Agile Development using Formal Methods

Erik Steenman
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.e.steenman@student.utwente.nl

ABSTRACT

Formal methods are a great way to provide more certainty about the correct behaviour of a software system. Using them, however, is generally associated with large costs. This research explores a more lightweight approach of using formal methods by creating a specification in a high-level language, which is transformed into both a formal model in mCRL2 and an implementation in Java. It has been tested that the mCRL2 toolchain works with this workflow, and that the mCRL2 and Java code generated are equivalent. This paper describes the structure of this intermediate model, along with its translation into mCRL2 and Java. It also shows that these translations are correct, and that the approach is effective in terms of reducing the effort of changing requirements later on. While the result of this research is not yet a complete language that could be used in every domain, it shows that the concept is viable and provides a solid starting point for such a language.

1. INTRODUCTION

For a long time, people have advocated the need of a way to guarantee the correctness of computer software to ensure correct behaviour of these systems [2].

Guaranteeing the correctness of large programs is nearly impossible due to the high complexity. Previous research has pointed out that finding software errors in the early stages of development significantly reduces the cost of fixing these errors [1].

Formal methods (FM) are often proposed as a solutions to these issues. However, several costs are associated with using these. For one, the programmer will have to be trained in the use of these methods, which is often not a skill programmers possess beforehand. Secondly, they generally focus on the requirements discovery phase, during which they are used extensively. Mistakes in this phase either mean that either the correctness can no longer be guaranteed, or it means re-doing a large number of steps in the development process, costing significant amounts of effort and time [1, pp. 6, 8]. This means that the requirements have to be frozen for a long period of time in order to guarantee as much certainty of the correctness of the formal specification as possible [1, p. 8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
24th Twente Student Conference on IT July 1st, 2016, Enschede, The Netherlands.
Copyright 2016, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

While this cost is worth it in a number of use cases, especially the cost of changing the requirements seems to be a major obstacle for widespread adoption of FM in software development [1]. This cost is hardly acceptable in certain types of software, where requirements change quickly based on iterations of customer feedback.

The goal of this research is to find a way of applying the benefits of FM in an agile development process. The main goal is to allow regular changes to the requirements in every step of the development process, without having to spend a large amount of time adapting the formal specification and the resulting codebase. This has been done by writing a language in which a system model can be defined, called the Intermediate Model (IM). A parser has been written that can transform this language into both a formal specification in mCRL2 [9] and an implementation in Java. The choice for mCRL2 has been made because a number of useful tools exist for it, which have been used in this research.

The novelty of this approach is primarily in using an intermediate model that creates both a formal specification and an implementation, rather than attempting to translate a formal specification into an implementation. This paper shows that this approach can result in a large reduction of workload spent on both the formal specification and the implementation.

1.1 Research Questions

An appropriate format for the intermediate model (IM) will have to be found. The desired format is dependent on how it can be transformed into a formal specification in mCRL2 and an implementation in Java. What this format should look like is the primary research question. To answer this question, there are a number of subquestions:

1. Is the behaviour of the generated mCRL2 code equivalent to the intended result based on the IM?
2. Is the behaviour of the generated Java code equivalent to the intended result based on the IM?
3. Does using this approach indeed lower the effort required to use FM in an agile development process?
4. Does using this IM still provide the benefits of FM?

1.2 Method of Research

To answer the research questions, a number of experiments will be performed. This will be done using an example system which describes a traffic light system. This system is described in detail in section 3.3.

1.2.1 Effort to change requirements

To answer the third research question, an analysis will need to be made of the effort needed to make changes to the definition if the requirements change. To do this, a requirement change will be made in the intermediate model. The total number of changes in the IM versus the changes in the generated mCRL2 and Java files will give an indication of how much easier making changes to the requirements is using this method.

1.2.2 Validating the generated mCRL2 with μ-calculus

To answer the first research question, the mCRL2 toolchain will be used. By using this toolchain to verify the correctness of the generated mCRL2 model with formal requirements written in μ-calculus [10], it can be shown that this model can be used to create a correct model. This will also partially answer the fourth research question, by showing that at least a number of the benefits of the mCRL2 toolchain are still available.

1.2.3 Validation using JTorX

To answer the second research question, the program JTorX [15] will be used. Because the mCRL2 model is proven correct separately, it can be assumed, by extension, that the Java program is also equivalent to the intended behaviour.

JTorX is a program that can be used to test the conformance of the behaviour Java program against a formal model as generated from a mCRL2 model. It can be used with a Java program that takes actions from *stdin* to verify against the formal model. By giving *jtorx* as a parameter to the generated Java program, all non-error output is disabled, which makes it compatible with the JTorX workflow.

Using JTorX to check the equality of the generated models is another part of the answer to the fourth research question, as this is a tool more often used in development using FM.

1.3 Structure of this paper

Section 2.1 describes some background knowledge the reader of this paper might find useful. After that, section 2.2 will describe work that has been done in former research to attempt to solve this problem.

