RoboProse - an introductory textual programming language

RICHARD BÁNYI, RASMUS POMMER ANDERSEN, DAVIDE LAEZZA, and NIELS ØRBÆK CHRISTENSEN

1 INTRODUCTION

The tiny robot Thymio[1] is bundled with a series of different languages that users of different target groups can use to program it. These languages are intended to make it possible to learn programming by doing, even though you know nothing about it to begin with.

For the youngest audience users there is the extremely abstract and high-level "Visual Programming Language" or "VPL" (see Figure 1a). It does not contain numbers or text at all and only relies on icons and graphics and their ordering to describe the semantics of each program.

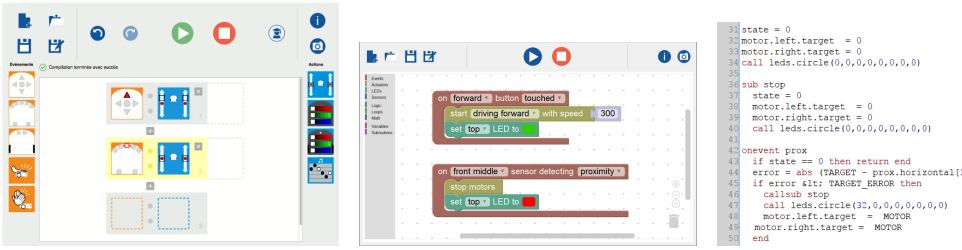
The second step on the language ladder is the Scratch-inspired[2] Blockly (see Figure 1b), which is another graphical language, where the user assembles different blocks by dragging them with the pointer. These different blocks are a series of actions contained inside events and possibly loops. These give an introduction to how textual programming languages often are structured, with expressions nested inside statements and these statements determining the flow of the program.

Thirdly, we have AsebaScript, the only textual language bundled with the Thymio (see Figure 1c). While still having some abstractions about the mechanics and handling of the robot, it is a much more low-level language than the other two. Plus, it introduces a completely new way of programming for the users: *Textual programming*, where you have to *write* everything and where the code fails if you do not follow the syntax correctly.

In this paper we will present RoboProse, a new language for the Thymio that is different from the ones already introduced. The idea is to create a language that lies much closer to natural language than the AsebaScript, but still follow a syntax that needs to be mastered in order to use it. One can say that it lies somewhere between the second and third language-step on this figurative ladder of programming languages.

[1] https://www.thymio.org/, retrieved 07/12/2017

[2] Scratch is a popular graphical programming language designed at MIT. https://scratch.mit.edu/about/, retrieved 07/12/2017



(a) Visual Programming Language (b) Blockly (c) AsebaScript

Fig. 1. The three programming languages bundled with the Thymio

2 PROBLEM ANALYSIS/PROBLEM STATEMENT

As stated in the introduction, our goal is to create a programming language for the Thymio that is textual, but not as low level as the AsebaScript. This should be a language that resembles English prose, while still adhering to certain rules and structures that need to be followed in order for it to be usable. The reasoning behind creating a language like this is to make something that can be used by people who do not have a programming background. These could be children of age 9-13 or any people who have never tried programming before.

The language should give a gentle introduction to certain aspects of programming, like learning a language syntax and getting accustomed to testing textual programs. It should still be kept simple, mostly excluding more advanced programing structures: Loops, conditional statements, variables, operators etc. We hope that this can help create a soft transition from the visual languages to AsebaScript.

3 ROBOPROSE BY EXAMPLE

In this section we will present and explain some simple examples of RoboProse programs in order to give the reader an overview of the structure, appearance, and capabilities of the language. All of these aspects will be handled in more detail in section 4.

The language we have implemented is as close as possible to a subset of natural English, while still keeping it usable as a programming language. We introduce some fixed sentence structures in order to keep the complexity of the final language low. One ambition of the language is that a person who has never done programming before should be able to read the code and understand how the robot will behave without any introduction.

