this relevant information from a MSL model. Observe that the class also makes use of intermediate classes such 'intermediateCondition' and 'MovementSet'.

An example of how intermediate classes can be used in a QVTo mapping is shown in listing 10. The Listing defines a mapping where the language element 'GeneralExecutionStatement' gets mapped to the intermediate class 'BehaviourStatement'. The 'GeneralExecutionStatement' is the class defined in the source language that describes typical MSL statements such as the ones in the example in Listing 1. This mapping is convenient as it determines the correct condition for a behaviour statement, the correct *set of movements* and the corresponding *action*. Observe that on line 9 in Listing 10 another mapping is called to extract the correct set of movements from a 'GeneralExecutionStatement'.

Finally, in order to generate the component model of Listing 6 the mapping from the 'GeneralExecutionStatement' to the intermediate class 'BehaviourStatement' is used and then another mapping called 'BehaviourStatementToComponents' is used to generate the Dezyne component. The implementation of this mapping is shown in Listing 11. Observe that in this mapping we generate a new component for each movement action pair for a 'GeneralExecutionStatement'. The loop on line 3 in Listing 11 iterates over the applicable *set of movements* for a 'BehaviourStatement'. Then on line 5 the earlier created Dezyne component is retrieved from a mapping called 'StatementToComponent'. This mechanism is called 'resolve' in QVTo and it allows a user to retrieve results from earlier mappings in the transformation. The resolve feature is very convenient in this case as we only want to create one component for each specific *movement*, *action* pair.

```
1  mapping BehaviourStatement::BehaviourStatementToComponents(...) {
2      var actionName := self.action.name;
3      self.setOfMovements.movements->forEach(move) {
4          var componentName := move.name + actionName;
5          var component := componentName.resolveoneIn(String::
               StatementToComponent);
6          if (component == null) {
7              component := componentName.xmap StatementToComponent(...);
8          };
9
10         var ifExpr := self.condtion.xmap inermediateConditionToIfExpression
               (...)
11         var behaviourExecutorPort := component.ports->(p | p.name = "
               iBehaviourExecutor").oclAsType(dzn::PortDeclaration);
12         var actionEvent := self.action.resolveoneIn(msl::GeneralDoAction::
               generalExecutionStatementToActionEvent);
13         var then := new CompoundBehaviourStatement();
14         then.stats += new ActionOrFunctionStatement(...);
15         then.stats += GenerateCheck(...);
16         var ifStatement := new IfStatement();
17         ifStatement.expr := ifExpr;
18         ifStatement.then := then;
19
20         component.behaviour.stats->first().oclAsType(dzn::OnEventStatement).
               stat.oclAsType(dzn::CompoundBehaviourStatement).stats +=
               ifStatement;
21     }
22 }
```

Listing 11. BehaviourStatement to component

```
1  transformation Msl2DZN(in source:genmsl, out target:dzn);
2
3  main() {
4    source.rootObjects()[genmsl::GENMSLModel].xmap MSL2DZN();
5  }
6
7  mapping genmsl::GENMSLModel::MSL2DZN() : List(dzn::ModelDeclarationList) {
8    result += self.xmap actionsToEvents();
9    result += self.xmap conditionsToConditionInterface();
10   result += self.xmap movementsToInterfaceAndComponents();
11   result += self.xmap behavioursToDezyne();
12 }
```

Listing 12. BehaviourStatement to component

Furthermore, observe that the if-statement on line 6 ensures that a new Dezyne component is created if it has not been mapped before. Next lines 13-20 implement the generation of the if-statement that corresponds to evaluating the condition and executing the action. First on line 13 it maps the condition to a Dezyne if expression. Then on lines 11-12 the QVTo mapping retrieves the *action* elements in the target language in order to build the body of the if-statement. Finally on lines 13-20 the if-statement along with its body is created and added to the component. In addition on line 15 a query is called that depending on whether the component checks the *no ambiguous speed* or *no ambiguous sensor* status property, generates the if statement that contributes to the verification i.e. the if-statements on lines 13 and 18. This mapping essentially generates the Dezyne component that is shown in Listing 6.

