Automated Translation of MATLAB Simulink/Stateflow Models to an Intermediate Format in HyVisual

Rajarshi Ray
Chennai Mathematical Institute

June, 2007

Dissertation submitted to Chennai Mathematical Institute,
in partial fulfilment of the requirements for the
award of the MSc degree in Computer Science

CERTIFICATE

This is to certify that the dissertation titled *Automated Translation of MATLAB Simulink/Stateflow Models to an Intermediate Format in HyVisual* is a bonafide record of work done by Rajarshi Ray under my supervision.

This dissertation is to be submitted to Chennai Mathematical Institute in partial fulfillment of the requirements of the M.Sc. degree.

Dr. Swarup Mohalik
General Motors R&D
India Science Lab
Bangalore.

Dated:

Acknowledgements

I would like to express my sincere gratitude to my project advisor Dr. Swarup Mohalik (Senior Researcher, GM R&D) for his guidance and for patiently listening and answering to all my doubts and questions. I am thankful to Dr. Suresh Jeyaraman (Researcher, GM R&D) for helping me with MATLAB. I am grateful to General Motors Research and Development, India Science Lab, Bangalore for providing me a 6 months internship.

My sincere gratitude to Prof. Madhavan Mukund (Chennai Mathematical Institute), Prof. Ramesh (Technical Fellow, GM R&D) and Dr. Sathyaraja (LGM, GM R&D) without whom my internship at GM R&D wouldn't have been possible.

Rajarshi Ray

Contents

List of Figures

Chapter 1

Introduction

Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, etc. As a result, scientists and engineers are looking for easy and reliable techniques to design, develop, test and verify these systems.

With model based design and development becoming a trend, industries use design and simulation tool sets like MATLAB and Mathematica. MATLAB Simulink/Stateflow (SL/SF) is a software for modeling, simulating and analyzing dynamic systems. It supports linear and non-linear systems, modeled in continuous time, sampled time or a hybrid of two. Systems can also be multirate, i.e, have different subsystems that are sampled or updated at different rates. Simulink is basically an add-on library to MATLAB with a number of blocks like Integration block, Summation block etc with the help of which one can design and capture the dynamic behavior of a system under consideration. To capture the discrete control states, one generally uses Stateflow which is a component of Simulink. It allows hierarchical state machine diagrams Statecharts to be combined with flowcharts. SL/SF is a widely accepted tool in the industry for model based development of systems.

As the complexity of safety critical systems grow, exhaustive testing becomes time consuming and incomplete. Hence, there is need of formal verification of the models against stated specifications. However, since SL/SF does not have a published formal semantics, SL/SF models need to be translated to semantically equivalent formal model which can then be verified using the existing tools of verification. Further, one would like to subject the model to different kinds of analysis provided by different analysis, verification and simulation tools. But unfortunately, different tools like Charon, HyTech, Checkmate, Shift etc, though they model and analyze hybrid system, differ subtly in the semantics and also in the syntax of their inputs. As a result, one needs to translate an SL/SF model to these tools each time beginning from

the SL/SF syntax. This close coupling is inconvenient because each time the SL/SF format changes (SL/SF is proprietary and the format changes have occurred many times alongwith version changes), the translators have to be reimplemented. An **intermediate format** to represent a hybrid system can solve this problem, in which case Simulink models can be translated to the intermediate format and from this it can be translated to Charon, HyTech, Checkmate etc. Fig 1.1 illustrates the exchange mechanism. When a format change occurs in SL/SF, only the front-end parser should change in the SL/SF to intermediate format tool implementation.

In my work, I deal with the following 3 issues.

1. Deciding an intermediate format for representing hybrid systems.
2. To come up with a translation mechanism from MATLAB SL/SF model to the chosen intermediate format.
3. To automate the translation process.

The report is organized as follows. In Chapter 2, we discuss about Simulink and Stateflow structure and semantics to some extent to familiarise with the source model. In Chapter 3, we describe the basic requirements of the intermediate format and decide that the chosen intermediate format's syntax should follow hybrid automata network [3] concept. We also justify the reason why we use HyVisual model [9] to represent the hybrid automata network. In Chapter 4, we give the implementation details of the translator. Chapter 5 has some discussions and possible extensions of the current work.

1.1 Related Work

Most of the existing work in the context addresses the problem of end-to-end translations. Tripakis et al.[7] presents a method of translating discrete-time Simulink models to Lustre programs. The method consists of three steps: type inference, clock inference and hierarchical bottom-up translation. It also explains and formalizes the typing and timing mechanisms of Simulink. The method has been implemented in a prototype tool called S2L.

The work reported in [10] motivates a methodology called "invisible formal methods" that provides a graded sequence of formal analysis technologies ranging from extended type checking, through approximation and abstraction, to model checking and theorem proving. As an instance of invisible formal methods, it describes the formal semantics of a fragment of Stateflow based on a modular representation called communicating pushdown automata and shows how this semantics can be used to analyze simple prop-

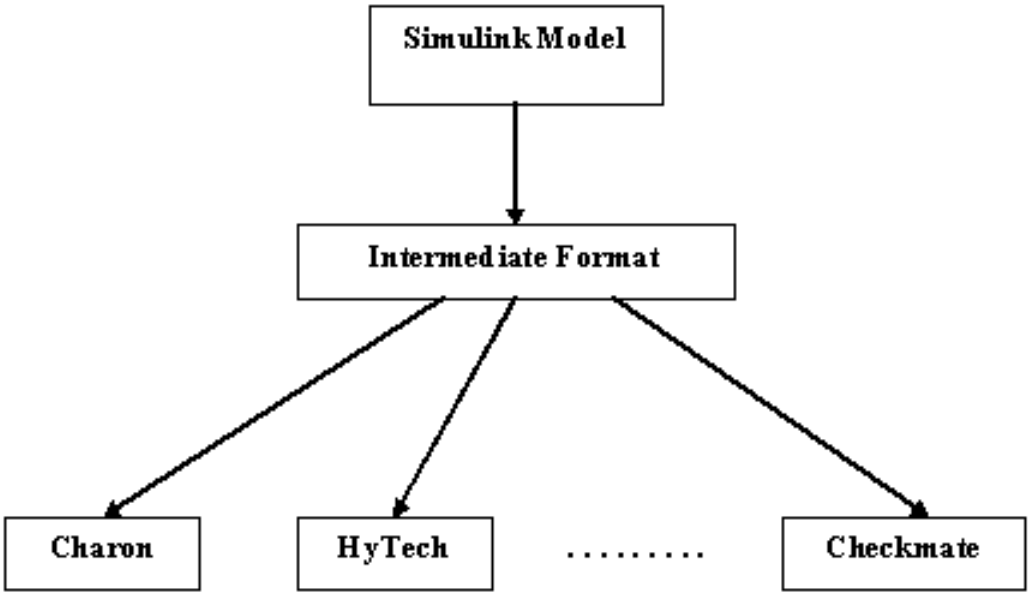


Figure 1.1: Need for an Intermediate Format in End to End Translation from Simulink to Other Tools of Analysis, Verification and Simulation.

erties of Stateflow models. It also describes a translation mechanism from SL/SF models to SAL, although there is no implemented tool based on the proposed idea.

[11] presents a correct by construction model transformation technique from Simulink to Ascet and vice-versa. The approach is based on the use of a common formal model, namely the synchronous reactive model of computation, which is used as the common ground to interpret system specifications given with different underlying models.

[4] describes a translation algorithm that converts a well-defined subset of the SL/SF into an equivalent hybrid automata. The translation has been specified and implemented using a metamodel-based graph transformation

tool called GReAT. We explore this approach of model transformation and discuss about it in Chapter 5.

Chapter 2

Simulink and Stateflow

Understanding the Simulink and Stateflow structure and semantics is essential for a model translation project where the source model is an SL/SF model. In this chapter, we give an overview of these tools.

2.1 Simulink

Simulink is a software from The Mathworks Inc. for modeling, simulating and analyzing dynamic systems. Most of the embedded and real time systems that we encounter in real life are hybrid systems, i.e, they exhibit both continuous and discrete behavior. These kind of hybrid systems can be modeled, simulated and analyzed using Simulink (with the help of Stateflow, which is a part of Simulink).

Systems can be modeled in Simulink by creating a network of blocks dragged from the Simulink block library and dropped in the GUI editor and connecting appropriate ports. We list some typical blocks in the following :

1. Sum, Subtract, Product and Divide
2. Abs, Gain, Saturation
3. Zero-delay, Memory, Integration
4. If block, General Expression block
5. Switch, Multiport Switch
6. Constant, Discrete Pulse Generator
7. Transfer Function blocks

8. Lookup Table, Stateflow

The connections between the blocks define the input output dependencies. Fig 2.1 shows an example Simulink model. Simulink allows hierarchical design which is essential in large industrial systems. In the above example, Vehicle, Transmission, Engine and Threshold Calculation are top level blocks which have further refinements. The refinement of the Vehicle block is shown in Fig 2.2. Shift_Logic is a Stateflow block. We will discuss about Stateflow in the next section.

The set of Simulink blocks that is in the scope of the translator includes Continuous, Discontinuous, Discrete, Logic operation, Signal Routing and Source blocks.