In section 3, the designed IM is described. This is done by first explaining the decisions regarding the design of the IM in section 3.1, and then describing the exact structure in section 3.2. Section 3.3 describes how this model is transformed into mCRL2 and Java.

Section 4 will describe the results of the experiments. Section 5 will discuss the implications of these results to answer the research questions. Finally, section 6 will describe the general conclusions that can be taken from the results of this research.

2. LITERATURE

2.1 Formal Methods

Formal Methods are techniques used to model systems as mathematical entities [3]. This is done using Formal Specification Languages, which are mathematically inspired languages. These languages enable the developer to define the behavior of a system in a way that allows for verification using mathematical proofs. The main rationale behind FM is that time spent on specification and design will be repaid by a higher quality product [6].

Using the Formal Specification Language (FSL), the developer writes a Formal Specification (FS). In the case of mCRL2[9], which will be used during this research, this FS consists of various processes which are composed to a system. Each of these processes will usually simulate the states and actions of one part of the final system, and consists of data (state) and a state machine definition.

One advantage of FM is that the FS can be transformed into various other formats, which can be simulated and visualized as a state machine, or be verified. While these verifications and proofs can be performed manually, Formal Requirements (FR) can be written in e.g. μ-calculus, after which verification of the specification can be done quickly and automatically by formal tools.[9]

2.2 Related Work

The possibility of applying FM in a less extensive way has been recognized before [3, The Lightweight Approach]. One idea that has been proposed in the past is to generate an implementation from a formal specification [11, 8, 7, 16, 4]. This lowers the cost of using FM by largely removing the effort of making the accompanying implementation. However, this generally results in very domain-specific languages, and is thus not useful in the general case.

Another proposed solution is Formal Specification-Driven Development (FSDD). The first explicit mention of the term FSDD in academical literature seems to be by Rutledge et al[13], where he advocates for the application of FM in the context of Test-Driven Development (TDD). They propose a development method that combines FM with TDD. TDD traditionally defines test cases, from which unit tests are developed. The developer then writes source code until the unit tests pass, thereby fulfilling the requirements of the software [13, Fig. 1].

FSDD, in contrast, begins by writing a formal specification based on the formal requirements. This formal specification is then used to generate both unit tests and stubs for the software. By writing code to fill the stubs in the program, the developer makes the unit tests pass. Because of the generated stubs, the program can still be tested and verified based on the formal specification, as the structure is known by the compiler. This method has been tested quantitatively by Fofung[5]. However, this approach still requires each involved developer to be well-versed with FM.

3. THE LANGUAGE

3.1 Designing the Intermediate Model

Generating implementation code from a formal specification is not a new approach [11, 8, 7, 16, 4]. However, these papers use a complete formal specification and generate an implementation from this. This means that completely finished formal model is still needed to use this approach. The novelty in the approach of this research is in being able to make regular changes in the formal definition by defining an intermediate model which makes it easier to define one part of the system at a time.

One of the major issues with defining a complete system in a formal language is often that a lot of concepts in conventional programming languages, such as I/O, are not easily defined in a formal model. The reason for this is that a formal language is in essence a state model, while

a lot of those systems are more dependent on behaviour than state. This similarity to a state model does give us an interesting starting point for the definition of the IM.

A lot of systems have a number of components that can be modelled as a simple state system. Think of the traffic lights on an intersection, the carts in a roller coaster or a vending machine. In all of these cases, there are a number of elements that each have a state, which may change when something happens. These elements are called *actors* (e.g. a traffic light, a roller coaster cart), which each have a *state* (e.g. current color, whether or not it is moving), and *actions* which they can perform to change their state (e.g. change to red, start moving forward).

By modelling each actor as a separate state machine, each of these actions could be performed at any time by any of the actors. However, in reality, it is desirable to limit when they can be performed. It is undesirable to have all traffic lights to be green at the same time, or for two roller coaster carts to collide due to moving to the same place at the same time. Which actions should and should not be allowed at a given moment depends on the state of different actors. For example, if traffic light A is not red, traffic light B can not turn green. One actor preventing an action by another actor is called a *guard*.

By focussing on writing a language to model elements like this, the part of software where strict requirements are most relevant can at least be modelled. With just these elements, complete models can be defined, which is needed to correctly model a wide variety of systems. However, 2 more elements will be added to the vocabulary of the IM to simplify its usage.

The first of these is to add new types, so that e.g. the colours red, yellow and green for the traffic lights can be defined. These are called *sorts* in the IM, based on the term mCRL2 uses for this.

The second are *instances* of a certain actor. In some cases, there will be several actors with the same behaviour, but with separate state. Examples are the rows of a vending machine or the carts of a roller coaster. By being able to define multiple instances of the same actor, this can easily be implemented without duplicating code.

