Contents

Abstract

Model Based Development (MBD) using Mathworks tools like Simulink, Stateflow, etc. is used to develop safety-critical software. The safety of many of these technologies is of the utmost importance, particularly in the automotive industry. On the other hand, automatic formal verification techniques for Simulink, particularly on the model level, are uncommon and frequently experience scaling constraints. Formal verification approaches are well-known for their ability to discover design flaws in safety-critical systems, hence reducing the lot of time and cost spent on the development process. When using formal methods, the process of formally verifying safety-critical software components becomes both more efficient and less prone to errors. In this research, we present a translation of models written in Matlab/Simulink into the verification language Dafny. Because of this translation, we are now able to perform formal verification of Matlab/Simulink models utilizing the Dafny program verifier as well as inductive invariant checking. In addition to this, verification targets for the most prevalent error classes. Our strategy enables us to give a formal verification method for Matlab/Simulink and the most prevalent error classes. This method scales better than many other methodologies that are already in use, and it does so in the majority of instances. In order to demonstrate that our methodology is applicable in the real world, we have applied it to a few different case studies.

1 Introduction

This report addresses the research question: How can we translate Simulink models to a formal model written in the verification language Dafny for the purpose of verification of these models? It involves below steps:

1. Developing and implementing rules that read Simulink models and construct a formal model in a chosen formalism i.e Dafny.
2. Validation via case studies and different examples on freely available Simulink models

Matlab/Simulink is a tool for modeling, simulating, and analyzing electrical, mechanical, and hydraulic systems. Simulink's library is extensive. It is an easy-to-use tool for analyzing and simulating any system into a graphical representation.

In spite of the fact that Matlab/Simulink is used to create safety-critical systems, it is well recognized that testing and simulation of the models is the only way to verify the correct functionality of systems at the model level. A more thorough formal verification of models is necessary since rectifying flaws in later stages of development is more expensive. Formal verification of Matlab/Simulink models can be done automatically; however, the approaches that are used, such as abstract interpretation and (bounded) model checking, have scalability difficulties that include the state space explosion problem.

As a result, strategies that can be implemented at the model level and that show promise of scaling well even for big and complicated models are needed. Deductive verification technique for Matlab/Simulink model verification may be a solution to this issue. We don't have to go through the entire model's state space when we use a deductive verification method. Model constraints are shown to be preserved in all conceivable executions through the use of automatic theorem provers such as Z3 which is used in Dafny. This method necessitates the specification of invariants that may be used to test for the required characteristics of the system.

In this, we present a strategy for the transformation of Matlab/Simulink models into formal methods (Dafny). This makes it possible to use the Dafny programming language for the verification of Simulink models. During this transformation, invariants are generated to enable the verification of models for some common error classes.

The translation from Simulink Model to Dafny can be expressed as: Given the Simulink model, implement rules to translate the blocks of simulink model to Dafny and verify it. This approach satisfies the preservation for the semantics of the Simulink Model and provides coverage of the Simulink blocks that are used most frequently. Also, it guarantees that the translation and verification is completed within a predetermined amount of time and on a conventional computer system.

The rest of this dissertation report is structured as follows: In Section 2, We have discussed the preliminaries and related work. In Section 3, We present the translation rules and algorithm to translate Simulink model to Dafny. We have illustrated an approach for the transformation with practical applicability using few case studies. In Section 4 We have presented the result and its analysis with respect to Related Work. In Section 5, Conclusion of the research and possible future work for the approach.

2 Related Work

2.1 Basic Terminologies

2.1.1 Simulink Model

Hybrid Systems are combination of connected and discrete. They are gradually being used in increasingly complex scenarios with dynamic and unpredictable environment. Modelling languages like as MATLAB Simulink have been proposed and frequently used in industry to deal with the complexity of Hybrid Systems. Simulink is a powerful tool for modelling, simulating, and analysing electrical, mechanical, and hydraulic systems such as Hybrid Systems. Simulink's library is extensive. It is an easy-to-use tool for analysing and simulating any system into a graphical representation. Below figure shows a simple Simulink model [**simulinkweb**].

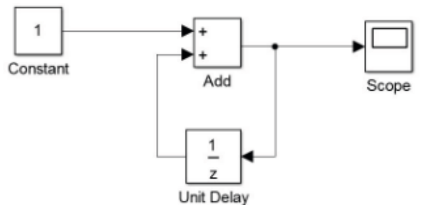


Figure 1: Simulink Model

Figure 1 shows a small example of a Matlab/Simulink system. It is consisted of four blocks: a Constant block, a Sum block, a UnitDelay block, and a scope to show the output. The signal lines link the blocks together. In this example, every time the simulation moves forward, the constant 1 is added to the output of the unit delay variable.

There are two major class in Simulink: blocks and lines. Blocks are used to generate, alter, combine and display the signal; however, lines are used to move a signal from one block to another. There are different classes of blocks in the Simulink Library, such as below:

- Sources: used to generate different signals.
- Sinks: used to display the output value.
- Math Operators: used to perform arithmetic operation such as add, multiple, gain, absolute value etc.
- Discrete: used for discrete-time system elements.
- Continuous: used for continuous-time system elements

Even though the data flow oriented notation of Matlab/Simulink is generally concurrent, Matlab/Simulink executes the blocks sequentially in a simulation. Matlab/Simulink enables hierarchical structuring of the model.

Matlab/Simulink makes it possible to structure the model in a hierarchical fashion by using sub-system blocks. Other blocks can only be contained within subsystem blocks. In order to model the signal flow into subsystems, mapping the inputs of the components that make up the subsystems to port blocks within the subsystem that are designated as "Inport" and "Outport".

2.1.2 Formal verification

Formal verification primarily focuses on determining whether hardware and software design operations are correct. Formal verification differs from validation in a way that validation aims to discover correctness by examining operation's behaviour whereas verification examines the correctness by mathematical proof (formal methods). The verification of these systems is accomplished by providing a formal proof based on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being known by construction. Examples of mathematical objects frequently used to represent systems include finite state machines, labelled transition systems, Petri nets, vector addition systems, timed automaton, hybrid automaton, process algebra, formal semantics of programming languages such as operational semantics, definition semantics, axiomatic semantics, and Hoare logic, and process algebra.