To fit all the mappings together the final mapping that QVTo requires is to define a transformation. Listing 12 shows the definition of a transformation and the main function associated with it. Line 1 states the definition of the transformation, specifying the source and the target language. Then on line 3 the main entrypoint for the transformation is defined where we specify that the objects in the source language are to be transformed by the mapping called 'MSL2DZN'. This main mapping is defined on line 7 which specifies that it returns a list of 'ModelDeclarationList' in the target language i.e. a list of component and interface models.

VII. Results

The developed model to model transformation extended the capabilities of the MSL with model checking capabilities and the capability to use the Dezyne C++ code generator that allows us to dismiss our C++ code generator for the MSL. We will discuss the results of applying the transformation in the running example and in a MSL model of the interventional X-ray system. First we will describe the violations of the model checking properties *no conflicting speeds* and *no ambiguous sensor status* in our running example. Then we describe the transformation outcome and model checking properties violations found on a MSL model integrated in the software of the interventional X-ray system.

A. Model checking violations in running example

In our running example in Listing 1 we deliberately put violations of the *no conflicting speeds* and *no ambiguous sensor status* properties. Both the *conditions* on lines 4 and 5 can evaluate to *true* this results in an ambiguous specification for the speed value. For example, when the Mars rover is on the border of the lake and the Mars area then both the conditions `Inside LAKE_AREA` and `Inside MARS_AREA` might both evaluate to *true*. This ambiguous specification is applicable to all of the movements in the intersection of both the specified *set of movements* in both lines i.e. {Forward, TurnLeft, TurnRight}.

Furthermore, there is also a violation of the *no ambiguous sensor status* property. We have specified that both sensor input should be enabled by default (line 7) while also the ultra sonic sensor input should be ignored when the bumper sensor is active (line 6). Hence, whenever the condition `inState BUMPERS_ACTIVE` evaluates to *true*, the Mars Rover's specification is ambiguous on whether or not it should consider or ignore the sensor input from the ultra sonic sensor. The violation of the ambiguous sensor status is applicable for all of the movements the system can perform as both of the applicable *set of movements* on lines 6 and 7 are all movements.

B. Model checking violations Philips IGT system

For the software of the interventional X-ray system a MSL model has been developed that describes the behaviour that is specified by the requirements. We have applied the transformation described in this paper to generate Dezyne models that were capable of verifying the properties *no conflicting speeds* and *no ambiguous sensor status*. The transformation generated approximately 1200 Dezyne models. The amount can be explained by the fact the transformation generates a new component model for each specific movement action pair in the MSL model. In the MSL model that is integrated in the X-ray system there are 55 movements and 20 *actions*. Hence, the maximum amount of models that our transformation could generate is 55*20=1100. Then for each specific movement there is also a Dezyne model generated with the addition of the interface and composition models. These models were mostly small in size and the model checker of Dezyne was

capable of verifying most of them. The model checker of Dezyne has found 4 violations of the *no conflicting speeds* property and 17 violations of the *no ambiguous sensor status* property. Not all of the violations were issues in the software because the order in the MSL instance happened to be such that the code generator of the MSL generated C++ code that did not implement behaviour that could cause an issue. In addition some of the found ambiguous specifications between two lines in the MSL were not ambiguous because although both conditions could be *true* at the same time in theory, in practice it concerned system states that could never be *true* at the same time. Unfortunately, there were 27 generated Dezyne models that had a very vast state-space, we did not explore the state-space of all of the models but one of the models had a state-space of approximately 160 million states. The Dezyne model checker was not able to verify these models in a acceptable time period, some models could take hours or longer to verify. Finally, the C++ code generated by Dezyne was integrated in the software of the interventional X-ray system. With the Dezyne code the regression test cases passed that every code change in the X-ray system's software need to pass. Hence, we have good confidence that the behaviour described by the MSL instance is preserved in the generated Dezyne models.