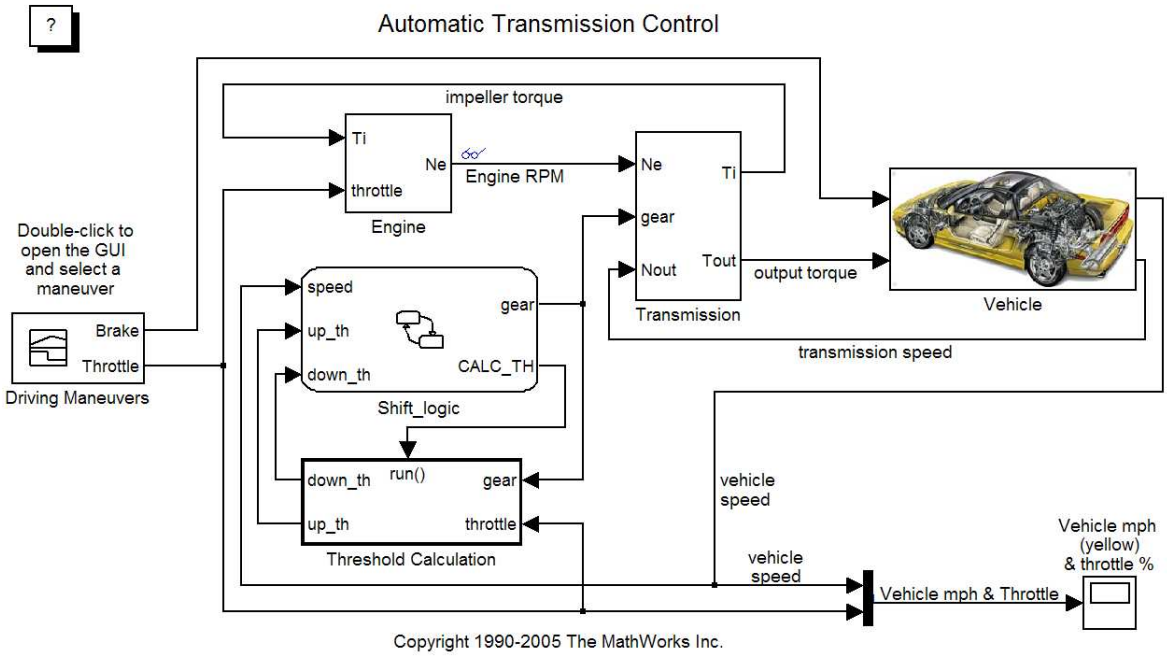


Figure 2.1: Simulink Model - An Example

2.2 Stateflow

Stateflow is an interactive graphical design tool that works with Simulink to model and simulate event-driven systems, also called reactive systems. It provides a graphical editor on which the Stateflow graphical objects dragged from the design palette can be put to create finite state machines. Stateflow

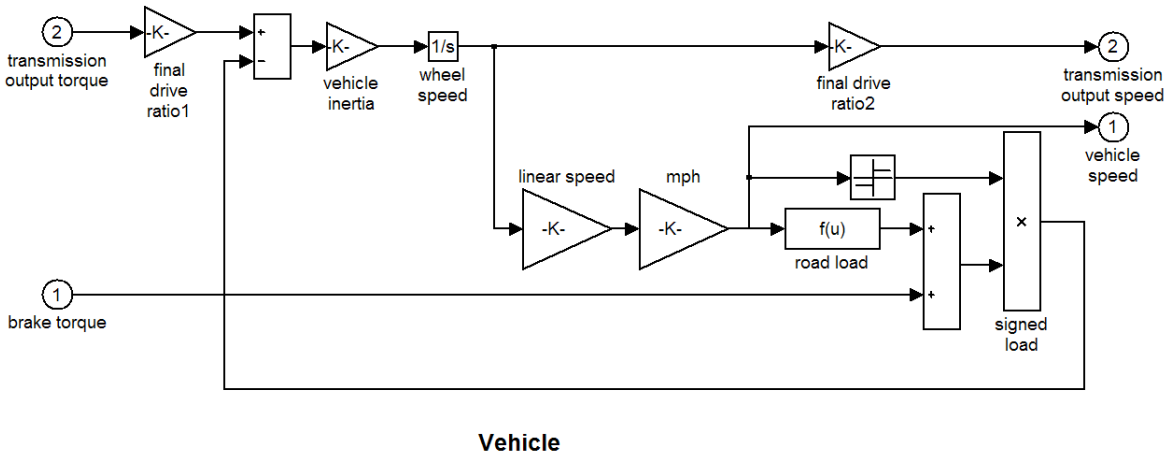


Figure 2.2: Refinement of the Vehicle Block in ATC example

extends the ease of modeling by adding hierarchy, parallelism, events, actions and history. Stateflow blocks are treated as standard Simulink blocks and hence can be integrated in Simulink models, for example the Shift_Logic Block in the ATC model is a Stateflow block. Figure 2.3 shows the refinement of the Shift_Logic block.

States form the basic object in Stateflow and reflect modes in a dynamic system. States can be active or inactive. Active state means that the Stateflow is in that mode. Multiple states can be active during a state, depicting parallelism. Events and conditions cause the states to change from inactive to active states and vice versa. States in Stateflow can have hierarchy. For example, gear_state is the parent of first, second, third and fourth state in the Shift_Logic block. The events and transitions between them are contained in gear_state. Every state has a decomposition that dictates what kind of substates it can contain. All substates must be of the same type as of superstate's decomposition. There are 2 types of state decompositions namely OR and AND decomposition. In OR decomposition, only one substate of the parent state can be active at a time. For example, gear_state has OR decomposition in the Shift_Logic example where first, second, third and fourth are its OR substates. OR substates have solid borders. In AND decomposition, the child states are parallel states. All the substates are active at the same time. For example, gear_state and selection_state are AND states and the parent is the Stateflow chart having AND decomposition. States have labels which denote the state's name, *entry* actions, *during* actions, *exit* actions

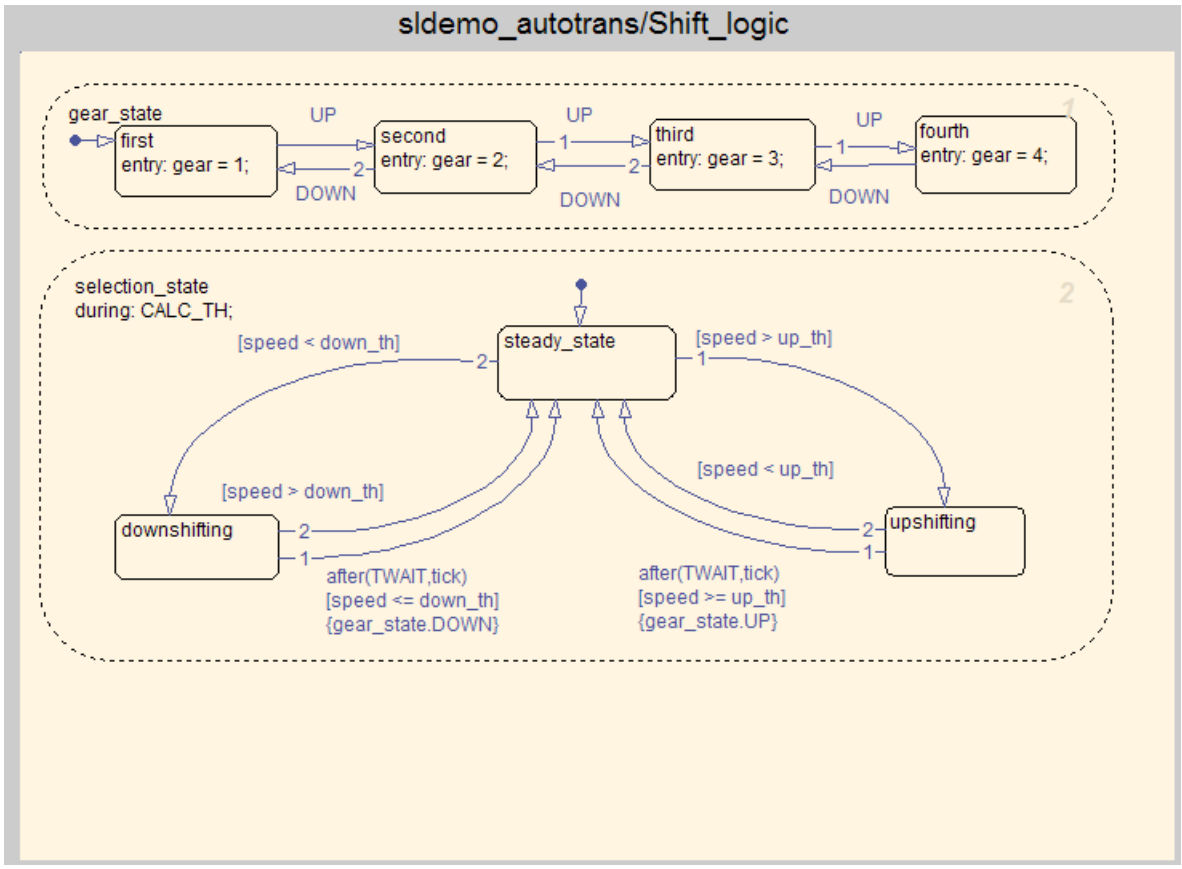


Figure 2.3: Refinement of Shift_Logic block in the ATC example

and *on event* actions.

- **Entry actions** define the actions to be taken when the state is entered or activated.
- **During actions** define the set of actions to be taken when the state is already active and some event occurs.
- **Exit actions** define the set of actions to be taken when the state becomes inactive from active.
- **On event actions** define the actions to be taken when a state is active and the mentioned event occurs.

Transitions in Stateflow means a jump from some source state to some target state. Transitions have a label associated with it. The label can consist

of an event, a condition, a condition action and/or a transition action having the following format.

event[condition]{condition action}\transition action

- **Event** specifies the event that should cause the transition to occur.
- **Condition** specifies a boolean expression that needs to be evaluated to true for the transition to take place.
- **Condition action** specifies the action to be immediately executed when the condition evaluates to true.
- **Transition action** specifies the action to be executed when the transition destination has been determined to be valid provided the condition is true, if specified.

There can be different types of transition in Stateflow like inner transitions, transitions between substates etc. For the current project, we consider only flat Stateflows without junctions, history junctions. Hence we do not discuss about them here. Refer [8] for details about Stateflow.

2.3 Solvers

The Simulink models are simulated by *solvers* which computes the states of the system at successive time steps over a specified time span. A user can specify the simulation type of a model as ***fixed-step*** simulation where the states of the system is calculated at fixed intervals. This size of the interval is called the *step size*. Smaller the step size, more the accuracy of simulation and more is the time required to simulate. On the other hand, one can also specify ***variable step*** simulation for a system in which case the solver reduces the time step where the system change is rapid and increases the time step when the change is slow. This approach is useful for models with rapidly changing or piecewise continuous states and maintains a specified level of accuracy.

The solvers can be broadly categorised into *continuous solvers* and *discrete solvers*. Continuous solvers compute the state of a system in the current time step using numerical integration from the system's state in the previous time steps and the state derivatives. Continuous solvers rely on the models blocks to compute the values of the models discrete states at each time step. Discrete solvers exist primarily to solve purely discrete models. They compute the next simulation time step for a model and nothing else. They do

not compute continuous states and they rely on the models blocks to update the models discrete states.

No single method of solving a model suffices for all systems. Accordingly, Simulink provides a set of solvers, each embody a particular approach to solving a model. A user can choose a solver for a model that is best suited.

2.4 About Simulink Model File

A MATLAB Simulink model is saved as a model file with .mdl extension. This is a text file which stores the information about the graphically created model in the Simulink GUI using a kind of markup language created by Mathworks. This file is the way by which the model specific information can be accessed. The mdl file for a model describes which blocks are used in the model, the connections between blocks and the parameters along with graphical display information. If the model contains a Stateflow block, then Stateflow block information is also put in the mdl file. Refer appendix A to know about Simulink model file format.