Formal method, models and evaluates a system using a three-step process. During formal specification, an engineer defines a system precisely using a modeling language, often by employing a formal, mathematical syntax and semantics that minimize imprecision and ambiguity. This is similar to defining system specifications, although not in plain English. The engineers then build a series of theorems regarding the behavior of a system based on the specification. The mathematical verification of these theorems ensures that the system's behavior is logically consistent and the desired one. As this enables designers and engineers to identify usability faults before the design is implemented in code, it eliminates costly errors from occurring later in the development process.

2.1.3 Formal methods and Dafny

Formal methods are procedures for modelling complicated systems as mathematical entities. By constructing a mathematically valid model of a complicated system, its properties can be verified more thoroughly than through empirical testing. Rigorous descriptions promise to improve system reliability, design speed, and comprehensibility, but at the expense of a steeper learning curve; the mathematical disciplines used to formally describe computing systems are beyond the scope of a conventional engineering degree.

Dafny is a programming language that includes specification constructs by default. The Dafny static program verifier can be used to validate the correctness of a program's functionality. Dafny ensures that your code conforms to the specifications you provide, while allowing you to write both code and specifications in the Dafny programming language. Due to the fact that verification is an inherent part of the development process, it reduces the possibility of expensive late-stage bugs that are generally missed by testing [10]. Microsoft created the Dafny research language. Its primary goal is data refinement. The language supports a wide range of mathematical features, including sequences, sets, and multi-sets, as well as functions, predicates, methods, and user-defined data types. The Abstraction Invariant in Dafny takes the form of a function that is applied as a pre and post condition to all methods and functions. With this function, one may ensure that the code provides an implementation that meets its specifications, even if the specification is defined in terms of one data structure and the code is implemented in terms of another. Dafny collaborates with Boogie [5], a static program verifier, and Z3, an SMT solver [8].

2.2 Simulink Design Verifier

Simulink Design Verifier detects latent design flaws in models using formal methods. It looks for model blocks that cause integer overflow, dead logic, array access violations, or division by zero. It has the ability to formally verify that the design fits the functional requirements. It creates a simulation test case for debugging for each design fault or requirement violation. Simulink Design

Verifier creates test cases for model coverage as well as custom objectives to supplement existing requirements-based test cases. Your model is driven by these test cases to meet condition, decision, modified condition/decision (MCDC), and custom coverage objectives. You can give unique test objectives in addition to coverage objectives to produce requirements-based test cases automatically. However, there are challenges with scalability, particularly for models with large state spaces, with these methods [7].

2.3 Formal Verification of Simulink Models

In [13]. the author demonstrated an algorithm which translates Simulink models into Why3 language. It consists of Identification of specification blocks, compute the abstract syntax tree for a model, then each signal from a block is identified, given a name, and translated to a real or Boolean function in Why3. Lastly, the verification objectives are added using structures such as the Hoare triple form from 'Require' blocks.

In [6], the Simulink model is translated into the input language of model checker(NuSMV).In this, the Simulink model is read from the textual representation by ignoring the graphics of the model such as color, size of the model block etc. Then, the input of each block is calculated by checking the output of it's preceding block. After that, building a NuSMV model in which each basic block is replaced by the NuSMV module(s) that are equal to it.

In [16], the transformation algorithm process a given Simulink model, applies the defined transformation rules, and incrementally generates a hybrid program. Handle all time-discrete stateful blocks in the correct order, i.e., we begin with blocks that have no inputs and then recursively handle all blocks where all input blocks have already been translated, assuming the system does not contain algebraic loops, each feedback loop in the original Simulink model containing at least one stateful block, and this algorithm always terminates.

In [4], the authors offer a method for the translation of discrete-time Matlab/Simulink models into the synchronous data flow language Lustre. In[9], using the theorem prover YICES, a method is described to demonstrate that a Matlab/Simulink model is accurately translated by an automatic code generator in order to demonstrate output equivalence between the Matlab/Simulink model and the generated C-code. In [12], a verification strategy that is based on a transformation into the input language of the UCLID verifier is provided, and UCLID is the tool that is used to check whether or not the models are correct. Because UCLID supports a lower number of theories than Z3, a greater number of approximations have been utilized in the transformation. However, these methods are founded on the presumption that the synchronous nature of Simulink's semantics. Rather, the underlying semantics of Simulink are sequential, and methods such as Conditional Execution Behavior and Conditional Execution Context Propagation are built on the basis of the sequential simulation semantics. As a result, We will present a formalization of the Matlab/Simulink semantics which will focuses on verifying certain error classes.

2.4 Formal verification of Hybrid Systems

The use of Hybrid Automaton is by far the most prevalent method of hybrid system verification. Hybrid automaton are generalized finite-state machines that can be used to represent hybrid systems. [2] In [3], the author uses the finite automaton for the verification of Hybrid system. The creation of the state space of the underlying model is a disadvantage of model checking techniques. The disadvantage of this model verification strategies is the exponential number of concurrent processes required to generate the state space of the underlying model. Consequently, model checking strategies are often not scalable for bigger systems.

HyTech was the first model checker to use symbolic reach-ability analysis for hybrid systems [11]. Each reachable set is represented for a hybrid automaton with n real-valued variables by associating a finite union of n-dimensional polyhedra with each mode, where a polyhedron is represented as a conjunction of linear inequalities over variables. On the other hand, it can only be applied to linear hybrid automaton.

In this [15], the approach used to dissect the system into components, test their local safety separately, then combine them to build an overall system that satisfy a global contract, without proving the complete system. They present the essential formalism for defining the structure and behavior of components, as well as a technique for composing components in such a way that safety features can be demonstrated to originate from the safety of the components themselves.

3 Solution

In this section, we will demonstrate an algorithm and translation rule for translating Simulink model to Dafny programming language.In addition to the translation of the models, we generate proof obligations to validate the absence of certain error classes. In this report, we've looked at the following types of errors: overflow, underflow, divide by zero, and range violation.