As a starting point an example is given in Listing 1. Here the robot is instructed to first move backwards for 2 seconds, then turn and repeat both steps infinitely. Proceeding to a step-by-step analysis, the phrase `"My Robot should"` at the beginning of a program marks the following instructions as the starting point for the robot. Next, there need to be a list of one or more instructions, that in this case includes two actions: the first one is `"move backwards for 2 seconds"`, having the type `"move"`, a direction and a duration; the second one is is `"turn"`, with a direction and its specific degree parameter. A delimiter is used between each action to mark where the first one ends: in this example, we can show `", then"` and `"and then"`. All lists of actions can be followed by a single ending, determining the further flow of the program: in this case it is called `"repeat"`, and restarts the current list of instructions.

```
My robot should move backwards for 2 seconds, then turn left 35 degrees and
    ↪ then repeat.
```

Listing 1. A simple robot repeating two actions.

```
My robot should move forward.
When it meets an obstacle it should move backwards for 10 seconds and then turn
    ↪  randomly. Then it should start over.
```

Listing 2. A robot with en event listener, trying to avoid obstacles.

In Listing 2 we show a slightly more complex program. In this case an *event listener* has been added. The beginning of an event listener is defined by using the keywords `"When it"`. After this an *event* should be defined, which in this example is `"meets an obstacle"`, meaning that the

event will trigger whenever the robot encounters something in its way. This should be followed by `"it should"` and a list of actions and an optional ending similar to what we have already seen.

In Listing 3 a *sublistener* is introduced. This program will make the robot move forward and then stop when tapped. But when it is then tapped again it should turn 90 degrees and restart the program. A sublistener is delimited at the start with `"(but"` and at the end with `")"`. Structurally a sublistener is the same as a top level event listener, and it has all of the same parameters and options. The main difference is when the sublistener is *listening*. A sublistener is only triggered if the matching event is observed and its parent is currently executing. In Listing 3 the actions of the event listener have the ending `"wait"`, meaning that execution will halt there until another event listener is triggered. So, as long as the execution is stopped within the parent listener, the sublistener is active and listening for an `"is tapped"` event.

```
My robot should move forward.
When it is tapped it should stop and wait (but when it is tapped it should turn
    ↪ right 90 degrees and then start over).
```

Listing 3. A robot that stops when tapped, and starts when tapped again.

Since a sublistener has all of the options of an event listener, is also means that it can have sublisteners itself. Sublisteners can be nested arbitrarily deep, and this allows for different ways of utilizing them. In Listing 4 the sublisteners are nested in such a way that each time a new sublistener is triggered the one beneath is the next one in a sequence. In this example there are 3 nested sublisteners, meaning that with each tap on the robot it will jump down to the next listener, in the end starting over again.

```
My robot should stop.
When it is tapped it should move forward for 2 seconds and wait
  (but when it is tapped it should turn left 90 degrees and wait
    (but when it is tapped it should move forward for 2 seconds and wait
      (but when it is tapped it should turn right 90 degrees and start over)))
```

Listing 4. A robot using nested sublisteners to execute a series of actions step-by-step

4 DESIGN

4.1 Meta-model

We now present the meta-model, which is the foundation of the language. The root element is an instance of, indeed, the *Root* class, and its exists because the Ecore framework requires the meta-model to be a single partonomy. The rest of the meta-model will be dealt with in the following paragraphs. We use Figures 2, 3 and 5 to discuss the remaining meta-classes, and as an example, an instance diagram of the program in listing 2 is shown in Figure 4.

4.1.1 How to instruct the robot. A robot, represented by the *RoboProse* class, is instructed by means of atomic units, each one called *Action*. Concrete *Action*s can be seen in the diagram in Figure 3a: so far, only very simple *Action*s such as *Move*, *Turn* and *Stop* are implemented, with rather self-explanatory semantics. In Figure 4, examples of *Action*s are *Move* and *Turn*. *Actions* can have different traits, such as having a duration or being randomized: these aspects are modeled by the abstract classes *ContinuosAction* and *RandomAction*. In the former case, when the duration is not set the *Action* lasts indefinitely. The semantics of the latter is that if its *isRandom* attribute is set to true, then all other undefined attributues are set to a random valid value, instead of the default