While understanding the mdl file format and going through the mdl files for a number of models, we made the following observations which are worth mentioning in the translation context.

Observation 1: A few model files contain some encrypted data within MatData block. It is not clear as to what this encrypted data is all about. Naturally a question arises, is it an important piece of data about a model? Is it safe to ignore it while translating the model to an intermediate format?

The answer is yet not clear.

Observation 2: Several data files like lookup tables are part of MATLAB workspace and are loaded by preloaded .m files from outside the model. These data needs to be incorporated in the translation process, either by manually running some scripts or otherwise.

Chapter 3

Deciding the Intermediate format

In this chapter, we specify the requirements of the intermediate format, consider available options and decide on a format based on hybrid automata.

3.1 Requirements of the Intermediate Format

The chosen intermediate format should be such that its syntax should be simple and it should be semantically rich enough to preserve the semantics of the SL/SF model. It should be further based on the following observations:

- Most of the real world SL/SF models are Hybrid systems, i.e, they exhibit continuous as well as discrete behavior. The intermediate format should be such that to be able to represent these hybrid systems formally.
- Most of the verification and simulation tools like HyTech, Charon, Checkmate, Hysdel etc are tools for representing and analyzing hybrid systems.

The above two observations lead us to choose an intermediate format which can model hybrid systems. Hybrid automata is then a natural choice.

3.2 Hybrid Automata

Hybrid automata were proposed as a formal model for Hybrid Systems in [1]. A Hybrid system is a dynamical system with both discrete and continu-

ous components. For example, Thermostat is an example of Hybrid system where the change of Temperature is continuous and the controller behavior is discrete. For basic definitions and notation concerning hybrid automata, we follow [1].

Definition 3.1 A *hybrid automaton* consists of the following components.

Variables. A finite set $X = \{x_1, \ldots, x_n\}$ of real numbered variables. The number n is called the dimension of H. We write $\dot{X}$ for the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ of dotted variables (which represent first derivatives during continuous change), and we write X' for the set $\{x', \ldots, x'_n\}$ of primed variables (which represent values at the conclusion of discrete change).

Control graph. A finite directed multigraph (V, E). The vertices in V are called *control modes*. The edges in E are called *control switches*.

Initial,invariant,flow conditions. Three vertex labeling functions *init*, *inv*, and *flow* that assign to each control mode $v \in V$ three predicates. Each initial condition *init(v)* is a predicate whose variables are from X. Each invariant condition *inv(v)* is a predicate whose free variables are from X. Each flow condition *flow(v)* is a predicate whose free variables are from $X \cup \dot{X}$.

Jump conditions. An edge labeling function jump that assigns to each control switch $e \in E$ a predicate. Each jump condition *jump(e)* is a predicate whose free variables are from $X \cup X'$.

Events. A finite set Σ of events, and an edge labeling function *event*: $E \to \Sigma$ that assigns to each control switch an event.□

We explain the execution semantics informally with the help of a small example hybrid system Thermostat as described in [1]. Refer [1] for formal semantics of hybrid automata.

The hybrid automata of Fig 3.1 models a thermostat. The variable x represents the temperature. In Control mode *Off*, the heater is off, and the temperature falls according to the flow condition $\dot{x} = -0.1x$. The invariant $x \geq 18$ in control mode *Off* specifies that the heater will remain off so far as the temperature remains above or equal to 18. In control mode *On*, the heater is on, and the temperature rises according to the flow condition $\dot{x} = 5 - 0.1x$. The invariant $x \leq 22$ specifies that the heater will remain on so far as the temperature remains below or equal to 22. The transition from *Off* to *On* occurs on the condition $x > 21$ (temperature goes beyond 21).

The transition from *On* to *Off* occurs on the condition $x < 19$ (temperature falls below 19). Initially, the heater is off and the temperature is 20 degrees. □

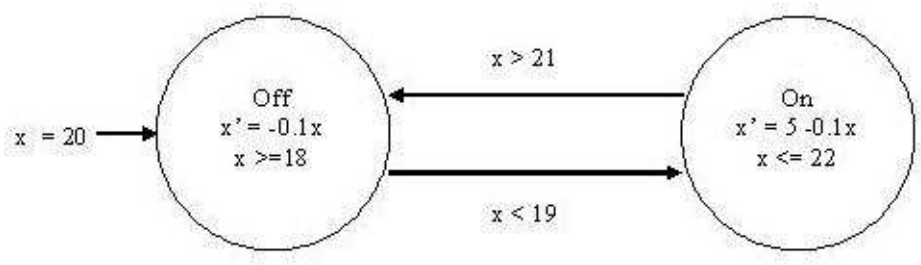


Figure 3.1: Thermostat Automaton

3.3 Network of Hybrid Automata

The real world systems are extremely large in size and coming up with a single hybrid automata for such models is a difficult and cumbersome task. The MoBIES team [3] has done considerable research on the same problem of defining an interchange format for hybrid systems and the result of it is an interchange format called Hybrid Systems Interchange Format(HSIF). In HSIF, hybrid systems are represented as a network of communicating hybrid automata. This approach is scalable since the huge models can be broken down into a number of smaller and simple communicating hybrid automata. For SL/SF models also, this approach is natural and close enough to the Simulink model structure of having a number of connected blocks.

According to HSIF description, a network has a set of global variables visible to all the hybrid automata in the network. Additionally, individual hybrid automaton has a set of local variables whose scope is limited only to the particular hybrid automaton. Global variables are further partitioned into signals and shared variables. Signals are of two types: input and output signals. They model the continuous variables of the network. Shared variables model the discrete variables. Shared variables can be updated only during a discrete step of a hybrid automaton by means of transition actions. Each output signal can be updated by exactly one hybrid automaton and

is called the output variable of that hybrid automaton. An input signal is called an input signal of hybrid automaton if the flow and algebraic equations of the automaton use that signal, but never update it.

Hybrid automata communicate with each other by means of signals and shared variables. If a hybrid automata A's output signal is an input signal to hybrid automata B, then HA B is said to be data dependent on HA A. HSIF requires the graph of data dependencies to be acyclic. Networks having cyclic dependency graph is rejected by the current HSIF semantics. Refer [3] for formal description of hybrid automata network.

We end this section with the conclusion that the chosen intermediate format's syntax should follow network of hybrid automata concept as defined in HSIF.

3.4 HyVisual

A natural question that may arise in the reader's mind at this point is why not use HSIF as the intermediate format. The answer to this question lies in the point that we want to verify the correctness of our translation. We have the following two possible approaches to the translation validation problem:

1. Restrict to a subset of SL/SF for which formal semantics is defined and prove that the translated model is equivalent to the original model according to a common formal semantics.

2. Prove (partially) by conformance testing that the translated model is equivalent to the original model.

Because of the application requirements, we do not want to restrict the SL/SF inputs. For the second approach, we need a tool that can simulate the translated model, however there is no simulator currently which is able to simulate HSIF models. Only a hybrid systems tool called HyVisual (part of Ptolemy [13] tool set) has the HSIF import facility but the import mechanism turned out to be buggy and the developers of HyVisual accepted the fact[1].

In HyVisual, one can represent hybrid automata as a ModalModel block. A ModalModel Block is a top level block which has further refinement as states and transitions with guards and actions. States can have further refinements which define the continuous behavior at that state. ModalModel block has input and output ports to communicate with other blocks. We can represent a network of hybrid automata in HyVisual by defining a ModalModel

[1] I found the bug and communicated it to the HyVisual developers group.

block for each hybrid automata and connect them by means of their input/output ports. Hence, We Choose **Network of Hybrid Automata representation in HyVisual** as the intermediate format. Refer appendix C to know more about HyVisual. The following points further justifies the decision.

- HyVisual models can be simulated. Hence, we can validate the translation by means conformance testing.
- HyVisual models are represented in an XML based language called Modeling Markup Language(MoML). Hence its syntax is simple and the model information can be easily retrieved by the use of a number of existing XML parsers.

Chapter 4

Implementation

We discuss the details of the translator implementation in this chapter. Figure 4.1 shows the high level flow diagram of the translator implementation. The Simulink model file is given as input to a parser which builds a model object having all the Simulink model details. This model object is used by the MoML code generator that outputs a HyVisual model represented in MoML. We discuss about the parser implementation in section 4.1 and about the MoML code generator in section 4.2.

4.1 MATLAB Model File Parser Implementation

The MATLAB SL/SF models are stored as mdl files, which is a kind of textual description of the Model, describing the blocks and the connections along with many other information needed for simulation and graphical display of the model in MATLAB. Any automated endeavor which involves MATLAB SL/SF models (Like Model Transformers, Model Analyzers etc) have to acquire the required information about the SL/SF models, and hence arises the need to parse the mdl file.

We developed a generic parser for MATLAB model files that provides an API to fetch the necessary information about a model.

4.1.1 Why Java Implementation?

The Parser is implemented in Java using Jdk 1.6 and Jflex 1.4.1. Java is chosen as the implementation language to make the parser platform independent and to make the development task easy by using the rich set of APIs the Java library provides. Also the concept of "lexical states" in Jflex

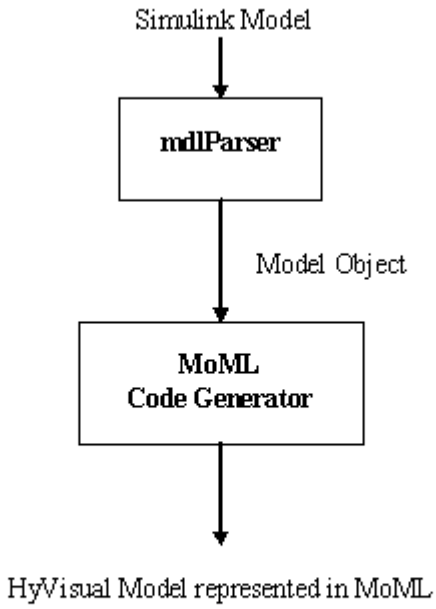


Figure 4.1: A High Level Flow Diagram of the Translator Implementation.

is very useful and gives the power of deterministic finite state automaton. The C scanner generator Flex does not have this functionality. The lexical rules in the specification file can be assigned to states. When the lexer is in a state, only the lexical rules in that state are matched. This functionality in Jflex comes very handy in the parser implementation as we shall see in the next section.

4.1.2 How the Parser Works?

The parser works using Jflex 1.4.1 as the backbone. As the lexical rules specified in the Jflex input specification file are matched, the corresponding actions builds up a Model object on the fly. When full scanning of the model file is completed, we end up in having a hierarchical model object that has all

the information of the model, preserving the hierarchy by means of parent-child relation among objects.