VIII. Concluding Remarks

To conclude we have implemented a model to model transformation from a DSL developed at Philips IGT called the MSL to another DSL developed by Verum called Dezyne. This transformation is the next step in the evolution of the MSL. The transformation to Dezyne can be utilized as an intermediate step between developing a MSL model and generating C++ code that can be integrated in our software. The additional benefit of the generated Dezyne models is that their behaviour can be model checked and thus, we can ensure that our MSL instance does not contain certain potential issues. In the MSL a developer can make certain specifications ambiguous by specifying multiple speed values that are both set in a specific system state. Furthermore, a developer can specify that certain sensor inputs should be ignored or considered. Specifying the consideration of a sensor input could also lead to ambiguous specifications as a developer could specify to both ignore and consider a certain sensor input in a specific system state. In order to prevent such ambiguous specifications the transformation generates Dezyne models that could detect these ambiguous specifications using model checking. In total the transformation generated approximately 1200 Dezyne models which, contained 21 ambiguous specifications. These models were generated from a MSL model integrated in the software of the interventional X-ray system.

There are some improvements that can be made using this approach. As mentioned in the previous section, the Dezyne models do not contain any domain specific knowledge, leading to flagging potential issues which in practice could never occur. Furthermore, there are some Dezyne models outputted by our transformation that have a state-space that is to vast for the model checker to output a result in a acceptable time period. Therefore, as future work we would like to solve the state-space explosion problem in addition we would like our Dezyne models to be more aware of domain specific concepts such as system states that exclude each other. Finally, another improvement we could make is seeing if there are more *actions* in the MSL that are tightly coupled to each other and could for example also lead to ambiguous specifications.

References

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

[2] R. V. Beusekom, J. F. Groote, P. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, and T. A. C. Willemse, "Formalising the dezyne modelling language in mcrl2," *Lecture Notes in Computer Science Critical Systems: Formal Methods and Automated Verification*, p. 217–233, 2017.

[3] G. H. Broadfoot and P. J. Hopcroft, "Analytical software design," *Verum Consultants BV*, 2003.

[4] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. D. Vink, W. Wesselink, and T. A. C. Willemse, "An overview of the mcrl2 toolset and its recent advances," *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, p. 199–213, 2013.

[5] J. F. Groote, A. Osaiweran, and J. Wesselius, "Analyzing the effects of formal methods on the development of industrial control software," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011, Williamsburg VA, USA, September 25-30, 2011)*. United States: Institute of Electrical and Electronics Engineers, 2011, pp. 467–472.

[6] J. F. Groote and M. R. Mousavi, *Modeling and analysis of communicating systems*. The MIT Press, 2014.

[7] D. Harel and H. Kugler, "The rhapsody semantics of statecharts (or, on the executable core of the uml)," in *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 2004, pp. 325–354.

[8] Y. L. Hwong, J. J. Keiren, V. J. Kusters, S. Leemans, and T. A. Willemse, "Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider," *Science of Computer Programming*, vol. 78, no. 12, pp. 2435–2452, 2013, special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011).

[9] I. Kurtev, "State of the art of qvt: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, M. Nagl, and A. Zündorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 377–393.

[10] OMG, "meta object facility (mof) 2.0 query/view/transformation specification_2008," Apr 2008.

[11] A. Osaiweran, M. Schuts, J. Hooman, J. F. Groote, and B. V. Rijnsoever, "Evaluating the effect of a lightweight formal technique in industry," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 1, p. 93–108, 2015.

[12] L. v. Schooten, "Extending a domain specific language using model transformations," Master's thesis, 2021.

[13] M. Schuts, M. Alonso, and J. Hooman, *Industrial Experiences with the Evolution of a DSL*. New York, NY, USA: Association for Computing Machinery, 2021, p. 21–30.

[14] M. Schuts, J. Hooman, and P. Tielemans, "Industrial experience with the migration of legacy models using a dsl," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, 02 2018, pp. 1–10.

[15] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Prentice Hall, 2015.

[16] U. Tikhonova and T. Willemse, "Designing and describing qvto model transformations," in *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, vol. 1, 2015, pp. 1–6.

[17] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser, "The asf+sdf meta-environment: A component-based language development environment," *Electronic Notes in Theoretical Computer Science*, vol. 44, no. 2, pp. 3–8, 2001, lDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).