3.1 Limitations and assumptions

There are blocks in Simulink modeling that represent calculations that are either excessively complex or for which it is even impossible to make an appropriate mapping using the types and operations that Dafny supports. Below are the limitations for translating and verifying Simulink model to Dafny:

1. The models must only include blocks from the discrete library and stateless component blocks, such as mathematical functions and signal routing blocks.
2. Our working hypothesis is that the model does not have any bus-capable blocks and composite data types, such as bus objects.
3. We don't support S-Function blocks or State-flow charts that add external code like C code or Matlab functions (M-code).
4. We assume that the model doesn't have bus-capable blocks and composite data types like bus objects.
5. We don't support time continuous blocks.
6. We are going to assume that the sample time is the same for each block.
7. We overestimate blocks such as lookup tables, signal builders, and those containing arithmetic operations that are not expressed in the specification language.

3.2 Data type mapping

Table below 1 shows the input data type mapping of Simulink models and Dafny data types.

Simulink data types	**Dafny data types**
Single and double	real (rational numbers)
boolean	bool
int8, uint8, int16, uint16..	int (Mathematical Integer)

Table 1: Data type mapping

Double and single data types can be directly mapped to one other. However, integer and real representations in Dafny are unbounded, the data types' boundaries can be retained by providing assertions for over- and underflow errors in simulink. For Example, int8 can be mapped into Dafny by restricting integer variables to have values from -128 to 127 as per the 8 bit integer values ranges from [-128, 127].

3.3 Translation rules

We will demonstrate transformation rules for Simulink model's block into methods of Dafny. We will be working on four blocks namely,

1. Direct feed-through blocks
2. Non direct feed-through blocks

3. Control flow blocks
4. Over approximated blocks

We will be using a Simulink model and it's block to translate into Dafny using our translation rules. Below figure 2 shows a simple Simulink model:

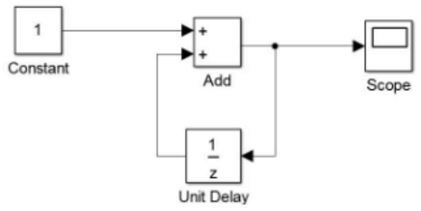
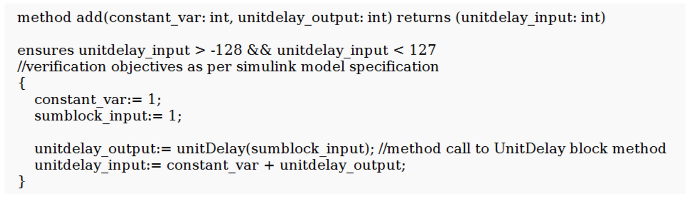


Figure 2: Simulink Model

Direct feed-through blocks are the blocks in which the output directly depends on it's input. In Matlab and Simulink, the arithmetic and logic blocks, together with the constant blocks, are examples of direct feed-through blocks. We can define such blocks in Dafny as methods with the inputs of block as parameters of the methods. Also, output of these blocks are the return values of these methods. For example, from Simulink model in Figure 2, sum block is designed with two inports, one of which is connected to the Constant block, and the other of which is connected to the UnitDelay block. The values of the input signals are added together by a Sum block. Sum block can be represented in Dafny as below:



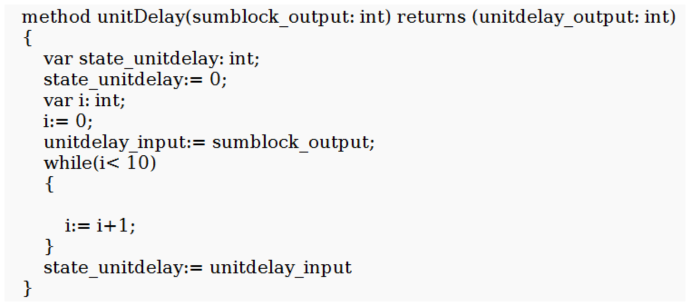
```
method add(constant_var: int, unitdelay_output: int) returns (unitdelay_input: int)

ensures unitdelay_input > -128 && unitdelay_input < 127
//verification objectives as per simulink model specification
{
   constant_var:= 1;
   sumblock_input:= 1;

   unitdelay_output:= unitDelay(sumblock_input); //method call to UnitDelay block method
   unitdelay_input:= constant_var + unitdelay_output;
}
```

Figure 3: Sum Block representation in Dafny

In our Simulink model, both the input signals of sum blocks are scalar i.e constant block with initial value as 1 and output of unit delay block, Therefore, the inputs of sum blocks are defined as Integer variables as shown in figure 3. In the Dafny representation of the Simulink model, it completely represents the behavior of the Simulink model without compromising any of it's semantics. Furthermore, it incorporates the verification of the sum block. At Sum block, only overflows and underflows can occur. Because of this, whenever we perform a translation, we first determine the data type of the output port, and then we build an automatic check for both overflow and underflow. Output of the sum block must fall within the range [-128, 127] because the output data type is an 8-bit integer. As a result, we add a post condition to cater for this range.

Finally, the Hoare Logic rule for loops is used to generate an invariant in loop, limiting the output signal boundaries of the Sum block to [-128, 127].

Non-direct feed-through blocks, the second category of blocks consists of those that do not have inputs that are fed directly via the block. Therefore, the outputs of these blocks are not directly dependent on the inputs; rather, they are based on the state of the components within the block. Nevertheless, the inputs of these blocks are used in the process of calculating the state for the subsequent step of the simulation.These kind of blocks are also referred to as stateful because they maintain an internal state variable. For example, Integrator and UnitDelay. In our example of simulink model 2, Unit delay block is non-direct-feed-through block. Below figure 4 shows the translation of the unit delay block.