We first describe the structure of the Model object and then show how the model object is created.

Model Object Structure

Refer Fig 4.2 to see the model object Data Model. The model object stores the list of connections, default Block(s), the actual model block(s) and the Stateflow description of the model. The defaultBlocks field within the model object contains a list of block(s) with default parameters-values scanned from BlockParameterDefaults section in the mdl file [refer Appendix A to see the model file format], whereas the blocks field contains a list of blocks with the default parameters-values as well as other parameters-values present in the model block instance scanned from within System section. The default block parameters are updated in the block instance if the parameter value is set to some different value in the actual block instance. Hence, the blocks list in the Model Object has the model blocks with all the parameter information required, the defaults, the overridden and the block specific parameters. The sub_models field in the model object is a list of references to model objects again, that represent essentially the sub-systems of the current model. This model object reference(child) within a model object(parent) notion captures the hierarchy in a model.

The sf_obj field in the model object is an object of StateFlowBlock Class. This object encapsulates all the Stateflow related information of the model and provides the APIs getTransitions(), getStates() etc to access them. If a model contains no Stateflow block, then sf_obj is set to null. sf_obj contains a list of transition and a list of states as present in the Stateflow description. Since the information about all the Stateflow blocks in a model is described as a single list of states and transitions in a mdl file, we define only a single Stateflow object for a model having all those information. To which Stateflow block a state or a transition belongs to, can be obtained from the getChart() API which returns the chart id of the state, transition.

To encapsulate Transition and State information, we define a Transition class and a State class. The State object encapsulates the state information in a Stateflow. It has the state id, the entry actions, the during actions, the exit actions and the on event actions associated the state. It has its own state label tokenizer method to instantiate its member fields from the state label string. The APIs it provides are getId(), getChart(), getEntryActions(), getDuringActions(), getExitActions() and getOnEventActions().

A Transition object has the source state id and destination state id of

the transition, the transition event, condition, condition actions and the transition actions. A transition object also has a transition label tokenizer method which extracts out the event, condition etc from the transition label and instantiates the member fields. It provides the APIs getDestinationId(), getSourceId(), getEvent(), getCondition(), getConditionActions(), getTransitionActions() etc.

The connections list within the model object stores the information about the model links. Each Connection object in the list represent a single line between some source block to some destination block. A connection specific information can be retrieved from the getSrcBlock(), getDstBlock(), getSrcPort() and getDstPort() APIs.

Each Block Object in the Blocks list contains a list of parameters. These parameters are instance of Pair class. A Pair object has 2 fields, parameterName and parameterValue. A block specific information can be retrieved using the getParameters(), getBlockType(), getBlockName() etc APIs. A pair specific information can be retrieved using getParameterName() and getParameterValue() APIs.

The parser is accompanied by an extensive API documentation using Javadoc[14].

Building the Model Object

The Model object is built dynamically as the model file is scanned from top to bottom. The lexical states and the corresponding actions associated with the states lexical rule matches build up the model object. The diagram in Fig 4.3 describes the model object build flow. [name] represents lexical state with name 'name' and the transition labels represent the condition when the transition should trigger. Strings in the transition label that do not evaluate to boolean values are actually tokens returned by the scanner, e.g, the string "Block {" represents a token in the diagram and says that the transition should trigger if that token is returned. Some transitions also mention actions that are taken as part of the transition. Strings within { } just below a state represent the task done when the control is in particular state. The diagram is simplified so as not to make it too much congested, although the main idea of model object building is conveyed.

Fig 4.4 shows the Stateflow object build state machine separately.

4.1.3 How to Compile and Run from the Source?

All the class implementations in the parser are present inside **ModelCreator Package**. The lexer specification file comes separately. Before compilation, the CLASSPATH environment variable should be set to include the parent directory of the location where ModelCreator Package is saved. Then follow the commands in the command prompt mentioned below. The mentioned commands should be run from the directory where the jflex lexer specification file is saved.

- jflex ⟨lexer specification file⟩
- javac Lexer.java
- java Lexer ⟨modelfilepath⟩

System is required to have Jflex 1.4.1 or above and Jdk 1.6 or above installed. The Parser is compatible to MATLAB version 6 and 7.

4.2 MoML Code Generator

The Model Object generated by the parser is used by a Java class called **GenMoMLCode** which gets the model specific information through the provided APIs discussed in the section above and generates a HyVisual model represented in MoML. The generated HyVisual model emulates the Simulink model as a network of hybrid automata.

4.2.1 Translation Algorithm

Algorithm

Step 1. For each Simulink block, define a hybrid automaton as a HyVisual ModalModel Block such that the following conditions hold:

- If the Simulink block has n input ports, then define n input ports with names $< x_1, \ldots, x_n >$ of the ModalModel block.
- If the Simulink block has n output ports, then define n output ports with names $< y_1, \ldots, y_n >$ in the ModalModel block.
- If the Simulink block has parameters $< p_1, \ldots, p_n >$, then define parameters $< p_1, \ldots, p_n >$ in the ModalModel. Instantiate the parameters to the values set in the Simulink block.

- The states,transitions and the state refinements of the ModalModel block should be defined such that its execution emulates the behavior of the Simulink block.

Step 2. For each connection in the Simulink model between block A's output port i to Block B's input port say j, connect the ModalModel of Block A's output port y_i to ModalModel of Block B's input port x_i by means of a relation named arbitrarily.

Notice that the algorithm emphasizes on the name of the IO ports. This is only to stick onto a fixed naming scheme for the ease of implementation, understandability and cleanliness of the generated code.

The fourth condition in step 1 of the algorithm is incomplete and we need to properly define the HA for each Simulink block whose execution exactly emulates the behavior of the Simulink block. In the next section, we define the HA's for some simple Simulink blocks.

4.2.2 Hybrid Automata Emulation of Simulink Blocks

Refer Fig 4.5 and 4.6 to see the hybrid automaton equivalent of some Simulink blocks. The description of the hybrid automata for each block in the figure is as follows:

- **GAIN**. The Gain block scales the input signal to k times, where k is a parameter defined by the user. The equivalent hybrid automata has an algebraic equation saying $y = k.x$, thus defining the output variable behavior appropriately.

- **INTEGRATOR.** The flow equation $\dot{y} = x$ captures the integrator behavior in the integrator HA.

- **ABS.** The initial state takes a transition to a state with algebraic equation $y = x$ if the input is positive, and similarly takes a transition to the other state with algebraic equation $y = -x$ if the input is negative. The two non-initial states then take transitions among themselves according to the input value and hence always output the absolute value of the input.

- **AND.** The transitions in the AND hybrid automata sets the output variable value appropriately depending on the inputs by means of transition actions. For example, one of the four self transition set the output to true when both the inputs are true.

- **PRODUCT.** The algebraic equation $y = x1 * x2$ in the state captures the behavior of the product block.
- **DIVIDE.** The algebraic equation $y = x1/x2$ in the state captures the behavior of the divide block.
- **SUM.** The algebraic equation $y = x1 + x2$ in the state captures the behavior of the sum block.
- **RAMP.** The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate. The initial state outputs 0 upto a user specified time k, where k is a local parameter of the hybrid automata. x is a local variable of the hybrid automata used to model time in the initial state with the flow equation $\dot{x} = 1$. At time instant k, the transition guard $x = k$ becomes true and control moves to the next state with the flow equation $\dot{y} = SL$, where SL is a parameter, y is the output variable. The flow equation makes y to change with a constant rate SL and hence achieves the Ramp behavior.

Fig 4.7 shows a simple Simulink model and the corresponding equivalent hybrid automata network of the model. The output variable x of the first hybrid automaton is the input to the second and the output variable y of the second hybrid automaton is the input to the third. The data dependency graph is $HA_1 \rightarrow HA_2$. Notice that HA_3 is not data dependent on HA_2 because inputs used in differential equations are not dependencies.

4.2.3 MoML Blocks Library

Notice that the step 1 of the algorithm can be statically implemented expect the initialisation of the parameters of the ModalModel block. Hence, we define the hybrid automata equivalents of the Simulink blocks as MoML code and store them beforehand in a MoML Blocks library. We initialise the parameters with some default values which are then dynamically replaced by the actual block parameter values as defined in the Simulink block by the GenMoMLCode class.

To generate MoML code for the Simulink blocks, we build on the existing HSIF import facility in HyVisual. The hybrid automata description of a Simulink block is encoded in HSIF and input to the HSIF import facility of HyVisual to get the ModalModel Block equivalent of the HSIF description. Remember that we said earlier, the HSIF import mechanism has bugs. To come up with this, we check the HyVisual translated Model and correct the bugs manually. The corrected HyVisual Model is then stored as MoML code

in the library. During a translation, we simply refer to the MoML code for a particular Simulink block in the library and then instantiate the parameters with the actual parameter values.

The reason why we encode the hybrid automata description in HSIF is because the flow, algebraic equations, and invariants of a state of a hybrid automata can be directly encoded in HSIF, but in HyVisual, they have to be encoded by means of continuous time blocks model in the state refinement which is not easy. One more reason is our observation that, the HSIF import mechanism includes **expression blocks** of HyVisual to encode the flow, algebraic equation and the invariant predicate of a state in the state refinement of the equivalent HyVisual model. Hence, the equations and the invariant predicate can be directly retrieved from the MoML code, which is going to be helpful while using the intermediate format to translate to other models.

Shown below is the hybrid automata representation of Integrator block encoded in HSIF. Refer Appendix B to see HSIF Abstract Syntax.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE DNHA SYSTEM "HSIF.dtd">

<DNHA name="INTEGRATION_BLOCK" parameterAccuracy="0.001000">
<HybridAutomaton name="INTEGRATION">
<RealVariable _id="idv1"  ref="idr2" name="x1" kind="Input"/>
<RealParameter name="InitialValue" value="0"/>
<RealVariable _id="idv2" initialMinValue="InitialValue"
initialMaxValue="IntialValue" ref="idr3" name="y1" kind="Controlled"/>
<Location _id="ids1" dst="" src="" name="S1" initial="true">
   <DiffEquation>
    <AExpr>
     <MExpr>
      <VarRef _id="idr2" var="idv1" unOp="NOP"/>
     </MExpr>
    </AExpr>
    <VarRef _id="idr3" var="idv2" unOp="NOP"/>
   </DiffEquation>
</Location>
</HybridAutomaton>
</DNHA>
```

The MoML code snippet for Integrator block in the MoML library is shown below. Refer Appendix D for MoML DTD and [5] for details about MoML. The XML version etc header code is not put in the block's library code as they are to be included only once in the generated file and is generated by the code generator's addHeader() method.