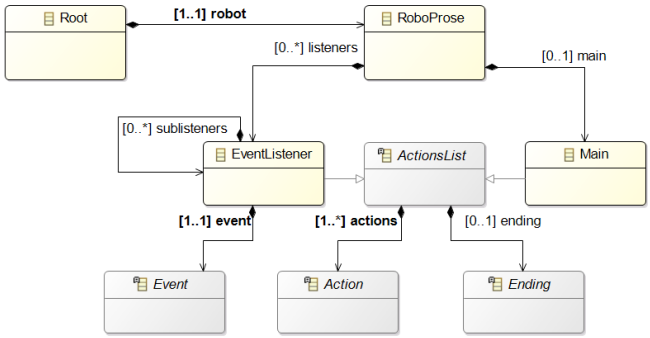
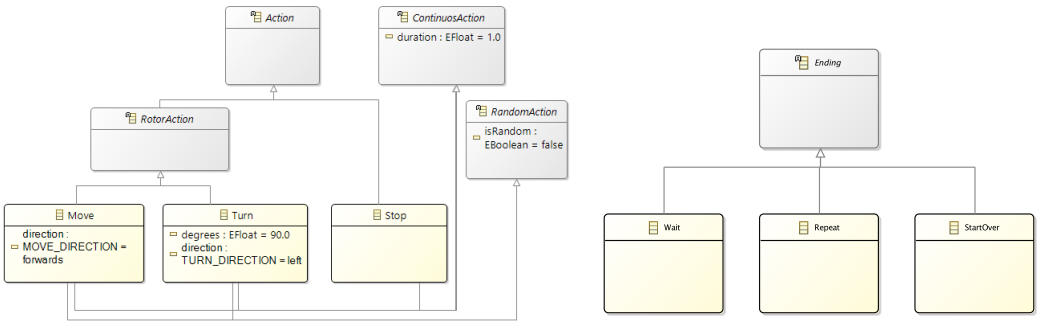


Fig. 2. The main meta-model diagram. *Action*, *Ending* and *Event* generalization trees are not shown here for the sake of clarity.



(a) Actions (b) Endings

Fig. 3. *Action* and *Ending* taxonomy diagram.

ones. For example, a *Turn* object with its *direction* set to ”left” and *isRandom* set to true, would be given a random duration, but if the direction were not set, it would also have been assigned it randomly.

4.1.2 Executing actions. Executing an action signifies that a command with a corresponding meaning is sent to the robot currently being controlled. *Action*s are executed when the containing *ActionsList* is scheduled (more information about this are found in the following section). *ActionList*s execute the *Action*s they contain in a serialized fashion. Each *ActionsList* can also have an optional *Ending*, describing the behavior of the robot after the list of *Action*s has been executed: it can run through the current *ActionList* again (*Repeat*), it can execute the *Main* (*StartOver*), or it can stay motionless until another listener is triggered (*Wait*). It is worth noting mentioning that using *StartOver* or *Repeat* as the ending in the *Main* has the same effect, as they both reference, absolutely and relatively respectively, the same *ActionsList*. The taxonomy of *Ending* is shown in Figure 3b, while in Figure 4 the only instance is *StartOver*.

Currently there is no way to resume to the *ActionList* of the parent listener (if there is one). It is only possible to either start the program from the beginning or to repeat the current block. This is a choice made to reduce complexity, but it would be possible to implement by adding an extra subclass of *Ending*.

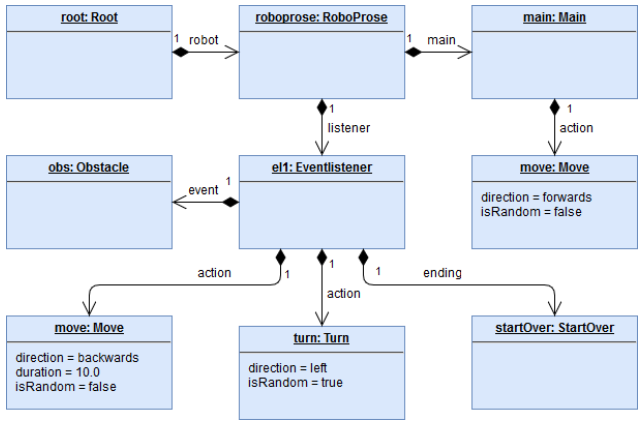


Fig. 4. Instance diagram for listing 2.

4.1.3 ActionLists scheduling. The way an *ActionList* is scheduled depends on its concrete class: *Main* is executed as soon as the Thymio is started, while *EventListener*s are triggered by *Event*s occurring. In such situations, the list of *Action*s currently executing is unscheduled, and the one belonging to the *EventListener* associated to the occurring *Event* is scheduled instead. Note that an *EventListener*'s list of actions can be interrupted by another *EventListener* if the latter's *Event* occurs while the former is scheduled.