Thus, the final IM has *sorts* and *actors*. Each actor can have multiple *instances* which all have *states*, *actions*, and *guards* towards other actors. The exact structure of the language, and how it is transformed into mCRL2 and Java will be discussed in the following sections.

To parse the IM, a parser generator in Python called PyParsing [12] is used. The grammar that it uses is described in section 3.2. The resulting parse tree is used for a basic analysis of the model itself, and it assist in generating the translations.

3.2 Intermediate Model structure

The IM has a number of repeating constructs. These are mainly *identifiers*, *blocks*, *conditions* and *actions*. Comments can also be added to a model description by using a #.

Identifiers are generally used to refer to a certain object, such as an actor, action or struct item. They have to be unique, and can consist of alphanumeric characters and underscores.

Blocks are used to show which lines belong with which grammatical construction, actor, action, or condition. A block starts at an opening brace and ends with a matching closing brace. A block is generally prefixed with a fixed token or an identifier.

Conditions and actions will be described in detail later in this paper.

An IM is constructed from 2 main components: *sorts* and *actors*. Both are declared in their respective top-level block. While sorts are completely optional, there has to be at least one actor, so the actors block is required.

An abstract syntax tree showing the full structure of the grammar can be found in appendix A.

3.2.1 Sort

A sort is a custom type that can be defined. One example of a sort is the position of a switch (i.e. up or down). In this iteration of the grammar, only one type of sorts is supported: structs, or enumerated types. These consist of a list of possible items.

The sorts block, as seen in snippet 1 is prefixed with the *sorts* token. In the block, any number of sorts can be defined with a *struct()* operator, prefixed with the identifier of that sort. Within the struct's parentheses, a list of comma-separated identifiers identify which values that type can have.

Snippet 1. IM Sorts

```
1 sorts: {
2     Segment: struct(station, lift, main
        ↪ , braking, repair)
3     SwitchPos: struct(up, down)
4 }
```

3.2.2 Actor

Within the *actors* block, as seen in snippet 2, multiple actors have been defined. Each actor is denoted by an *identifier*, followed by a block that contains its properties.

Snippet 2. IM Actors

```
1 actors: {
2   cart: {
3     ...
4   }
5   switch: {
6     ...
7   }
8 }
```

Each actor has 4 types of properties: *instances*, *states*, *actions* and *guards*. *States* are optional, and if an actor has no states, then its *instances* are optional as well. Finally, both the *actions* and *guards* are optional, but an actor with neither of those will not do anything useful.

Snippet 3. IM actor states

```
1 states: {
2   position: Segment
3   locked: Bool
4 }
```

Actors will often have one or more states associated with them. These are defined in the *states* block, as shown in snippet 3. Each state is defined by an identifier, followed

by a *type*. The type can either be the built-in *Number* or *Boolean*, or a custom type as defined in the sorts block.

A basic actor will have one *instance*. In this case, it suffices to simply have a list of *assignment*s for each of the actor's states, as shown in snippet 4, and that instance can be referred to with the actor's identifier. However, if there are multiple actors with exactly the same behaviour, the *instances* block can be used to do so, as shown in snippet 5. If more instances are created this way, either all of an actor's instances or a specific instance in guards can be referred to by referring to the actor's identifier or the instance's identifier respectively. This can be seen in snippet 8 on line 4 and in snippet 10 on line 4 respectively. Inside an instance definition block, the initial states for that instance are defined.

Snippet 4. IM single instance

```
1  instances: {
2    position = main
3    locked = true
4  }
```

Snippet 5. IM multiple instances

```
1  instances: {
2    cart1: {
3      position = braking
4      locked = false
5    }
6    cart2: {
7      position = repair
8      locked = false
9    }
10 }
```

3.2.3 Actions and Guards

Each *actor* can have any number of *actions* and *guards*. Since the syntax for both is very similar, these will be treated together.

First, it is important to know the difference between *actions* and *functions*.

An *action* is any action an actor can perform, such as moving or changing colour. An action has an identifier and a list of *parameter type*s it can take. Finally, an action has one or more *function block*s, which contain *if/elseif/else* statements and *functions*.

A *function* defines the effect of the action on the states of the actor. It has a list of possible values for the *parameters* of the action, along with a list of state changes that it causes.

An action is defined by an identifier, which should describe what the action does, and has one or more *function blocks*. Each function block has a function call, possibly being surrounded by if/elseif/else statements.

A *function* call, as seen in snippet 6, describes what parameters the action can take, and what the resulting change in the actor's *state* is.

The *parameters* can be seen as conditions, as the function can only be executed with those parameters. A parameter can either be a *value* of the type of the parameter, or it can be an *Any* or *Not* operator.

Snippet 6. A function call in the IM

```
1  -> <parameters> {
2    <assignments>
3  }
```

By giving *Any(Struct)* as a parameter to a function, the function can be called with any of that struct's values. For example, *Any(SwitchPos)* allows the function to be called with either *up* or *down*, as defined in snippet 1.