```
<entity name="INTEGRATION" class="ptolemy.domains.fsm.modal.ModalModel">
    <property name="_tableauFactory"
    class="ptolemy.vergil.fsm.modal.ModalTableauFactory">
    </property>
    <property name="InitialValue"
    class="ptolemy.data.expr.Parameter" value="0">
    </property>
    <property name="_location" class="ptolemy.kernel.util.Location"
    value="[10.0, -50.0]">
    </property>
    <port name="y1" class="ptolemy.domains.fsm.modal.ModalPort">
        <property name="output"/>
        <property name="minValue" class="ptolemy.data.expr.Parameter"
        value="">
        </property>
        <property name="maxValue" class="ptolemy.data.expr.Parameter"
        value="">
        </property>
        <property name="initialMinValue"
        class="ptolemy.data.expr.Parameter" value="InitialValue">
        </property>
        <property name="initialMaxValue"
        class="ptolemy.data.expr.Parameter" value="IntialValue">
        </property>
        <property name="_type" class="ptolemy.actor.TypeAttribute"
        value="double">
        </property>
    </port>
    .
```

```
    .
    .
    .
    .
    </entity>
    <relation name="y1Relation" class="ptolemy.actor.TypedIORelation">
    </relation>
    <relation name="x1Relation" class="ptolemy.actor.TypedIORelation">
    </relation>
    <link port="y1" relation="y1Relation"/>
    <link port="x1" relation="x1Relation"/>
    <link port="_Controller.y1" relation="y1Relation"/>
    <link port="_Controller.x1" relation="x1Relation"/>
    <link port="S1.y1" relation="y1Relation"/>
    <link port="S1.x1" relation="x1Relation"/>
</entity>
```

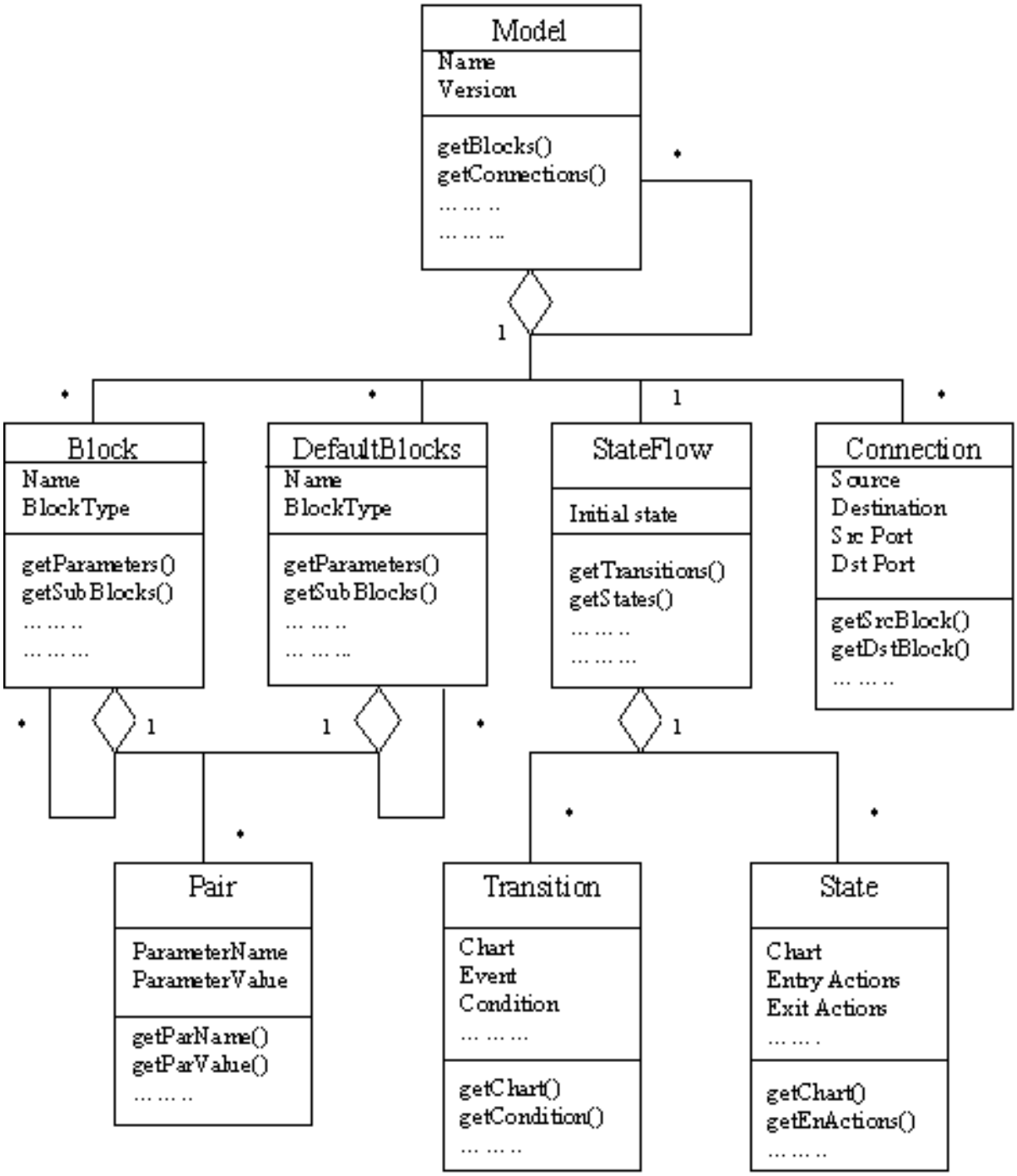


Figure 4.2: Model Object Data Model

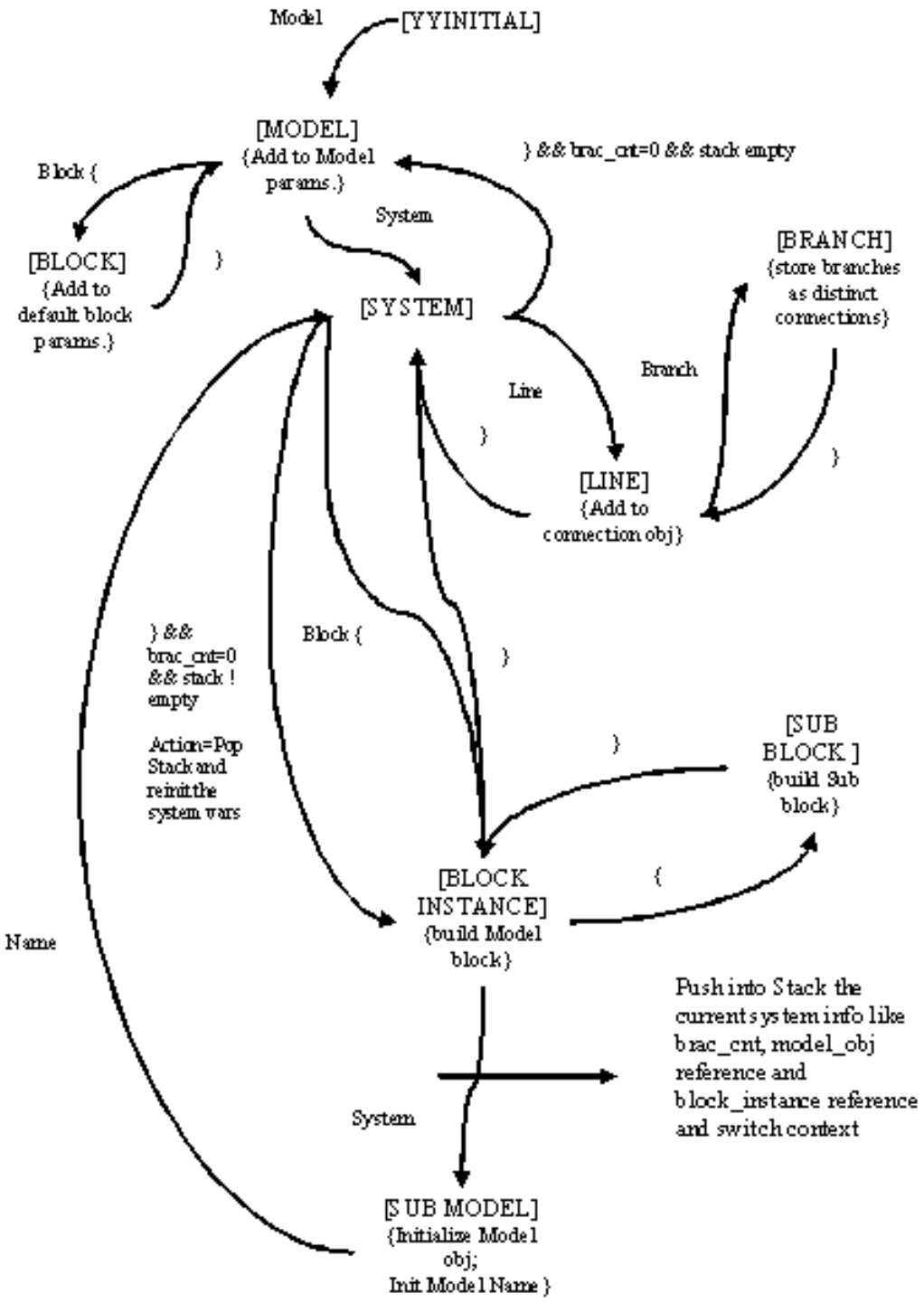


Figure 4.3: Model Object Build State Machine

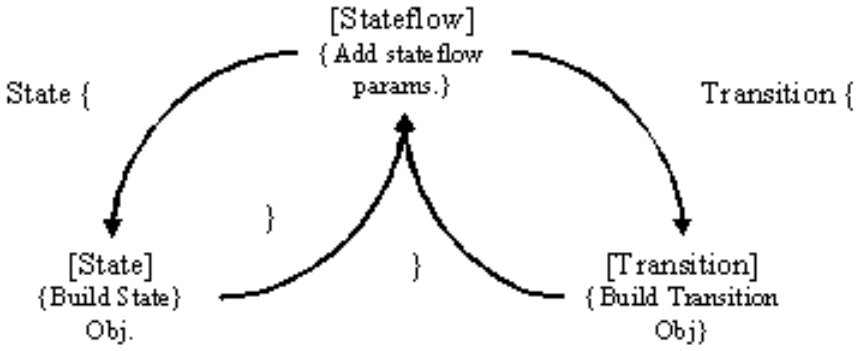


Figure 4.4: Stateflow Object Build State Machine

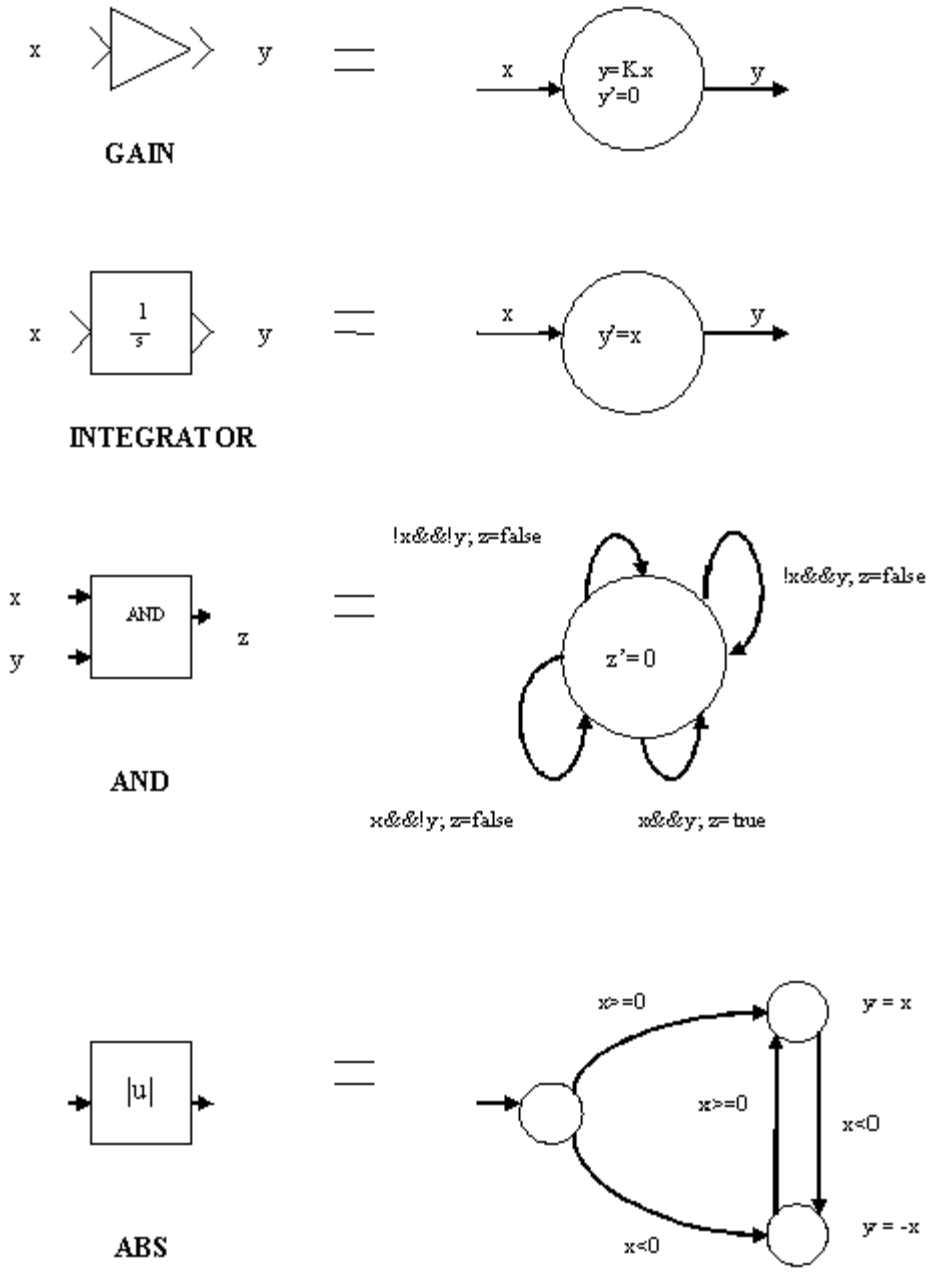


Figure 4.5: Hybrid Automaton equivalent of some Simulink Blocks

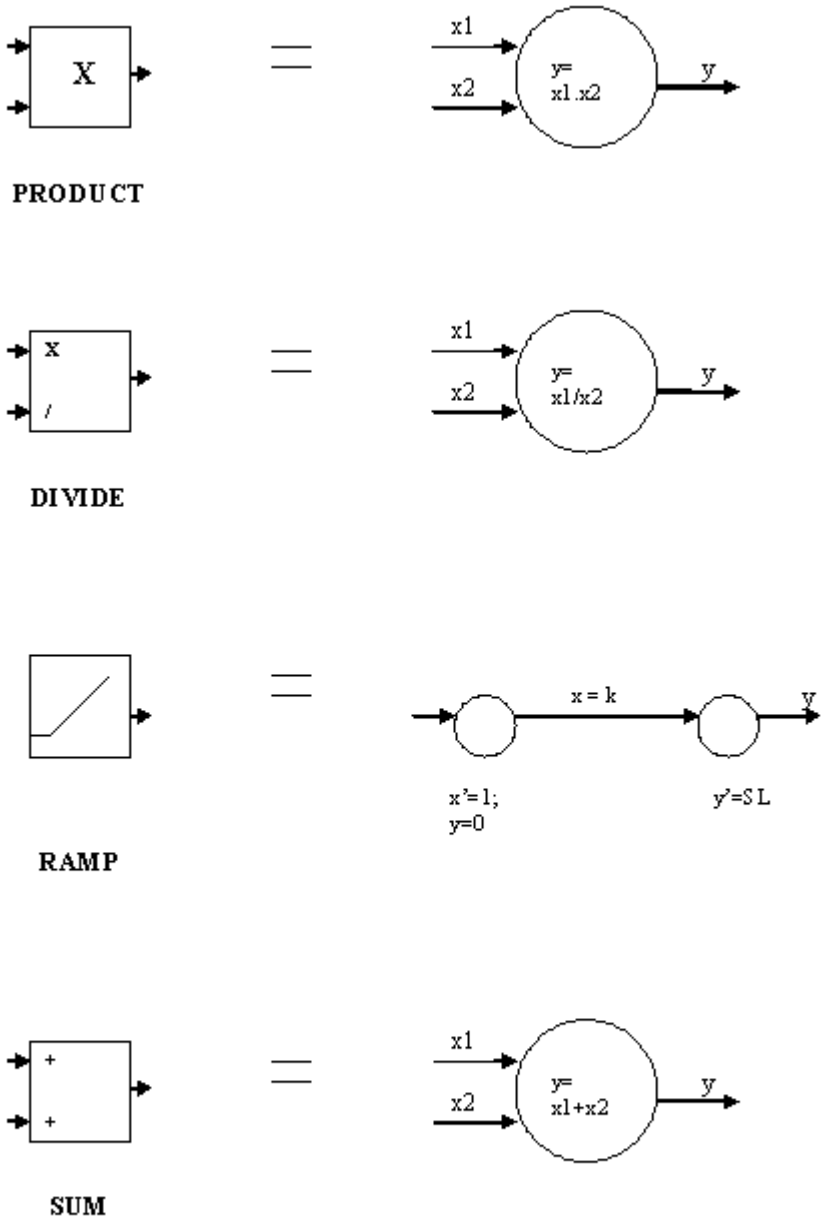
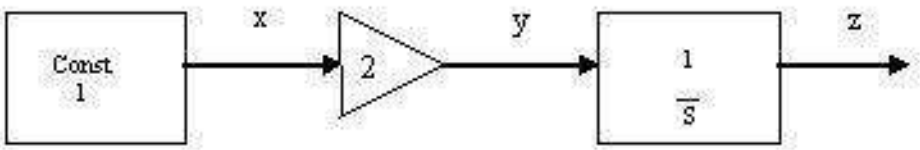
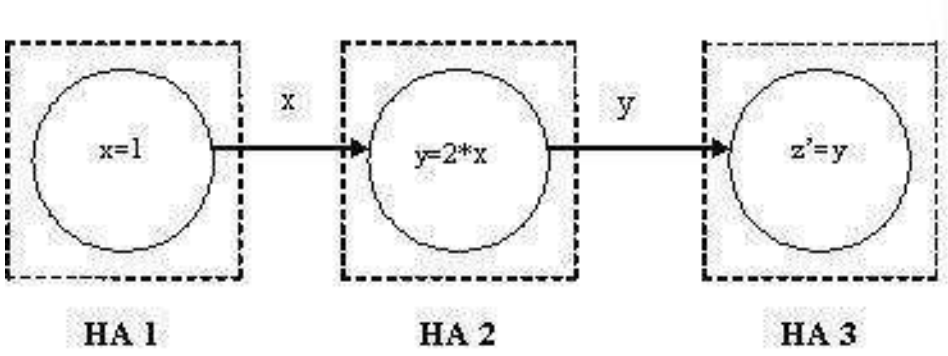


Figure 4.6: Hybrid Automaton equivalent of some Simulink Blocks



Simulink Model



Hybrid Automata Network

Figure 4.7: Hybrid Automata Network Equivalent of a Simulink Model

Chapter 5

Discussions and Future Work

5.1 Discussions

In this chapter we discuss an alternative approach to the problem, the experience we gained from that and conclude with some future work in this context.

5.1.1 The GReAT Attempt

GReAT [12] is a model to model transformation tool developed as a part of the MoBIES project. Simulink model to HSIF(and not HyVisual) model transformation using GReAT was the first approach we followed. GReAT works on the principle of Graph Grammars and Graph Transformations. There has been work done on translating a subset of SL/SF models to HSIF using GReAT[4].
To specify a model transformation in GReAT, we need 3 things:

1. The source meta model
2. The Target meta model
3. Transformation rules.

GReAT attaches itself to GME (Generic Modeling Environment) tool. The source, target meta models and the transformation rules are defined in GME graphically. The target meta model, i.e, the HSIF meta model is available from the HSIF documentation. The Simulink meta model needs to be constructed and defining the transformation rules forms the translation algorithm. Transformation rules in GReAT are like specifying patterns in the source meta model and then mapping the pattern to some pattern

in the target meta model. While doing a model transformation, GReAT searches for the source patterns in the input model and replaces them with the corresponding target patterns in the output model.

To gain a firsthand experience in GReAT, I emulated the House2Order transformation described in the GReAT tutorial that comes with the installation. I followed the steps exactly as defined in the tutorial, but the resulting transformation ended up giving errors. The same transformation was working on some other system. The problem was traced to version mismatch of various softwares required to run GReAT like GME, VisualStudio and UDM (Universal Data Model). The system where the transformation was found to be working had older versions of the required softwares and we didn't have the access to some of them due to license issues. As a result, we decided to change our approach and departed from GReAT.