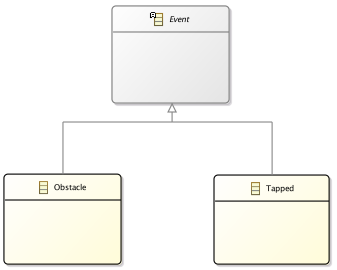


Fig. 5. *Event* taxonomy diagram.

Furthermore, *EventListener*s can be nested, with the semantics being that the inner *EventListener* are only scheduled if the outer listeners *Action*s list is currently executing. An example of *EventListener* can be seen in Figure 4, reacting to *Obstacle*. At present, only two concrete implementations of *Event* exist: *Obstacle*, fired when the proximity sensors detect anything near, and *Tap*, broadcast when a top button is pressed. Even though *Event* could currently be an enumeration type, it is modeled as a class in order to ease the implementation of event attributes, which are natively supported by Thymio and Aseba Scripting Language.

4.2 Concrete Syntax

This language is designed to write simple small programs for a small simple robot, and it is in this context that the syntax looks best. The entire grammar of the language, written in extended Backus-Naur form is shown in Figure 6: each of the endings and events (*END* and *EVT* in Fig. 6, respectively) are just different terminals corresponding to the classes from Figures 3b & 5. This simple solution is possible since the classes contain no attributes or references, and therefore no other information needs to be captured. The actions (*ACT* in Fig. 6) on the other hand, have several attributes that can be set and their grammatic rules are therefore more complex.

A remarkable feature about this language is that all of the words in all valid RoboProse-programs are registered keywords. There are no IDs, variable-names or named sub-routines in the language, and this is a precise design choice that allows us to keep the language simple and fairly understandable no matter the naming convention whims of the programmer. The only elements that are not registered keywords are the numbers that can be set as attributes to the actions. It could be a

$$S \rightarrow (\texttt{'My''robot''should'}\ ACT\_LIST\ \texttt{'.'})?\ (EVT\_LI\ \texttt{'.'})^* \quad (1)$$

$$EVT\_LI \rightarrow (\texttt{'When'|'when')'it'}\ EVT\ \texttt{'it''should'}\ ACT\_LIST \quad (2)$$
$$\hookrightarrow (\texttt{'('but'}\ EVT\_LI\ (\texttt{'.''And'}\ EVT\_LI)^*\texttt{')'})?$$

$$ACT\_LIST \rightarrow ACT\ (ACT\_DL\ ACT)^*(ACT\_DL\ END)? \quad (3)$$

$$AC \rightarrow (MOVE \mid TURN \mid STOP) \quad (4)$$

$$ACT\_DL \rightarrow ((\texttt{'and'}\ \texttt{'then'?}) \mid (\texttt{','}\ \texttt{'and'?}\ \texttt{'then'?}) \mid (\texttt{'.'}\ \texttt{'Then'})) \quad (5)$$
$$\hookrightarrow (\texttt{'it'}\ \texttt{'should'})?$$

$$END \rightarrow (\texttt{'repeat'|'start''over'|'wait'}) \quad (6)$$

$$EVT \rightarrow (\texttt{'meets''an''obstacle'|'is''tapped'}) \quad (7)$$

$$MOVE \rightarrow \texttt{'move'}((\texttt{'forward'|'forwards'}) \mid (\texttt{'backward'|'backwards'|'back'}))? \quad (8)$$
$$\hookrightarrow (DURATION \mid RANDOM)?$$

$$TURN \rightarrow \texttt{'turn'}(\texttt{'left'|'right'})?\ (DURATION \mid (FLOAT\texttt{'degrees'}) \mid \quad (9)$$
$$\hookrightarrow RANDOM)?$$

$$STOP \rightarrow \texttt{'stop'}DURATION? \quad (10)$$

$$DURATION \rightarrow \texttt{'for'}\ FLOAT\ (\texttt{'second'|'seconds'}) \quad (11)$$

$$RANDOM \rightarrow \texttt{'random'|'randomly'} \quad (12)$$

$$FLOAT \rightarrow \texttt{'-'}?\ INT\ (\texttt{'.'}\ INT\ ((\texttt{'E'|'e')'-'}?\ INT)?)? \quad (13)$$

$$INT \rightarrow (\texttt{'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'})^+ \quad (14)$$

Fig. 6. The concrete syntax of RoboProse in extended Backus-Naur form, where S denotes the starting rule.

possibility to extend the language with custom events or named subroutines to be used as actions, as a way to introduce the concepts of referencing and function calls to the aspiring programmers.