A *Not* operator has similar behaviour and functionality. However, it also takes a list of either struct values or a state of the current or guarded actor to be excluded, in the case of actions and guards respectively. For example, by giving *Not(Segment: main, braking, repair)* as a parameter, the function can be called with either *station* or *lift*, since these are *Segment*'s values as defined in snippet 1. If the actor has a state called *position* of type *Segment*, that identifier can also be used in the list of excluded values. This will exclude *position*'s value at the time of execution from the function call as well.

Any function call can be surrounded by *if/elseif/else* blocks. These behave in the same way they do in other languages. Only if the *conditions* of the surrounding block are met, can the function call (with the given parameters) be executed.

All of the elements described above can be seen in snippet 7. For actions, these elements are enough to define what an actor can and can not do based on its own states.

Snippet 7. An action surrounded by an 'if'-statement in the IM

```
1  actions: {
2    # define an action with 2 parameters
3    # of type Segment
4    forward(Segment, Segment): {
5          # Allow this block if the actor
             ↪ 's current
6          # position is 'lift'
7          if (position == lift) {
8            # call the function
9            -> lift, main {
10                 # change current
                     ↪ position
11                 position = main
12           }
13         }
14   }
15 }
```

Basically the same syntax, however, can also be used to describe what another actor can and can not do. In the *guards* block, a block can be added for each actor that should be influenced. Inside that block, one or more functions of the guarded actor can be influenced.

The syntax for a guard can be seen in snippet 8. While either a complete actor or single instances of that actor can be guarded, overlapping guards are not allowed. This means that, if a guard is defined for an actor, defining guards on the same function for a single instance of that actor will not work.

Snippet 8. A guard with an 'else-if'-statement in the IM

```
1  # In the switch actor
2  guards: {
3    # Influence any cart's actions
4    cart: {
5      # Influence the 'forward' action of
         ↪ any cart
6      forward: {
7        if (switch_position == down) {
8          # Allow cart to move from
             ↪ braking to station
9          # The state of the switch does
             ↪ not change
10         -> braking, station { }
11       } elseif (switch_position == up)
           ↪ {
12         -> repair, station { }
13       }
14       # Regardless of the switch's own
           ↪ state, always
15       # allow the cart to move anywhere
           ↪ else.
16       -> Any(Segment), Not(Segment:
           ↪ station) { }
17     }
18   }
19 }
```

3.3 Translating the Intermediate Model

To demonstrate how various components of the IM are transformed into mCRL2 and Java, an example system describing traffic lights on a four-way intersection will be used. The full full source can be found on the repository for this project [14].

This is a system that can very easily be modelled by the IM. Each direction is labelled by a wind direction (north, east, south and west), and has 3 traffic lights (left turn, straight on and right turn).

Each traffic light will block specific other lights from turning green if it is not red itself. For example, if the northern traffic light for straight on is green or yellow, the eastern and western traffic lights for straight on should not be allowed to turn green. This is shown in figure 1.

The model has three actors, one for each type of light. Each actor has an instance for each direction. Each instance has a state variable for its direction, which will never be changed, and a state variable for its current color. This is enough to model the complete interaction between the traffic lights.

Note that this implementation could also have been made with either an actor for each individual light, or with a single actor that also has a state for its light type (left, straight, right). However, the current implementation provides a middle ground between readability (guards for each traffic light type are grouped) and code duplication, and it demonstrates more features of the IM.

The following sections will describe how this model is translated. For a couple of IM features that are not used by this traffic lights model, snippets from the roller coaster example in the repository will be used.

3.3.1 Translation to mCRL2

Sorts
The only custom sort currently supported is a struct. This is a structure that is directly taken from mCRL2 syntax, and therefore this is by far the most simple translation. An example in IM can be seen on line 1 of snippet 9.

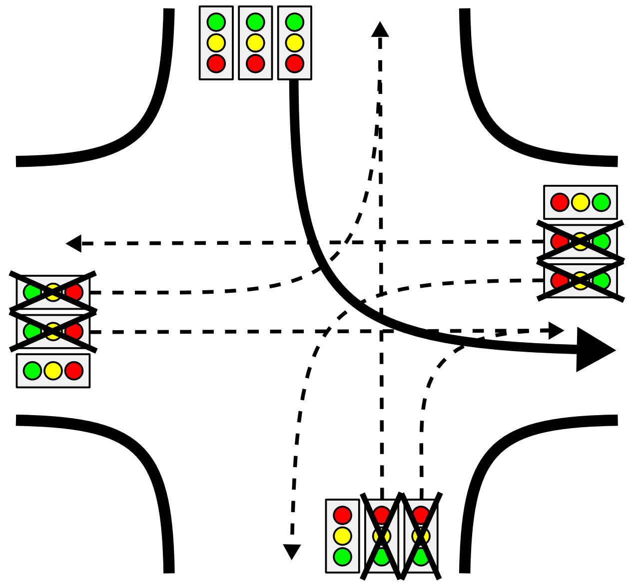

Figure 1. lights blocked by left turn being green

Snippet 9. Actor definitions in the example IM

```
1  sorts: {
2    WindDir: struct(north, east, south,
       ↪ west)
3      Colour: struct(red, yellow, green)
4  }
5
6  actors: {
7      straight_light: {
8      states: {
9        winddir: WindDir
10        color: Color
11     }
12
13     instances: {
14       straight_north: { ... }
15       straight_east: { ... }
16       ...
17     }
18     ...
19   }
20   ...
21 }
```

For each struct, a *sort* type will be defined in mCRL2, along with each possible value. This can be seen on line 6 of snippet 11.