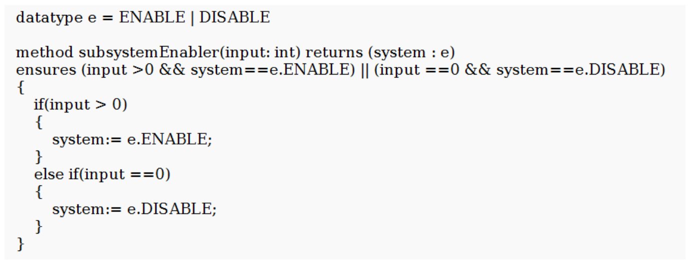


```
method unitDelay(sumblock_output: int) returns (unitdelay_output: int)
{
   var state_unitdelay: int;
   state_unitdelay:= 0;
   var i: int;
   i:= 0;
   unitdelay_input:= sumblock_output;
   while(i< 10)
   {

      i:= i+1;
   }
   state_unitdelay:= unitdelay_input
}
```

Figure 4: Unit delay block representation in Dafny

In the translation of Unit Delay block to Dafny, we have assumed that the ten iteration of a while loop can be considered as a unit delay and designed a method for unit delay block. UnitDelay is a non-direct-feedthrough block in the model. Thus, two variables are created. As mentioned in Example, since the block's starting value is 0, we have initialised UnitDelay block's state variable to 0. At the beginning of the loop body, the unit delay output variable is assigned to the block's output variable. At the end of the loop, the state variable is updated to reflect the new state, which is the value of the Sum block.

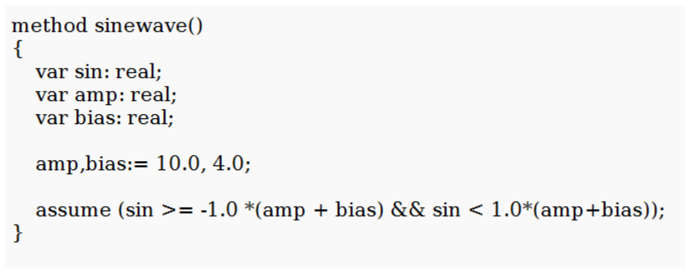
Control flow block : Matlab/Simulink models control flow with conditionally executed subsystems such blocks are control flow blocks such as switch block. An enabled subsystem, for instance, comprises a specific block that regulates execution, such as an EnablePort block that enables or disables the subsystem. The subsystem is enabled if the input of an EnablePort block is greater than 0. The subsystem is disabled if the input of an EnablePort block equals zero. We can define the enable and disable options as an enum in Dafny. We have defined the switch block behaviour using if..else if..else in Dafny. The enabled system can be translated to Dafny as shown in below figure 5: As if input is greater than 0, the subsystem is enabled and if input is 0 it is disabled, such behaviour is achieved in Dafny providing conditions to if and else if controls. In addition to that, we have verified this behaviour by providing a post-condition which ensures that the if input value is 0, subsystem will receive a enable signal and if input is 0, subsystem will receive disable signal.



```
datatype e = ENABLE | DISABLE

method subsystemEnabler(input: int) returns (system : e)
ensures (input >0 && system==e.ENABLE) || (input ==0 && system==e.DISABLE)
{
   if(input > 0)
   {
      system:= e.ENABLE;
   }
   else if(input ==0)
   {
      system:= e.DISABLE;
   }
}
```

Figure 5: Control flow block representation in Dafny

Over-approximated blocks: There are several blocks in Simulink that can't be properly converted into Dafny. These building blocks include elements like complicated mathematical functions, often known as S-functions. In addition, there are block, such as lookup tables, where a direct translation would significantly add to the complexity of the specification and, as a result, the amount of work required to verify it. In our translation, such blocks are always overly approximated. For example, there's a block Sine Wave that generates a sine wave based on how much time has passed in the simulation. The amplitude, frequency, phase, and bias of a Sine Wave block are all controlled by parameters within the block.



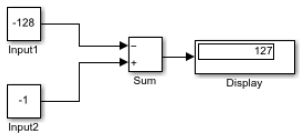
```
method sinewave()
{
   var sin: real;
   var amp: real;
   var bias: real;

   amp,bias:= 10.0, 4.0;

   assume (sin >= -1.0 *(amp + bias) && sin < 1.0*(amp+bias));
}
```

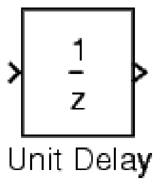
Figure 6: Sine wave block representation in Dafny

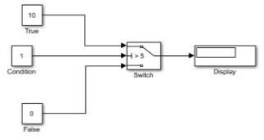
The equation of sine wave is $y = amplitude * sin(frequency * x + phase) + bias$. When attempting to get a close approximation of the Sine Wave, we make advantage of the fact that $-1 <= sin(x) <= 1$. For every Sine Wave block with an amplitude of a, bias of b, and a simulation time of t, the equation $a + b <= asin(t) + b <= a + b$ holds. In this example, we have used *assume* keyword in Dafny to approximate the behaviour of sine wave block of Simulink. Below figure shows the sine wave block representation in Dafny:

Below table shows the different Simulink blocks and it's representation in dafny with the help of transformation rules we have defined along with it's verification metrics:

Simulink blocks	**Graphical representation of Simulink blocks**	**Dafny representation**

Sum block		method sum(input1: int, input2: int) returns (output: int)
		//define pre-conditions if required
		Ensures output==input1+input2
		{
		   output:= input1+input2;
		}

Unit delay block		method unitDelay(input: int) returns (output: int)
		{
		   var state_unitdelay: int;
		   state_unitdelay:= 0;
		   var i: int;
		   i:= 0;
		   output:= input
		   while(i< 10)
		   {
		      i:= i+1;
		   }
		   state_unitdelay:= input
		}

Switch block	 10
	True
	1
	Condition
	0
	False
	Switch
	Display
		method switch_block(in1: int, in2: int, cond_var: int) returns(out: int)
		//define pre-conditions if required
		ensures (out==in1 && cond_var>5)
		||(out==in2 && cond_var<=5)
		{
		   if(cond_var>5)
		   {
		      out:= in1;
		   }
		   else{
		      out:=in2;
		   }
		}

Sine wave	Sine Wave
	Scope Y
		method sinewave(amp: real, bias: real)
		{
		   var sin: real;
		
		   assume (sin >= -1.0 *(amp + bias) && sin <= 1.0*(amp+bias));
		}

Table 2 Simulink block and it's representation in Dafny

3.4 Translation and verification Algorithm

We've designed an algorithm to translate a Simulink model into Dafny. Even though we are translating a Simulink block to Dafny using translation rules which are defined in section 3.3, we need a certain steps which will translate entire Simulink model to dafny, so that we can verify the entire model. This can be useful in future to define and verify global properties of a model. Below are the steps to transform and verify the Simulink Model to Dafny:

1. Identify the different blocks which can be translated to Dafny
2. Figure out the input values for the each blocks from steps 1 and convert it into Dafny input variables for methods as per the 3.2.
3. Convert each block into methods in Dafny with it's input variables as per the rules desribed in section 3.3. Figure out the pre-conditions and post-conditions of these blocks as per the specification of Simulink model.
4. Identify the blocks which inputs are dependent on the output of other blocks and convert it into Dafny methods.
5. First lines of such methods(mentioned in step 4), initialize the required input variables for the blocks using the method call to the corresponding methods an figure out the post-conditions from the specification of Simulink models.
6. If there exists a loop in Simulink model, identify the blocks which are in the loop. Create a method in Dafny with the loop body in it. Loop body will have the different method calls to the identified blocks.
7. Figure out the loop guard and loop invariant using Hoare Logic for loop and function calls.[1]

3.5 Illustrating Example: Temperature Control System

In this section, we will use the temperature management system presented in figure 7 as an example to explain our transition from Simulink to Dafny. We are using this example from paper [17].

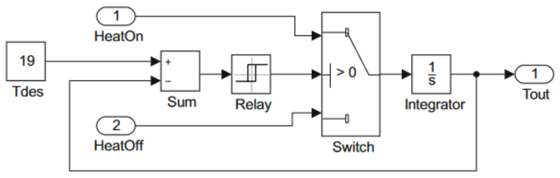
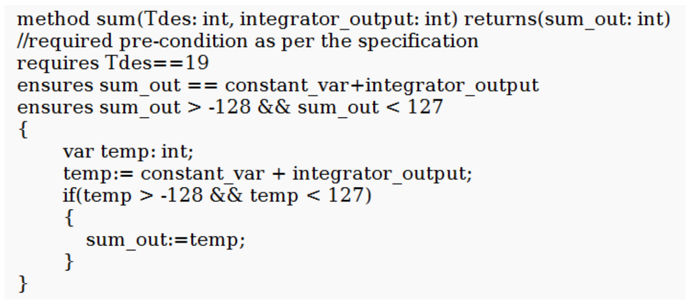


Figure 7: Temperature Control System

Temperature Control System is designed for an operating room and permits fine-grained temperature control: it accepts heating and cooling values (representing the temperature gradient) as input and adjusts the current temperature accordingly. Tdes defines the desired temperature as a constant block. A Switch block determines whether a heating or cooling value is utilized in a feedback loop. The Integrator block calculates the output temperature (Tout) by integrating the input values over time, and the result is used to control the Switch. The control signal is relayed to prevent quick switching, i.e., the control signal only changes if the output temperature deviates from the desired temperature by more than a set minimum or maximum value.

In the example of Temperature Control System, we are translating simulink blocks into it's Dafny representation using the Transformation rules defined in section 3.3. The transformation of Simulink to Dafny consistes of three methods for blocks sum, relay and switch block. As we don't support contineous block in simulink, we cannot map the integrator block to Dafny representation.

Sum block has two inputs i.e. constant Tdes and output of integrator block. We have translated this block into Dafny as shown below figure 8:

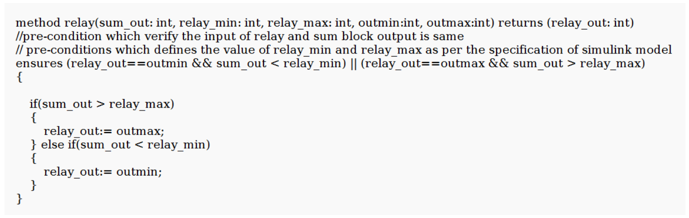


```
method sum(Tdes: int, integrator_output: int) returns(sum_out: int)
//required pre-condition as per the specification
requires Tdes==19
ensures sum_out == constant_var+integrator_output
ensures sum_out > -128 && sum_out < 127
{
      var temp: int;
      temp:= constant_var + integrator_output;
      if(temp > -128 && temp < 127)
      {
         sum_out:=temp;
      }
}
```

Figure 8: Transformation of sum block of temperature control system to Dafny

As we are using int8 datatype of Simulink, we have restricted the output values between [-128, 127] using the *ensures* keyword in Dafny which is used to define post condition of a method. At Sum block, only overflows and underflows can occur. Because of this, whenever we perform a translation, we have built a check for both overflow and underflow errors.

The Relay block's output can swap between two given values. When the relay is activated, it remains active until the input falls below the Switch off point parameter value. When the relay is off, it remains off until the input exceeds the parameter's value for Switch on point. The block accepts a single input and produces a single output. We have translated relay block into Dafny as shown below figure 9:

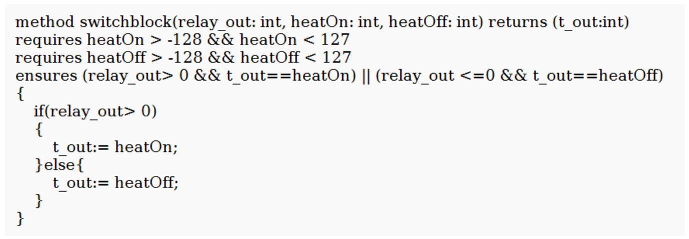


```
method relay(sum_out: int, relay_min: int, relay_max: int, outmin:int, outmax:int) returns (relay_out: int)
//pre-condition which verify the input of relay and sum block output is same
// pre-conditions which defines the value of relay_min and relay_max as per the specification of simulink model
ensures (relay_out==outmin && sum_out < relay_min) || (relay_out==outmax && sum_out > relay_max)
{

  if(sum_out > relay_max)
  {
    relay_out:= outmax;
  } else if(sum_out < relay_min)
  {
    relay_out:= outmin;
  }
}
```

Figure 9: Translation of relay block of temperature control system to Dafny

In this, we have implemented the behaviour of relay block in Dafny and added a verification conditions to it. Even though relay block accepts single input, relay block has different parameters incorporated which we have defined as a parameters to method for relay block.