In order to recreate the feel or flow of normal prose, several of the terminals have variations, such as the action delimiter (*ACT_DL* in Fig. 6), that can be written in a total of 14 different ways. This allows the user to writes programs like "`My robot should move forward and then turn left. Then it should move backwards, turn right and stop.`" instead of the unvaried (but still valid) version "`My robot should move forward. Then it should turn left. Then it should move backwards. Then it should turn right. Then it should stop.`" The language could be extended with arbitrarily many synonymous alternatives to each of the terminals, but we have chosen a small set to demonstrate the concept.

4.2.1 Nesting of listeners. One of the issues we had to tackle is how to make the nesting of sublisteners explicit in the code. If an *EventListener* has more than one *sublistener* and one of these has a sublistener itself, it becomes hard to distinguish whether a given sublistener is the sibling or the child of the previous listener. For this problem a simple "." at the end does not suffice - we need a both a starting and an ending delimiter for each list of sublisteners. Otherwise we get into a "Dangling Else"-situation[3] and our grammar becomes ambiguous. We could then configure the parser to deterministically pick one option and remove the ambiguity, but in our case that would mean that sublisteners could only be one level deep, or that sublisteners could not have siblings.

This is not a problem that normal prose gives many tools to solve, and we would like to avoid the use of special characters like {}, [] or <>. A version we considered was using a white-space-aware grammar like that of Python[4] or YAML[5] and use indentation to determine the nesting of expressions,

[3] A famous grammar problem, where the nesting of an "else" becomes ambiguous. For more information see http://dx.doi.org/10.1145/365813.365821, retrieved 10/12/2017

[4] https://www.python.org/, retrieved 07/12/2017

[5] http://yaml.org/, retrieved 07/12/2017

```
"There should be no EventListeners for the same event on the top level"
  ->
    inv[RoboProse] {
      self => uniqueEvent(self.getListeners.toList)
    },
"There should be no SubListeners for the same event on the same level"
  ->
    inv[EventListener] {
      self => uniqueEvent(self.getSublisteners.toList)
    }
```

Listing 5. Example of constraits

but normal prose does not look like that. It would look very unfamiliar to a reader used to normal prose, which is something we would like to avoid as much as possible.

Instead we decided to use normal parenthesis to determine nesting of sublisteners inside a top level Eventlistener. Parenthesis are normally used in prose to inject half a sentence (or just some words) into another sentence, and the extra words receive some intuitive nesting from this relationship. A ”but” is also introduced to explain to the user that this event allows the robot to stop the actions it was currently performing and execute another list instead. For simple use cases, like Listing 3, this solution looks fine, but once used in a more complex way, this of cause quickly stops resembling normal prose (as for example in Listing 4). This might be unavoidable though, since normal prose rarely has semantic nesting several levels deep.

4.3 Static Semantics

As good modeling practice suggest, constraints are implemented directly in the meta-model as much as feasible. Indeed, all the fundamental and intuitive structural restrictions are expressed with cardinality constraints (see Fig. 2): for example, in our *RoboProse* meta-model we require that every instance of *RoboProse* has at most one *Main*. Also, we force total generalization by making super classes abstract, as seen for instance in the taxonomies of *Action* and *Event* (fig. 3a and 3b respectively). To express all the other restrictions that can not be easily declared directly in the meta-model, we use constraints written in Scala[6].

For instance, we have not captured the EventListener - Event associations semantics precisely in our meta-model; to tackle this problem, we had to add the two constraints shown in Listing 5: the first constraint states that for each *RoboProse* instance there is no *EventListener* for the same event on the top level; the second constraint checks the same but at any other level. This is an example of constraint originating from the domain of the problem, since not declaring anything specific about this would lead to ambiguities. Another example of constraint springing from the domain area is that non-terminating actions can only be at the end of an *ActionsList*, and in such case the *Ending* must not be set: both rules are necessary to have sensible semantics. Some constraints come instead from the implementation area, such as the inability for a *Turn* action to have both a duration and degrees: in fact, the speed of the robot is bounded, which implies that it is not physically possible to implement all of such combinations.