Actors, instances and states
For each instance, a process will be generated in the mCRL2 code. The only difference between instances is the process name. The main reason for this is that this makes it easy to directly refer to that instance's actions and process.

Actions and Guards
For each action and guard in the IM, a single line is emitted in the mCRL2 model, separated by '+' signs. Each line starts with any if/else-if/else statements, followed by any *sum* operators for special parameter operations (explained below). After this, there will be the specific action or guard, with a specific naming scheme as can be seen on lines 2 and 6 of snippet 12. Finally, the process with any changed parameters is emitted.

Snippet 10. Action definitions

```
actions: {
  green(): {
    if (color == red) {
      -> { color = green }
    }
  }
  ...
}

guards: {
  straight_north: {
    green: {
      if (winddir == east  winddir ==
          ↪ west) {
        if (color == red) { -> {} }
      } else { -> {} }
    }
  }
  ...
}
```

Snippet 11. Translation of snippet 9 to mCRL2

```
act
  left_east_red ;
  left_west_red ;
  ...

sort
  WindDir = struct north | east | south
      ↪  | west ;
sort
  Color = struct red | yellow | green ;

proc straight_north(winddir: WindDir,
    ↪ color: Color ) =
  ...
proc straight_east(winddir: WindDir,
    ↪ color: Color ) =
  ...
```

The actions and guards on these actions are synchronized in the *init* block of the mCRL2 model. This means that the individual actors can communicate internally through synchronized actions.

Special parameter operations

There are two special parameter operators, which can be used to define a function call with a number of possible values. These are *Any* and *Not*. A detailed explanation on these parameters can be found in section 3.2.3, and an example can be seen in snippet 13.

Since mCRL2 has an operator that can be used to allow an action with multiple parameters, the *sum* operator, these can be used. For the *Not* operator, a number of 'if'-statements can be added after this sum operator. The result can be seen in snippet 14.

Other mCRL2 structures

There are two more structures in mCRL2 that have to be generated. The first of these is the list of available actions. To do this, a list of all actions and guards is kept and append these to the start of the document, as can be seen on line 1 of snippet 11.

A subset of these, the *perform* actions, will also have to be emitted in the *init* block of the mCRL2 model. However, this has been omitted from these snippets.

Snippet 12. Translation of snippet 10 to mCRL2

```
proc straight_north( ... ) =
  ( color == red ) -> (
    straight_north_green .
        ↪ straight_north ( color =
        ↪ green )
  )
  + ...  +
  ( winddir == east || winddir == west
      ↪ ) -> (
    ( color == red ) -> (
      straight_north_allows
          ↪ _straight_north_green .
          ↪ straight_north ( )
    )
  )
    <> ( straight_north_allows
        ↪ _straight_north_green .
        ↪ straight_north ( )
  )
  +

init
  allow ( { ... } ,
  comm ( {
    gate_open | button_allows_gate_open
        ↪  -> perform_gate_open , ...
  } ,
  straight_north ( winddir = north ,
      ↪ color = red ) || straight_west
      ↪ ( winddir = west , color = red
      ↪ ) || ...
```

Snippet 13. A guard with Any and Not parameters

```
forward: {
  if (pos == station) {
    -> Any(Segment), Not(Segment:
        ↪ station) {}
  }
  ...
}
```

Snippet 14. Translation of snippet 13

```
( pos == station ) -> (
  sum segment_1 : Segment . sum
      ↪ segment_0 : Segment . (
      ↪ segment_0 != station ) ->
      ↪ cart1_allows_cart2_forward (
      ↪ segment_1 , segment_0 ) . cart1
      ↪  ( )
)
```

3.3.2 *Translation to Java*

For the translation to Java, a number of classes are generated. This can be seen in figure 2.

First off, classes are generated for all the actors. These are called *Models* in Java. Each of these classes have an instance variable for each of the actor's parameters, a constructor that takes the values for these instance variables, and a method for each of the actions and guards the actor has. A part of one of these classes is shown in figure 3.