5.1.2 Generic Use of the Parser

The MATLAB Model file parser implemented in my project has generic use other than my project. Any tool to be implemented that needs to retrieve SL/SF model information can use the parser. It is to be noted that the current version of the parser is unable to preserve the hierarchy information of Stateflow blocks in the built Model Object.

5.1.3 Translation Validation

Due to the lack of formal semantics of Simulink, we validate the translator by conformance testing. Test results showed that we have successfully achieved the goal of having a prototype implementation of the translator, although more exhaustive testings are being planned. Shown in Fig 5.3 - 5.4 are the snapshots of the MATLAB Simulink output and HyVisual output for an example Simulink model and the translated HyVisual model for a particular set of input respectively.

5.2 Future Work

The current translator implementation is limited in its scope. We want to extend it in the following directions:

1. Extending the scope of the translator to blocks like Sinewave generator, Saturation block, Unit delay etc and Lookup Tables.

2. Encoding hierarchy in Dynamic Network of Hybrid Automata.

3. Extending DNHA to take care of feedback loops.

4. A formal translation validation. □

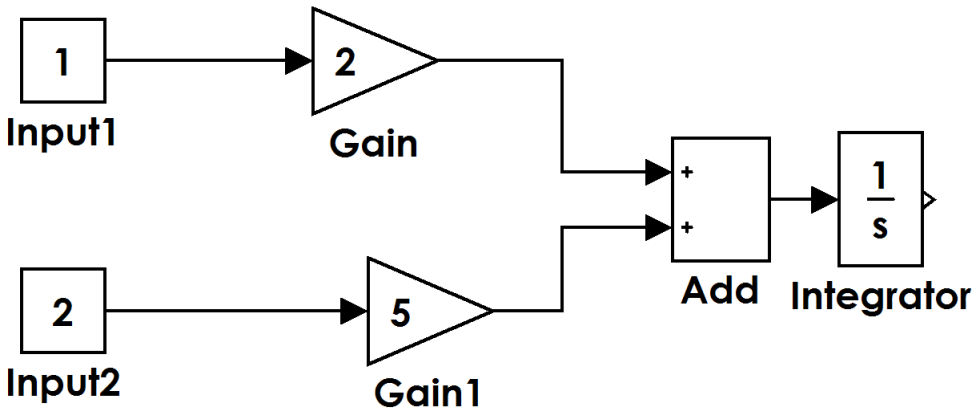


Figure 5.1: An Example Simulink Model.

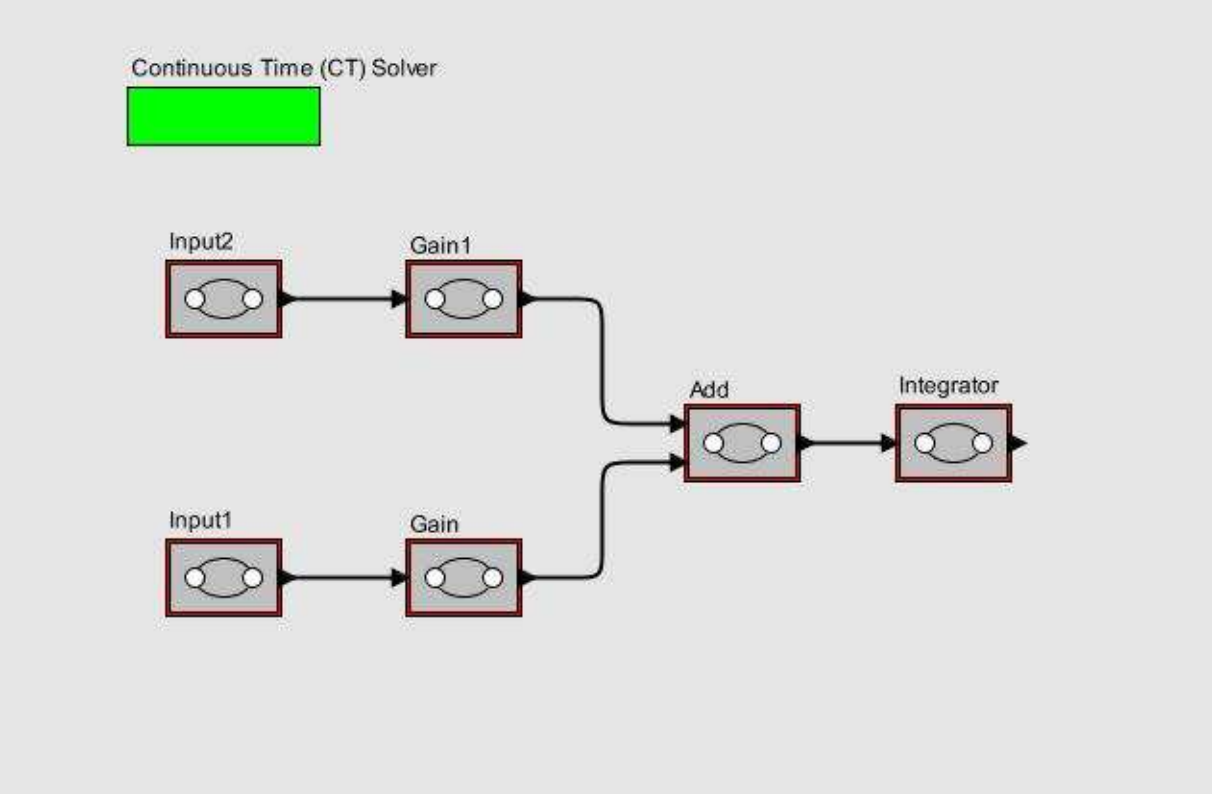


Figure 5.2: The Translated Model in HyVisual.

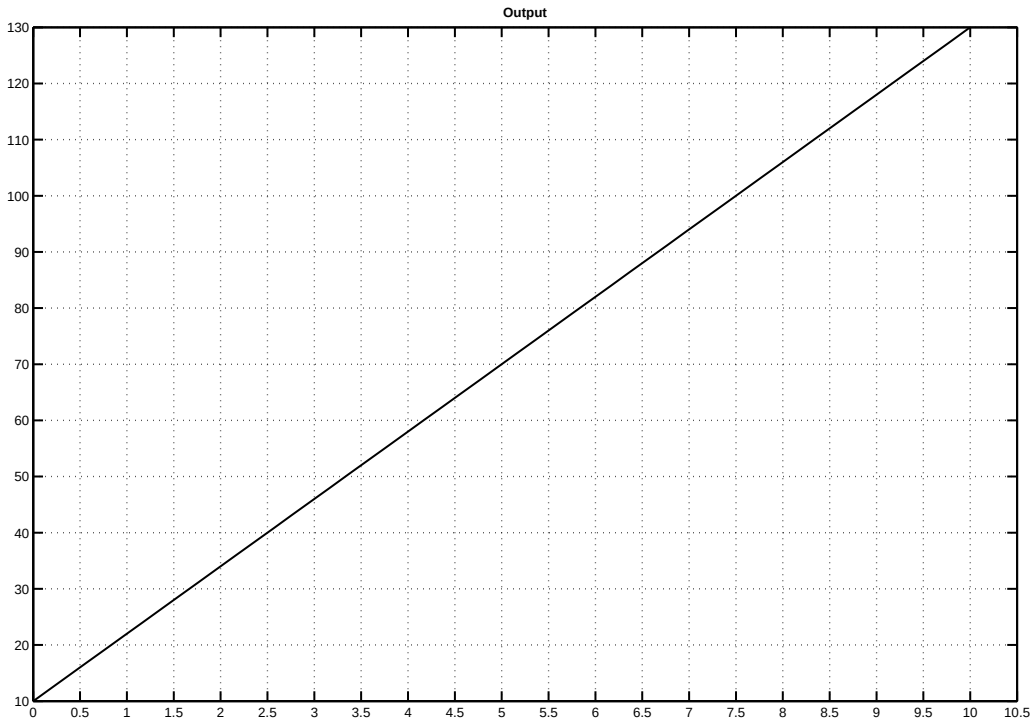


Figure 5.3: Simulation Output of the Simulink Model.

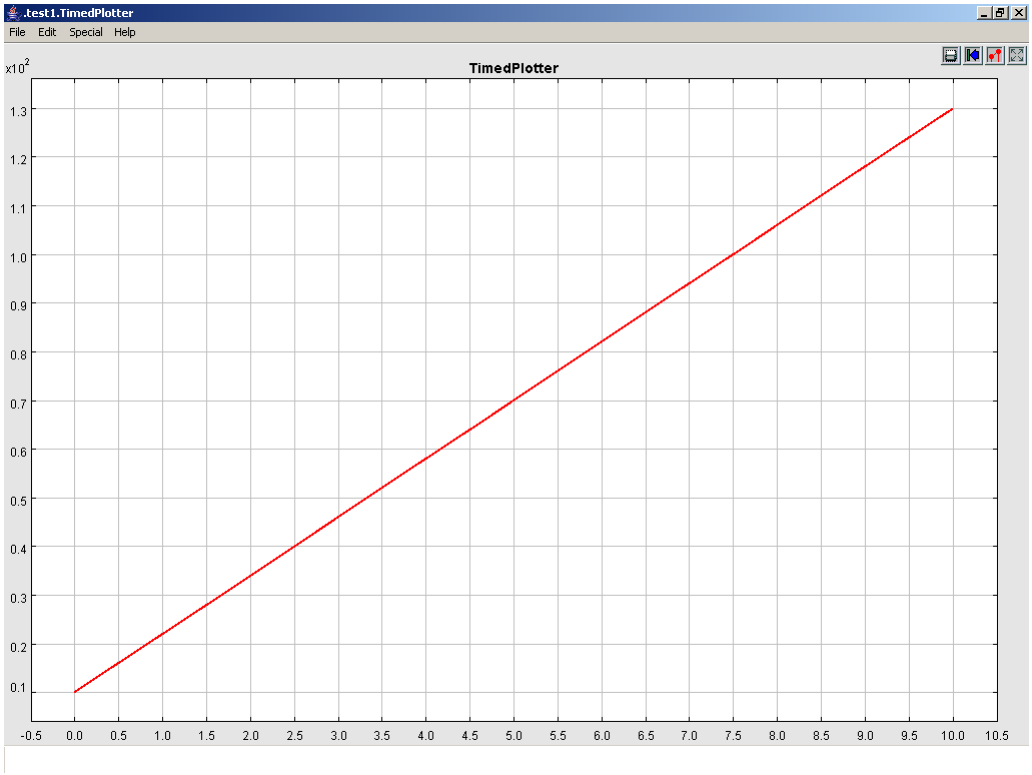


Figure 5.4: Simulation Output of the Translated HyVisual Model.

Appendix A

Simulink Model File Format

This chapter describes the Simulink model file format as defined in MATLAB Simulink documentation.

A model file is a structured ASCII file that contains keywords and parameter-value pairs that describe the model. The file describes model components in hierarchical order.

The structure of the model file is as follows.

```
Model {
  <Model Parameter Name> <Model Parameter Value>
  ...
  Array {
    Simulink.ConfigSet {
      $ObjectID <Object ID>
      <ConfigSet Parameter Name> <ConfigSet Parameter Value>
      ...
    }
  }
  Simulink.ConfigSet {
    $PropName "ActiveConfigurationSet"
    $ObjectID <Object ID>
  }
  BlockDefaults {
    <Block Parameter Name> <Block Parameter Value>
    ...
  }
  BlockParameterDefaults {
    Block {
      <Block Parameter Name> <Block Parameter Value>
```

```
      ...
    }
  }
  AnnotationDefaults {
    <Annotation Parameter Name> <Annotation Parameter Value>
    ...
  }
  LineDefaults {
    <Line Parameter Name> <Line Parameter Value>
    ...
  }
  System {
    <System Parameter Name> <System Parameter Value>
    ...
    Block {
      <Block Parameter Name> <Block Parameter Value>
      ...
    }
    Line {
      <Line Parameter Name> <Line Parameter Value>
      ...
      Branch {
        <Branch Parameter Name> <Branch Parameter Value>
        ...
      }
    }
    Annotation {
      <Annotation Parameter Name> <Annotation Parameter Value>
      ...
    }
  }
}
```

A.1 Short Description

- The Model section defines model parameters, configuration sets, and configuration references
- The Simulink.ConfigSet section identifies the active configuration set or configuration reference

- The BlockDefaults section contains default settings for parameters common to all blocks in the model.
- The BlockParameterDefaults section contains default settings for block-specific parameters.
- The AnnotationDefaults section contains default settings for annotations in the model.
- The LineDefaults section contains default settings for lines in the model.
- The System section contains parameters that describe each system (including the top-level system and each subsystem) in the model. Each System section contains block, line, and annotation descriptions.

  For more information regarding Simulink model file format, refer MATLAB Simulink documentation. [8]

Appendix B

HSIF Abstract Syntax

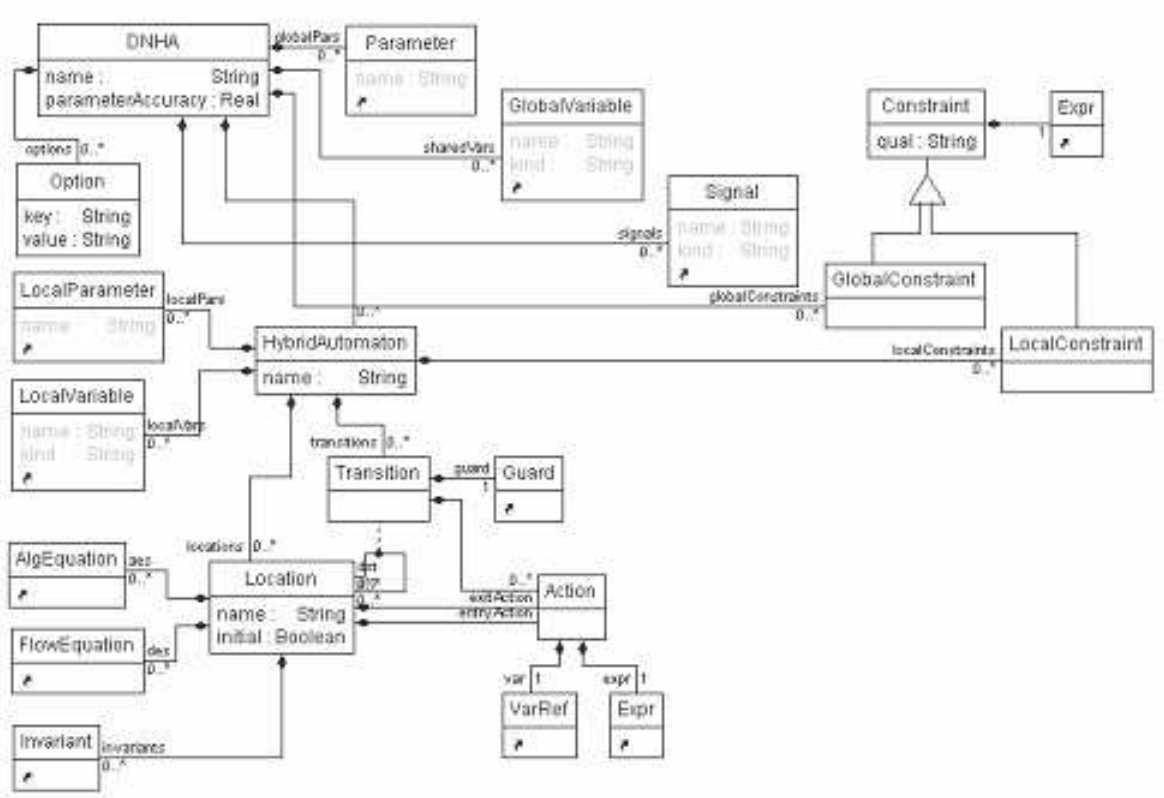


Figure B.1: Core HSIF data model

Appendix C

HyVisual

The Hybrid System Visual Modeler (HyVisual) is a block-diagram editor and simulator for continuous- time dynamical systems and hybrid systems. Hybrid systems mix continuous-time dynamics, discrete events, and discrete mode changes. This visual modeler supports construction of hierarchical hybrid systems. It uses a block-diagram representation of ordinary differential equations (ODEs) to define continuous dynamics, and allows mixing of continuous-time signals with events that are discrete in time. It uses a bubble-and-arc diagram representation of finite state machines to define discrete behavior driven by mode transitions.

HyVisual provides a sophisticated numerical solver that simulates the continuous-time dynamics, and effective use of the system requires at least a rudimentary understanding of the properties of the solver.

HyVisual is built on top of Ptolemy II, a framework supporting the construction of such domain-specific tools. See http://ptolemy.eecs.berkeley.edu for information about Ptolemy II. The latest version of HyVisual comes with Ptolemy-II 6.0.2 and can be downloaded from
http://ptolemy.eecs.berkeley.edu/ptolemyII.

Appendix D

MoML version 1 DTD

```
<!ELEMENT model (class | configure | director | doc | entity | import |
link | property | relation | rendition)*>
<!ATTLIST model name CDATA #REQUIRED
class CDATA #IMPLIED>

<!ELEMENT class (class | configure | director | doc | entity | import |
link | property | relation | rendition)*>
<!ATTLIST class name CDATA #REQUIRED
extends CDATA #IMPLIED>

<!ELEMENT configure (#PCDATA)>
<!ATTLIST configure source CDATA #IMPLIED>

<!ELEMENT director (configure | property)*>
<!ATTLIST director name CDATA "director"
class CDATA #REQUIRED>

<!ELEMENT doc (#PCDATA)>

<!ELEMENT entity (class | configure | director | doc | entity | import |
 link | port | link | port | property | relation | rendition)*>
<!ATTLIST entity name CDATA #REQUIRED
class CDATA #IMPLIED>

<!ELEMENT import EMPTY>
<!ATTLIST import source CDATA #REQUIRED
base CDATA #IMPLIED>
```

```
<!ELEMENT link EMPTY>
<!ATTLIST link port CDATA #REQUIRED
relation CDATA #REQUIRED
vertex CDATA #IMPLIED>

<!ELEMENT location EMPTY>
<!ATTLIST location value CDATA #REQUIRED>
<!ELEMENT port (configure | doc | property)*>
<!ATTLIST port class CDATA #IMPLIED
name CDATA #REQUIRED>

<!ELEMENT property (configure | doc | property)*>
<!ATTLIST property class CDATA #IMPLIED
name CDATA #REQUIRED
value CDATA #IMPLIED>

<!ELEMENT relation (property | vertex)*>
<!ATTLIST relation name CDATA #REQUIRED
class CDATA #IMPLIED>

<!ELEMENT rendition (configure | location | property)*>
<!ATTLIST rendition class CDATA #REQUIRED>
<!ELEMENT vertex (location | property)*>
<!ATTLIST vertex name CDATA #REQUIRED
pathTo CDATA #IMPLIED>
```

Bibliography

[1] Thomas A. Henzinger. The theory of hybrid automata. Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1996, pp. 278-292. An extended version appeared in Verification of Digital and Hybrid Systems (M.K. Inan, R.P. Kurshan, eds.), NATO ASI Series F: Computer and Systems Sciences, Vol. 170, Springer-Verlag, 2000, pp. 265-292.

[2] Edward A. Lee, Haiyang Zheng: Operational Semantics of Hybrid Systems. HSCC 2005: 25-53

[3] HSIF semantics (version 3,synchronous edition). Technical Report, University of Pennsylvania.(2002)

[4] Aditya Agrawal, Gyula Simon, Gabor Karsai: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. Electronic Notes in Theoretical Computer Science 109: 43-56 (2004)

[5] Edward A. Lee and Steve Neuendorffer. MoML - A Modeling Markup Language in XML - Version 0.4. Technical report, University of California at Berkeley, March, 2000.

[6] Ofer Strichman and Michael Ryabtsev. Translation validation: from Simulink to C. A progress Report, Technion, Haifa, Isreal.

[7] Stavros Tripakis, Christos Sofronis, Paul Caspi and Adrian Curic: Translating Discrete-Time Simulink to Lustre, ACM Transactions on Embedded Computing Systems, Vol.4, No.4, November 2005, pages 779-818.

[8] The Mathworks. MATLAB/Simulink. http://www.mathworks.com

[9] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, H. Zheng, HyVisual: A Hybrid System Visual Modeler, Technical Memorandum UCB/ERL M03/1, University of California, Berkeley, CA 94720, January 28, 2003.

[10] A. Tiwari , Formal Semantics and Analysis methods for Simulink Stateflow Models, Technical report, SRI International, 2002.

[11] M. Baleani, A. Ferrari, L. Mangeruca, A.L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, H.-J. Wolff: Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE05).

[12] http://www.isis.vanderbilt.edu/Projects/mobies/downloads.asp

[13] http://ptolemy.eecs.berkeley.edu/ptolemyII

[14] The MATLAB Model File Parser API documentation. Rajarshi Ray (rajarshi@cmi.ac.in). Chennai Mathematical Institute.