[6] http://www.scala-lang.org/, retrieved 07/12/2017

```
const mapActionList = (actionList, defaults) => {
    const actions = actionList.get('actions')
        .map(model2ThymioAction.bind(null, defaults));
    const ending = Option(actionList.get('ending'))
        .map(ending => ending.eClass.get('name').toLowerCase())
        .unwrapOr(null);
    return {actions, ending};
};
```

Listing 6. Model-to-model transformation of ActionsList

4.4 Implementation of the language

The backing implementation of the language is an interpreter written in JavaScript running on the Node.js[7] platform, supported by a comparatively small amount of Aseba Script code generation.

The reasons behind this dual nature of the back-end lies in the fact that the interpreter leverages on the DBus interface the Aseba ecosystem provides to communicate with Thymio, namely *asebamedulla*[8], which implies that it needs to be run on a DBus-capable device connected to the robot that the RoboProse program is meant for. However, local Thymio events, such as button press and sensor proximity, are not available from outside the robot itself, making impossible to trigger the execution of the *EventListeners* without broadcasting them on the DBus network: this is where the code generation becomes necessary, in the form of a small Aseba script loaded on the robot, detecting and broadcasting events.

Our choice of Node.js as a platform springs from the inherent parallelism of DBus signals handling: our previous experiments with DBus and python showed us that reacting to signals is a source of concurrency problems, in contrast with polling data from the robot periodically to detect changes. Therefore, we decided to leverage on Node.js serialized handling of asynchronous events to side-step the problem.

The whole back-end can be divided in two phases: an model-to-model transformation and the actual execution of the actions on the robot. The former happens first, serving the purpose of transitioning from a human-friendlier form, closer to the meta model, to a more machine-oriented representation, namely the *Thymio* class used in the second phase, which makes it easier to communicate with and instruct the robot.

The foundations of the back-end is the *Thymio* class, that has methods such as *moveForward()* and *turnLeft()*, that are called when the corresponding *Action*s need to be run. In the model-to-model transformation, lists of *Action*s are mapped to Javascript Arrays of Objects, in which every item describes the method which should be called, and optionally, some arguments. When it is created, the instance can be executed by means of the *run()* method, that traverses the aforementioned lists when appropriate, executing every action in a serialized fashion.

4.5 Limitations of the language

We have already mentioned some of the limitations of the language in the prior sections, for example the fact that one cannot define named subroutines or events and reference them from other places in the code. Nevertheless, another quite important limitation that we have not mentioned yet, is the complete lack of *state*.

[7] https://nodejs.org, retrieved 07/12/2017

[8] Stéphane Magnenat and Francesco Mondada paper on DBus integration for Aseba, https://infoscience.epfl.ch/record/140494/files/aseba-esb-ro09.pdf?version=2, retrived 10/12/2017

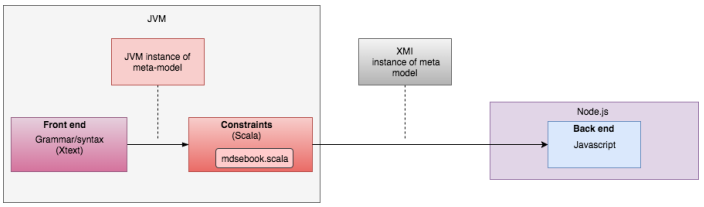


Fig. 7. Architecture of the tool chain

For instance, consider the simple scenario of a robot that has a repeating unconditioned stream of actions and reacts in a certain way the first time it meets an obstacle, returns to execute the main again and then reacts in a different way when it meets another object. This behavior is impossible to express in our language, as there is no manner of keeping track of the amount of objects that have been encountered. The language has no variables, neither mutable nor immutable, so there is nowhere to store them. This is chosen to keep the language simple and to focus on an event-driven way of programming.

But as the attentive reader might recall, we have seen something resembling state in listing 4, and it is possible to simulate certain kinds of state-like behavior by nesting listeners, although it quickly becomes extremely verbose and hard to maintain.

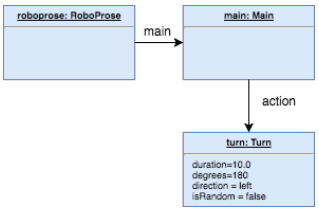
5 ARCHITECTURE OF THE ENTIRE TOOL CHAIN

The whole system is articulated in the three canonical phases: front-end, constraint validation and back-end. The front-end is implemented in Xtext[9], thus it runs on the Java Virtual Machine; so does the constraint validation step, written in Scala with the aid of mdsebook.scala[10] library; finally, the back-end is powered by JavaScript and the Node.js runtime.