Then, a *Sorts* class is generated. This class contains an enum for all of the custom structs that have been defined. All of the models and the controller import this class to

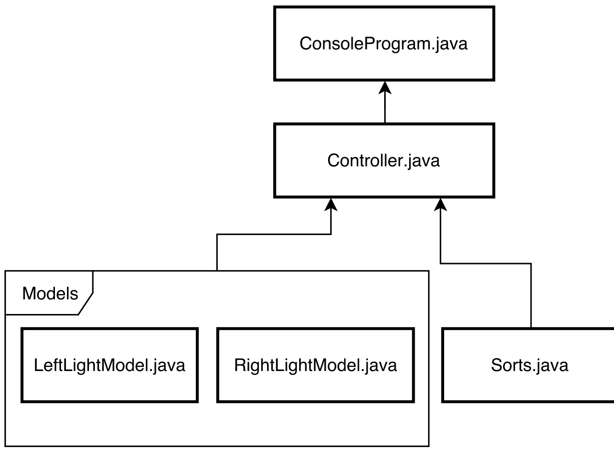

Figure 2. Class diagram of generated Java

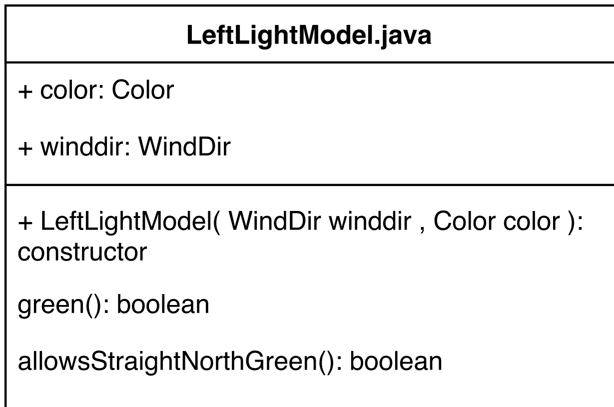

Figure 3. Partial LeftLightModel.java description

refer to the custom sorts.

A controller class is generated which holds an instance of each model for each instance the actors have in the IM. For each action, it has both an executing method (e.g. *performLightGreen()*) and a method that just returns whether or not a given action would be allowed (e.g. *isLightGreenAllowed()*). Both return a boolean value indicating whether or not it is/was allowed, but only the former will perform the relevant state changes if it was allowed.

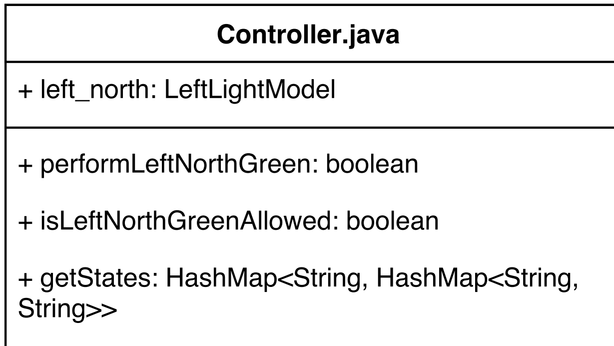

Figure 4. Partial Controller.java description

These classes can be used directly in a program, as they are encapsulated in their own sub-package. Currently, a sample class *ConsoleProgram.java* is also generated, which runs the model from a console or terminal directly. When doing so, it will show the current state of all actors, and accept inputs through the console. This program can also be used to test if an implementation in MCRL2 and Java are equivalent.

4. RESULTS

In this section, the results of the experiments described in section 1.2 are shown. A discussion of these results can be found in section 5.

4.1 Effort to change requirements

In a second iteration of the traffic light system, bicycle lights have been added. The bicycle lights are mutually exclusive with the normal traffic lights. Thus, if any bicycle light is not green, the other traffic lights should be red, and vice versa.

The old implementation can be found in the repository [14] in the *trafficlights_v1* folder, and the second iteration can be found in *trafficlights_v2*. The differences between each set of files between the first and second iterations can be found in table 1. This table shows both the number of lines (excluding empty and pure comment lines) and the number of lines that were changed by the requirements change.

Table 1. Differences in the intermediate model vs. generated files

Files	Initial lines	Changed lines
IM	453	108
mCRL2	1115	483
Java	1210	414

4.2 Validating the generated mCRL2 with μ-calculus

The properties that each traffic light should have are that it can only turn green when that can not cause a collission with cars from another light, that each traffic light will take on each color at some point, in a fixed order (green, yellow, red, green), and that the model contains no deadlocks.

A number of formulas have been written in μ-calculus, which describe these properties. One of these formulas can be found in snippet 15. Using these formulas and the mCRL2 toolchain, it has been verified that the generated model satisfies each of these properties.

Snippet 15. An example of μ-calculus used to verify the mCRL2 code

```
1 [true . perform_left_north_green . (!
      ↪ left_north_red) . (
2   perform_left_east_green ||
        ↪ perform_straight_east_green ||
3   perform_left_west_green ||
        ↪ perform_straight_west_green ||
4   perform_right_south_green ||
        ↪ perform_straight_south_green
5 )] false
```

4.3 Results of validation using JTorX

JTorX needs an explicit list of possible input actions to the program. Since, for this model, all permitted actions are input actions and there are no output actions, the list of actions of the mCRL2 model's *allow* block can be used.

Since JTorX does not attempt to visit each transition in the model, the best way to reach full coverage is by randomly

performing tests a large number of times with various random seeds. Both the initial version and the changed version of the traffic light system, as defined in section 4.1, are tested this way. In table 2, the number of states and transitions for both of these systems is shown. For both of these, ten runs with different seeds have been done, during each of which 20,000 random actions will be performed.

By manually changing one of the labels the generated Java program accepts into an invalid label, it has been verified that JTorX does indeed detect invalid actions and fail the run. The used settings for JTorX can be found in this project's code repository [14]. The seeds used for the ten runs are the numbers 1 through 10.

All of the runs passed.

Table 2. Sizes of mCRL2 models

Model	States	Transitions
Traffic lights v1	1737	8640
Traffic lights v2	1817	8964

5. DISCUSSION

The approach described in this paper has, so far, been used to model two systems: The traffic light system described in this paper, and a rollercoaster system. During this process, various issues with the initial version of the project have been discovered, and for a large part fixed. One weak point of the research presented in this paper, however, is that this method has only been applied to these two systems so far. Thus, it is not yet clear what features are missing from the language.

5.1 Effort to change requirements

Working with the IM has, based on the results of this test, very clear advantages versus working with the mCRL2 and Java code directly. For 100 changed lines in the IM, there are four times as many changes in both the mCRL2 and in the Java code. While not all of these changes will be substantial, and an eight times work reduction can thus not be assumed even in this particular case, this is still a good demonstration of the reduced amount of work needed to change requirements.

5.2 Validating the generated mCRL2 with μ-calculus

The written μ-calculus formulas give a good coverage of the expected behaviour of the model, both in terms of what it has to be able to do and what it should never do. The generated code passing the validation against these formulas gives reason to believe the behaviour is correct.

This means that at least a correct behavioural model can be generated from the IM. Based on this, it can be concluded that at least for a relatively restricted set of language features, this approach works.

However, creating the μ-calculus formulas is not yet a smooth integration with this project. The mCRL2 tools have to be called manually, and the generated files have to be examined to see the labels of the available actions. This approach could be improved a lot by automatically generating these formulas. This idea has also been mentioned in earlier research by Rutledge [13].

5.3 Validation using JTorX

Having a large number of random actions without any issues gives a very solid reason to believe that the Java program has at least the same features as the mCRL2 model, which has been proven to be correct separately in subsection 5.2.

This alone does not yet guarantee that the Java model will not have additional behaviour that is undesirable. However, the systems that have been used to test the approach are small enough to make it possible to manually verify the Java code, and there certainly does not seem to be additional behaviour. At the same time, together they use enough features of the language to make it a good test of the project. Also, since the logic to generate the mCRL2 and the Java code is very similar, it is very likely that no unexpected behaviour will have been added.

This test, however, does not prove the generated implementations are equal in general. Thus, this test would have to be run for every project separately.

6. CONCLUSION

Based on the results of the research method as found in section 4, the first two research questions can be considered answered. While the experiments done to validate the correctness of both the mCRL2 and the Java program can not guarantee complete correctness in the general case, the correctness of the example systems seems very likely.

The fourth research question has been answered in various parts of this research. During the development of both systems, the toolchain has been used extensively to simulate the formal model. This made it easy to see if the implementation and translation were correct.

Furthermore, it was relatively easy to write μ-calculus formulas to verify these generated models. This is also helped by the labels that have been generated by the actions mostly being constant. However, they are not easy to predict without looking at the generated code, which makes this integration still somewhat bothersome.

Because the generated model can be used directly with the toolchain, it can be concluded that at least a number of the benefits of working directly with a formal specification in mCRL2 are still available.

Testing the difference in lines of code changed clearly shows the potential benefits in reduced workload by using an intermediate model. Apart from the reduced amount of lines of code that have to be changed, a large part of the reduction in the effort comes from only having to implement the functionality once. This reduces the chance of errors being introduced during either development step.

It also invites the developer to keep using FM throughout the development process, rather than focussing only on the Java implementation due to time constraints. This means that even if the requirements change weekly and there are only a few days to implement and test these changes, a more formal development approach can be followed using this approach.

Thus, it can be assumed that the answer to the third research question is that this approach indeed lowers the effort needed to change requirements. This also implies that applying this approach in an agile development environment is much more viable than the traditional way of using FM.

Based on the answers to the subquestions, it can be concluded that the language described in this paper is a good basis for an intermediate model with which FM can be used better in an agile development environment. While it is not yet a complete language, it has been shown with two quite different systems that it is a functional and efficient

way to describe systems.