The Switch block is a toggle switch that allows just one of its two inputs to reach its output. It selects the respective value by checking the condition. Below figure 10 shows the behaviour of switch block for Temperature Control system to Dafny:



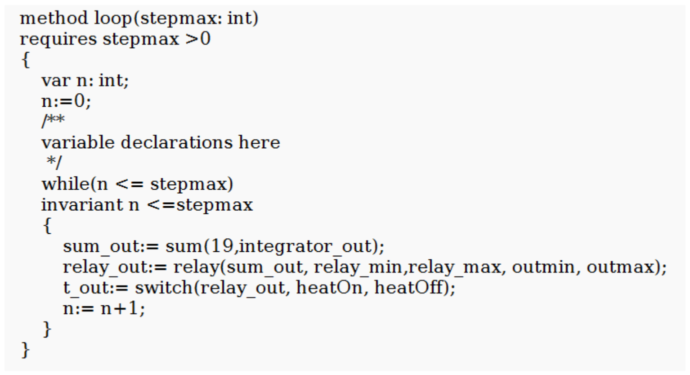
```
method switchblock(relay_out: int, heatOn: int, heatOff: int) returns (t_out:int)
requires heatOn > -128 && heatOn < 127
requires heatOff > -128 && heatOff < 127
ensures (relay_out> 0 && t_out==heatOn) || (relay_out <=0 && t_out==heatOff)
{
  if(relay_out> 0)
  {
    t_out:= heatOn;
  }else{
    t_out:= heatOff;
  }
}
```

Figure 10: Translation of Switch block of temperature control system to Dafny

In this, we have added verification conditions as per the switch block's behaviour. Also, we have added preconditions to check the overflow and underflow errors for the input values heatOn and heatOff.

The sum, relay, and switch blocks in the Temperature Control System 7 are in a loop. As per our algorithm in section 3.4, we have designed a new method, which will consist of a loop body that calls the methods that correspond to the each blocks.The method of loop structure is shown in below figure 11

In order for our translation to accurately reflect the simulation semantics, we have made the following assumptions: (1) we assume that the model is run for an finitely and unbounded number of steps; (2) we use the symbolic constant stepmax as the simulation bound; and (3) we assume that the number of simulation steps is greater than or equal to the number of simulation steps.



```
method loop(stepmax: int)
requires stepmax >0
{
   var n: int;
   n:=0;
   /**
   variable declarations here
    */
   while(n <= stepmax)
   invariant n <=stepmax
   {
      sum_out:= sum(19,integrator_out);
      relay_out:= relay(sum_out, relay_min,relay_max, outmin, outmax);
      t_out:= switch(relay_out, heatOn, heatOff);
      n:= n+1;
   }
}
```

Figure 11: Translation of loop structure of temperature control system to Dafny

With this approach, we are able to validate the model for an arbitrary number of simulation steps by using inductive invariant checking.

Using our translation rules and algorithm, we are able to translate and verify the Temperature Control System without any errors. In this, we are able to verify that none of signal will have any underflow or overflow errors as we are restricting the values within certain limits, such as for 8 bit integer, we are restricting it's value between -127 to 128. As we do not have any translation rules for time continuous blocks, we are unable to translate the Integrator block on the model into Dafny.

3.6 Illustrating Example: Water Tank Level

We are using this example from the Water Tank Level example used in the paper [12]. We will use the translation rules mentioned in section 3.3.

As shown in Figure, there is a fluid-filled tank with one pipe going in and two going out. Each pipe has a valve with the name V1, V2, or V3 that can be open or closed.

In MSS, a valve is linked to a switch. The level of fluid in the tank (h) and the flow through valve V3 can be measured by sensors (em flow). The state machine in the figure is used by a controller to control the system. When the system is first turned on, V1 is closed and V2 is open. When the tank's height goes above 10 units, the V1 and V3 outlets are opened. When the flow through V3 goes over 5 units, the flow through V2 shuts off. When the fluid level drops below 8 units, the inlet V2 opens and the outlet V1 closes.

In the Simulink model for Water tank control System, there are three swicth blocks, one sum block, one controller i.e for-each subsystem block and one in-

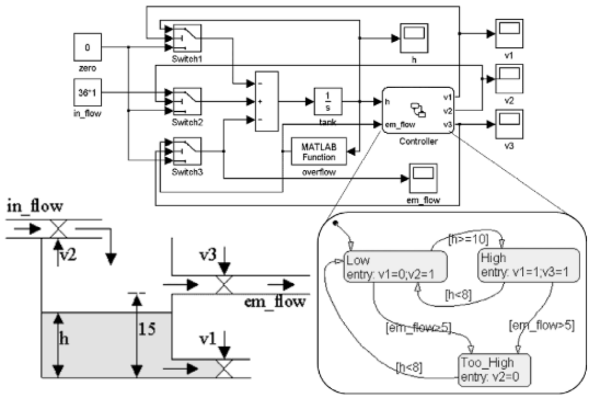
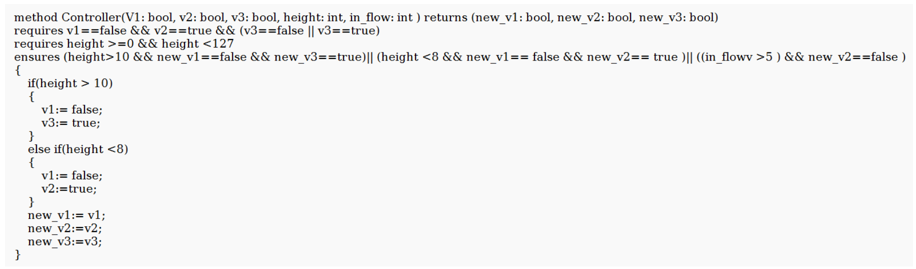


Figure 12: Water Tank Level Control System with three valves

tegrator block. Out of which, we can translate all blocks to Dafny and verify it except Integrator block as it is continuous block and we don't support continuous blocks. For translating these blocks to Dafny, we are using Transformation rules for these blocks as discussed in section 3.3.

Below figure shows the typical behaviour of subsystem block i.e Controller in the Water Tank Level Control System.