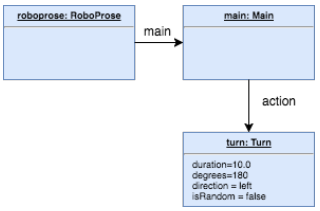
As a result of using two different runtime environments, the steps must exchanging information in a platform-independent format: in our case it is xmi, and the means of communication is the filesystem. In particular, the front-end and the constraint validation are run sequentially in the same program at compile time, sharing information in JVM format: if the process succeeds an *xmi* file is then produced and saved in the disk. After this, the back-end will be run with such file as a parameter, but before actually instructing the robots, the constraints are checked again, by spawning a child process, in order to avoid failures in case the *xmi* file has been tampered with.

Third-party software wise, a number of additional external packages have been used in the back-end, where the most part of the programming lies. The very first library to consider is the Node.js DBus binding, dbus-native[11]; then, the package used to parse xmi and ecore files is ecore.js[12]; thirdly, the collection of functional tools and heplers lodash[13] is utilized, mainly in the model-to-model transformation. Finally, the asynchronous coding patterns typical of the Node.js platform are handled by a massive usage of ReactiveX[14] library, in its JavaScript flavor[15]. This allows to overcome rather delicate issues such as event throttling and action execution interruption by using methods of the *Observable* class; as a side effect, it also increases the amount of nice functional-looking code.

[9] https://www.eclipse.org/Xtext/, retrieved 07/12/2017
[10] https://bitbucket.org/modelsteam/mdsebook/src/f2aa8c860391a91338a253bafb2c6d1dc454a9ad/mdsebook.scala/?at=master, retrieved 07/12/2017
[11] https://github.com/sidorares/dbus-native, retrieved 07/12/2017
[12] https://github.com/emfjson/ecore.js retrieved 10/12/2017
[13] https://lodash.com/ retrieved 10/12/2017
[14] http://reactivex.io/, retrieved 07/12/2017
[15] http://reactivex.io/rxjs/, retrieved 07/12/2017



(a) An unexpected RoboProse negative instance with *Turn* action *degress* and *duration* set.



(b) A positive test case for testing *indefinite* constraint.

Fig. 8. Example of positive and negative test case instances.

6 QUALITITY ASSURANCE

6.1 Testing Constraints

To ensure that our constraints are valid we have created both positive and negative instances and tested on each constraint, with some examples shown in Figure 8. Our objective was to ensure that the constraints actually catches the error, thus we were more focused on the negative corner cases. For instance we tested if the constraint catches the error for an unexpected RoboProse instance 8 (a) where both *Degree*s and *Duration* where set on *Turn Action*.

6.2 Back-end tests

The leading principle for back-end tests is code coverage. It has been applied to both of the main areas of the interpreter, that is the model-to-model transformation and the actual execution of the actions: however, the implementation of such principle in these two cases is inherently different. In fact, the model-to-model transformation is yet another data-manipulating program, and as such is tested with the usual tests pattern; on the other hand, the actions execution phase produces results by setting values on external hardware, meaning that the main testing procedure is to check, after a suitable amount of time, whether the values on the robot are the expected ones.

7 EXTENSIONS

We designed the language to be simple, so that it would be simple to learn and simple to use. This of course means that there is a lot of room for extensions, but is also means that it might not be a good idea to implement of all of them, as the additional complexity could defeat the purpose of the language. An option could be to add referencing and state to an ”advanced” version of the language, as is done in VPL (Fig. 1a), but it is not something we will elaborate on in this paper.

There are still extensions that are completely within the purpose of the language though, and these are mainly about expanding the types of Actions and Events.

It would be useful to have events corresponding to all of the Thymio’s sensors, which include sound, color and proximity. For the obstacle-event adding more attributes could help differentiate between obstacles at different angles and also at which proximity the robot should react to them.

Ideas for more actions could come from Thymio’s different modes of output: It could play a sound or show colors on the LED’s. But new methods could also be made by abstracting over the current ones, like an action called ”Wiggle” or ”Shake” where the robot quickly turns from side to side. You could also implement even more abstract Actions, for example you could make an ”Avoid” action that uses the proximity sensors and odometry to try to avoid an object and return to the same path.