6.1 Future work

The approach itself, both working with actor and guards in a mCRL2 model and generating both the formal model and the implementation from an intermediate model, is still very new. Especially for the latter, no other research has been found discussing the approach. Thus, it is important to do more research in this regard, and especially to try to use these approach in real, larger software projects.

The IM language can also still use a lot of work to ease the development process, and to support more types of systems.

To reduce the amount of code duplication, extra custom parameter operations can be added, besides *Not* and *Any*. Examples of these would be custom functions that map one struct item to another, to construct ring or mesh structures between instances. Another idea is to introduce inheritance, so there is no need to duplicate code if multiple actors have the same methods.

To make the language easier to work with, parsing errors still have to be improved a lot. There are only standard PyParsing errors for the moment, which do not describe the exact issue and location very well.

Debugging an IM implementation is currently quite bothersome if larger amounts of code are introduced at once, since it is not possible to see why a certain action is not allowed. Being able to visualize either real-time, or based on a log, what actors are blocking a certain action, would make debugging a lot easier.

Finally, better integration with the mCRL2 toolchain would help a great deal. It is currently hard to create μ-calculus formulas or Java integrations with the model, as there is no simple way to see which actions are available. Even better would be a way to automatically generate the μ-calculus formulas.

References

[1] D. M. Berry. “Formal methods: the very idea: Some thoughts about why they work when they work”. In: *Science of Computer Programming* 42.1 (2002). Special Issue on Engineering Automation for Computer Based Systems, pp. 11–27. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/S0167-6423(01)00026-0. URL: http://www.sciencedirect.com/science/article/pii/S0167642301000260.

[2] J. Bowen and V. Stavridou. “Safety-critical systems, formal methods and standards”. In: *Software Engineering Journal* 8.4 (July 1993), pp. 189–209. ISSN: 0268-6961.

[3] M. Collins. *Formal Methods*. 1998. URL: https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/.

[4] H. Evrard and F. Lang. “Automatic Distributed Code Generation from Formal Models of Asynchronous Concurrent Processes”. In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Mar. 2015, pp. 459–466. DOI: 10.1109/PDP.2015.96.

[5] T. D. Fofung. “Quantitative Analysis of Formal Specification Driven Development”. MA thesis. Southern Polytechnic State University, 2015. URL: http://digitalcommons.kennesaw.edu/cgi/viewcontent.cgi?article=1687&context=etd.

[6] R. M. Hierons et al. “Using Formal Specifications to Support Testing”. In: *ACM Comput. Surv.* 41.2 (Feb. 2009), 9:1–9:76. ISSN: 0360-0300. DOI: 10.1145/1459352.1459354. URL: http://doi.acm.org/10.1145/1459352.1459354.

[7] X. Liu and R. Fan. “A Lightweight Framework for Code Generation from B Formal Specification”. In: *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*. Vol. 1. Mar. 2010, pp. 133–136. DOI: 10.1109/ETCS.2010.204.

[8] J. Rash M. G. Hinchey and C. A. Rouff. *Requirements to Design and to Code and Towards a Fully and Formal and Approach to Automatic and Code Generation*. Tech. rep. NASA, 2005. URL: http://ntrs.nasa.gov/search.jsp?R=20070014068.

[9] *mCRL2 documentation*. University of Eindhoven. 2015. URL: http://www.mcrl2.org/ (visited on 06/11/2016).

[10] *mu-Calculus*. University of Eindhoven. 2015. URL: http://www.mcrl2.org/dev/user_manual/language_reference/mucalc.html (visited on 06/11/2016).

[11] C. F. Ngolah and Y. Wang. “Exploring Java code generation based on formal specifications in RTPA”. In: *Electrical and Computer Engineering, 2004. Canadian Conference on*. Vol. 3. May 2004, 1533–1536 Vol.3. DOI: 10.1109/CCECE.2004.1349698.

[12] Pyparsing. *Pyparsing Wiki Home*. 2016. URL: http://pyparsing.wikispaces.com/ (visited on 06/20/2016).

[13] R. Rutledge et al. “Formal Specification-Driven Development”. In: *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). Association for Computing Machinery (ACM), 2014, p. 7. DOI: 10.1145/2184512.2184595. URL: http://dx.doi.org/10.1145/2184512.2184595.

[14] L. E. Steenman. *Repository Bachelorreferaat*. 2016. URL: https://github.com/lesteenman/bachref (visited on 06/11/2016).

[15] EWI Utwente. *JTorX - a tool for Model-Based Testing*. University of Twente. 2016. URL: https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/ (visited on 06/11/2016).

[16] P. Valderas and V. Pelechano. “A Survey of Requirements Specification in Model-Driven Development of Web Applications”. In: *ACM Trans. Web* 5.2 (May 2011), 10:1–10:51. ISSN: 1559-1131. DOI: 10.1145/1961659.1961664. URL: http://doi.acm.org/10.1145/1961659.1961664.

APPENDIX

A. ABSTRACT SYNTAX TREE OF IM