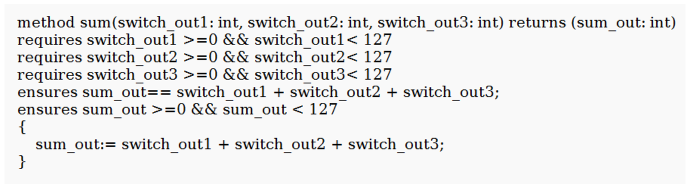


```
method Controller(V1: bool, v2: bool, v3: bool, height: int, in_flow: int ) returns (new_v1: bool, new_v2: bool, new_v3: bool)
requires v1==false && v2==true && (v3==false || v3==true)
requires height >=0 && height <127
ensures (height>10 && new_v1==false && new_v3==true)|| (height <8 && new_v1== false && new_v2== true )|| ((in_flowv >5 ) && new_v2==false )
{
   if(height > 10)
   {
      v1:= false;
      v3:= true;
   }
   else if(height <8)
   {
      v1:= false;
      v2:=true;
   }
   new_v1:= v1;
   new_v2:=v2;
   new_v3:=v3;
}
```

Figure 13: Translation of Controller block in Dafny

The method controller is designed as per the specification of the controller block which is depicted in Figure 12. Also, we have added the input checks for overflow and underflow errors. The three output of the controller unit is of type Boolean as the output is forwarded as the conditions to the all switch blocks.

There are three switch blocks in the Simulink Model for Water Tank Level Control System, each switch block's output is forwarded as input to the add block, the translation of sum block is as shown below figure 14:

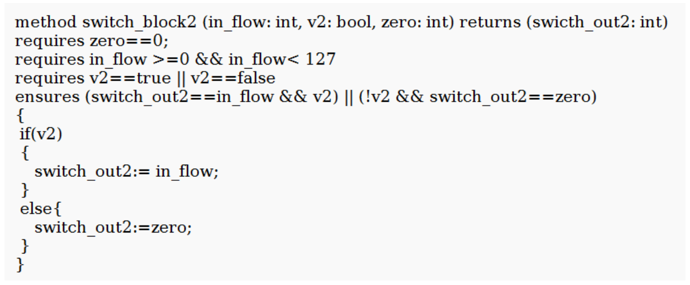


```
method sum(switch_out1: int, switch_out2: int, switch_out3: int) returns (sum_out: int)
requires switch_out1 >=0 && switch_out1< 127
requires switch_out2 >=0 && switch_out2< 127
requires switch_out3 >=0 && switch_out3< 127
ensures sum_out== switch_out1 + switch_out2 + switch_out3;
ensures sum_out >=0 && sum_out < 127
{
   sum_out:= switch_out1 + switch_out2 + switch_out3;
}
```

Figure 14: Translation of sum block in Dafny

In the translation shown above, sum block has three inputs which are outputs of three switch blocks and the translation of sum block has been carried out without losing any semantics of Simulink model and it also been verified by checking underflow and overflow errors along with the specification requirements.

Switch blocks can easily translated into the Dafny using the transformation rules for Control flow blocks, also we can refer to the example 10, which translates the switch block of Temperature Control System into Dafny method. Below figure shows the one of the switch block(out of three switch blocks) behaviour implementation in Dafny:



```
method switch_block2 (in_flow: int, v2: bool, zero: int) returns (swicth_out2: int)
requires zero==0;
requires in_flow >=0 && in_flow< 127
requires v2==true || v2==false
ensures (switch_out2==in_flow && v2) || (!v2 && switch_out2==zero)
{
 if(v2)
 {
   switch_out2:= in_flow;
 }
 else{
   switch_out2:=zero;
 }
}
```

Figure 15: Translation of switch block in Dafny

Above figure, shows the verification specification for Switch block in Water Tank Level Control System along with the translation. We are able to translate and verify the Water Tank Level Control System without any errors. In this, we are able to verify the entire model and none of the signal of model will have any underflow or overflow errors as we are restricting the values within certain limits, such as for 8 bit integer, we are restricting it's value between -127 to 128. In this,

4 Evaluation

In this section, we are evaluating our approach of translating and verifying a Simulink model to Dafny. Our approach supports different basic block types and the common parameter configuration for each of those block types. The Simulink model have up to 263 blocks and 364 signal lines. Among these are the following types of blocks: unit delays, arithmetic blocks, logic blocks, switches, subsystems, constants. However, in the future, further translation rules for various block types and parameter combinations can be implemented. We mainly focus on Direct-feed through blocks, non-direct feed through blocks, control flow graphs and we over approximate some blocks. This approach does not support blocks like S-function, state flow charts or blocks which uses external codes like C-code or M-code. It also does not support the bus-capable blocks.

We used two industrial Simulink model examples to demonstrate applicability of our algorithm. The first is a model of a temperature control system, and the second is a tank level control model. As a demonstration, we explored the possible safety measure: No signal or variable can underflow or overflow. Our approach successfully verifies four error classes: overflow, underflow, divide by zero and range violations.

In [13], they have presented an approach to translate Simulink to Why3 model, however it only focuses the certain properties such as stability of the Simulink models. Also, in [6], they have presented an approach to automatically translate the Simulink model to the input checker NuSMV, they are only verifying the safety property of the model. In contrast, in out approach we check the specific error classes such as divide by zero, range violation, overflow an underflow etc. In addition to this, we were successfully able to verify loop structure in Simulink.

In our approach, we have translated the Simulink blocks to the Dafny method and verified it by adding contracts in Dafny i.e pre conditions and post-conditions. We provided the essential preconditions and invariant for determining the absence of variable under- or overflows. Also we can verify the loop structure in Simulink using inductive invariant verification. As shown in the example of 3.5, we have designed a method with loop body in it and using the assumption that the model runs for finite number of steps and using a variable stepmax for finite step, we provide a loop guard. We specify a invariant using the specification and a loop guard as per the Hoare Logic rules for loop.

Our approach is very much similar to the verification approach used in [14] as they are also checking four error classes: range violation, over-flow, under-flow, divide by zero errors. However, they have used an intermediate representation of a model in Control Flow Graph and then translating it into Boogie language. In our approach, we are not representing our model in any intermediate graph or tree. Using our translation rules, we are directly representing it into specified language i.e Dafny. Unlike the approach used in [14], according to our strategy, it is not required to have any intermediate representation, and as a result, we do not incur the additional burden of creating an intermediate representation.

Our approach of translating Simulink models to Dafny language can be reused to translate and verify the model to JML(Java Modelling language) as both Dafny and JML language uses the same verification paradigm.. Java Modeling Language (JML), is a specification language for Java programs that follows to the design-by-contract and makes use of preconditions, postconditions, and invariants written in the Hoare logic.

We have discussed few approach for verification of Simulink models and Hybrid Systems in our Related Work section 2. In section 2.2, similar to our approach, Simulink design verifier also checks for the overflow, underflow, divide by zero and access violation errors. However, it has a scalability issue and it fails to verify the models with large state space. In Formal verification of Simulink models 2.3, the premise that the semantics of Simulink is synchronous underpins the approaches. Simulink's true simulation semantics are sequential, and methods such as Conditional Execution Behavior and Conditional Execution Context Propagation are developed in terms of this sequential simulation semantics.Also, it does not support the verification and translation of discrete blocks of Simulink models. We propose a more accurate formalisation of the Simulink semantics as well as we also provided the translation and verification of discrete blocks. In formal verification of hybrid systems discussed in section 2.4, the model checking strategies are not scalable for bigger systems as it requires lots of state space for exponential number of concurrent processes.

We have designed translation rules and algorithm to translate a Simulink model to Dafny without losing it's semantics and able to verify the certain error classes successfully. We are unable to design rules to translate time-continuous blocks. Also, we don't support some Simulink model input as there is no direct mapping to Dafny datatypes. Our approach and translation rules can be used in different verification language such as JML(Java Modelling language).

5 Conclusion

In this paper, we present our methodology for the formal verification of Matlab/Simulink models with the Dafny language The fundamental principle of the approach is the transformation of a given Matlab/Simulink model into a specification in the Dafny, followed by the verification of the model using the Dafny program verifier. The translation is based on the Simulation semantics of Matlab/Simulink models. In addition to the translation or safe over-approximation of the set of supported blocks, the translation also provides (1) loop invariants for the loops, (2) verification goals for each block according to the error classes of interest, and (3) a Dafny specification depending on the desired verification strategy. All of these are dependent on the translation of the set of supported blocks.

We have evaluated this approach by two industrial Simulink examples and we are successfully able to show that verification avoids the for error classes such as divide by zero, overflow, underflow, range violations.

In future, the implementation of additional translation rules and it's verification can be done for a wide variety of block types and parameter/input combinations. Also, we can verify the common properties(global properties) of Simulink model which are applicable for all the blocks using the classes and invariant for the classes in Dafny.

References

[1] Vaughan R. Pratt. “SEMANTICAL CONSIDERATIONS ON FLOYD-HOARE LOGIC”. In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976, pp. 109–121. DOI: 10.1109/SFCS.1976.27.

[2] Rajeev Alur et al. “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems”. In: *Hybrid systems*. Springer, 1992, pp. 209–229. doi:10.1007/3-540-57318-6$_3$0.

[3] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. “HyTech: A model checker for hybrid systems”. In: *International Conference on Computer Aided Verification*. Springer. 1997, pp. 460–463. doi: 10.1007/3-540-63166-6$_4$8.

[4] Stavros Tripakis et al. “Translating discrete-time Simulink to Lustre”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 4.4 (2005), pp. 779–818.

[5] Mike Barnett et al. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. ISBN: 978-3-540-36750-5. doi:10.1007/11804192$_1$7.

[6] B. Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. “Tool for Translating Simulink Models into Input Language of a Model Checker”. In: *Formal Methods and Software Engineering*. Ed. by Zhiming Liu and Jifeng He. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 606–620. ISBN: 978-3-540-47462-3. doi:10.1007/3-540-64358-3$_4$4.

[7] Grégoire Hamon et al. “Simulink design verifier-applying automated formal methods to simulink and stateflow”. In: *Third Workshop on Automated Formal Methods*. 2008.

[8] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. doi:10.1007/978-3-540-78800-3$_2$4.

[9] Michael Ryabtsev and Ofer Strichman. “Translation validation: From simulink to c”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 696–701. doi:10.1007/978-3-642-02658-4$_5$7.

[10] K. Rustan M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4. doi:10.1007/978-3-642-17511-4$_2$0.

[11] Rajeev Alur. “Formal verification of hybrid systems”. In: *Proceedings of the ninth ACM international conference on Embedded software*. 2011, pp. 273–278. doi:10.1145/2038642.2038685.

[12] Paula Herber, Robert Reicherdt, and Patrick Bittner. “Bit-precise formal verification of discrete-time MATLAB/Simulink Models using SMT Solving”. In: *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. 2013, pp. 1–10. DOI: 10.1109/EMSOFT.2013.6658586.

[13] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. “Formal verification of control systems’ properties with theorem proving”. In: *2014 UKACC International Conference on Control (CONTROL)*. 2014, pp. 244–249. DOI: 10.1109/CONTROL.2014.6915147.

[14] Robert Reicherdt and Sabine Glesner. “Formal Verification of Discrete-Time MATLAB/Simulink Models Using Boogie”. In: *Software Engineering and Formal Methods*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer International Publishing, 2014, pp. 190–204. ISBN: 978-3-319-10431-7. DOI: 10.1007/978-3-319-10431-7_14.

[15] Andreas Müller et al. “A Component-Based Approach to Hybrid Systems Safety Verification”. In: *Integrated Formal Methods*. Ed. by Erika Ábrahám and Marieke Huisman. Cham: Springer International Publishing, 2016, pp. 441–456. ISBN: 978-3-319-33693-0. doi: 10.1007/978-3-319-33693-0_28.

[16] Timm Liebrenz, Paula Herber, and Sabine Glesner. “Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X”. In: *International Conference on Formal Engineering Methods*. Springer. 2018, pp. 89–105. doi: 10.1007/978-3-030-02450-5_6.

[17] Timm Liebrenz, Paula Herber, and Sabine Glesner. “A Service-Oriented Approach for Decomposing and Verifying Hybrid System Models”. In: *Formal Aspects of Component Software*. Ed. by Farhad Arbab and Sung-Shik Jongmans. Cham: Springer International Publishing, 2020, pp. 127–146. ISBN: 978-3-030-40914-2. DOI: 10.1007/978-3-030-40914-2_7.

A Appendix

Git repository related this research is located at: