Translation validation: from Simulink to C

Research Thesis

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

Michael Ryabtsev

Submitted to the Senate of
the Technion - Israel Institute of Technology

IYYAR, 5769 HAIFA MAY, 2009

The Research Thesis Was Done Under The Supervision of
Dr. Ofer Strichman,
In the Department of
Computer Science.

The Generous Financial Help Of the Technion Is Gratefully Acknowledged.

Thanks are Due to my Family for their Patience and Support.

Thanks are Due to all those Great Persons Surrounding me, who Helped me to Carry Out this Work and not to Give Up.

Thanks to Dr. Ofer Strichman for his Guidance and Help.

Table of Contents

List of Figures

Abstract

Translation validation is a technique for formally establishing the semantic equivalence of a source and a target of a code generator. In this work we present a translation validation tool for the Real-Time Workshop code generator that receives as input Simulink models and generates optimized C code.

Chapter 1

Introduction

Model-based design has been established as an important paradigm in the development of embedded systems, which in many cases contain safety critical aspects. The main principle of the paradigm is to use models all along the development cycle, from design to implementation. Using models, rather than, say, building prototypes, is essential for keeping the development costs manageable. However, models alone are not enough. They need to be accompanied by powerful reasoning tools: simulation, verification, synthesis, code generation, and so on. Automation here is the key: high system complexity and short time-to-market make model reasoning a hopeless task, unless it is largely automatized.

There is an awareness in the industry of the fact that the application of formal verification/validation techniques may significantly reduce the risk of design errors in the development of such systems. However, if the validation efforts are focused on the model level, the question is how can we ensure that the quality and integrity achieved at the model level is safely transferred to the implementation level. Today's process of the development of such systems consists of automatic code generation from verified/validated models, followed by extensive unit and integration testing. Thus, there is a strong need for technology that could convincingly demonstrate to the certification authorities the correctness of the generated code. Although there are many examples of compiler verification in the literature, the formal verification of

industrial code generators is generally prohibitive due to their size. Another problem with compiler verification is that it tends to freeze their design, as each change of the code generator nullifies its previous correctness proof.

An alternative to compiler verification is *Translation Validation* [13, 11], which is a formal method for validating the semantic equivalence between the source and target of a code generator. The concept is to construct a fully automatic tool which establishes the correctness of the generated code individually for each run of the code generator. A translation validation tool receives as input the source and target programs, as well as a mapping between their input, output and state variables. Based on the semantics of the source and target languages, it then builds a verification condition, formally representing the source and the target code and their relations based on the mapping. It is valid if and only if the generated code faithfully preserves the semantics of the source code.

Hence, translation validation is applied separately to each translation, in contrast to the alternative of verifying the code generator itself, which is applied once. It has the advantage of being less sensitive to changes in the code generator, but on the other hand it can be sensitive to changes in the modeling language or the format in which it is represented. Proving code equivalence is simpler than compiler verification if the source and target code have a known and relatively simple structure. The second condition is typically met in the case of embedded code, since normally it does not use dynamic memory allocation and unbounded loops, and its structure is restricted by the reactive nature of control systems. Further, when the code generator is not open-source this is the only option.

In this thesis we apply Translation Validation to the Real-Time Workshop code generator, which is Mathwork's code generator from Simulink models. Simulink is an environment for simulation and Model-Based Design of dynamic and embedded systems in multiple domains. It lets the user design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. In automotive control Simulink became

a de-facto standard.

The code generated by Real-Time Workshop is optimized for speed and memory, and hence the translation is not straight-forward. As part of the thesis we developed the tvs (Translation validation from Simulink to C) tool, which applies translation validation to the RTW code generator.[1] tvs was developed with a focus on control systems. The largest example that we used it for comes from the automotive domain (a fuel control system), as will be described in Chap. 7.

Thesis Outline

We continue in the next section by describing Simulink modelling and execution semantics. In Chap. 3 we describe the verification condition generated by tvs. In Chap. 4 we explain how tvs generates automatically the verification condition. In Chap. 5 we describe the tvs structure. In Chap. 6 we discuss related work, and finally, in Chap. 7 we summarize our work and report on our experience with validating an industrial example.

[1] Simulink allows to combine state diagrams in controller models. called Stateflow, which tvs does not support.

Chapter 2

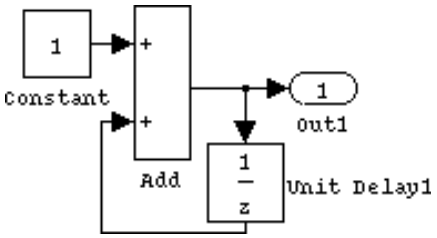
Simulink

2.1 Introduction

SIMULINK [9, 8], developed by *The MathWorks*, is a software package for model-based design of dynamic systems such as signal processing, control and communication applications. Models in SIMULINK can be thought of as executable specifications, because they can be simulated and, as mentioned earlier, they can be translated into a C program via the RTW code generator.

SIMULINK's graphical editor is used for modeling dynamic systems with a block diagram, consisting of blocks and arrows that represent signals between these blocks. A wide range of signal attributes can be specified, including signal name, data type (e.g., 16-bit or 32-bit integer), numeric type (real or complex), and dimensionality (e.g., one-dimensional or multidimensional array). Each block represents a set of equations, called block methods, which define a relationship between the blocks input signals, output signals and the state variables. Blocks are frequently parameterized with constants or arithmetical expressions over constants.

SIMULINK diagrams represent synchronous systems, and can therefore be translated naturally into an initialization function and a step function. The block diagram in the left of Figure 1, for example, represents a counter; the corresponding initialization and step functions that were generated by RTW appear to the right of the same




```
void example_state_initialize(void)
{
 Out1 = 0.0;
 UnitDelay1_DSTATE = UnitDelay1_X0;
}

void example_state_step(void)
{
 UnitDelay1 = UnitDelay1_DSTATE;
 Add = Constant_Value + UnitDelay1;
 Out1 = Add;
 UnitDelay1_DSTATE = Add;
}
```

Figure 2.1: A SIMULINK diagram representing a counter (the bottom-right block represents a state element), and the code generated from this block by the REAL-TIME WORKSHOP code generator. The generated code has the form of two functions corresponding to the initial state and the step. The variable UnitDelay1_DSTATE is a state variable, which is initialized to a constant stored in UnitDelay1_X0 (by default this value is 0).

figure. These two functions appear within a template program that includes all the necessary declarations.

Model simulation and code generation

Model simulation is done in several phases. In the first phase the code generator converts the model to an executable form following these steps:

- It evaluates the parameters (given as expressions over constants and Matlab's workspace constants) of the model's blocks;
- It determines signal attributes, e.g., name, data type, numeric type, dimensionality and sample rate (these are not explicitly specified in the model file) and performs type-checking, i.e., it checks that each block can accept the signals connected to its inputs;
- It flattens the model hierarchy by replacing virtual subsystems with the blocks that they contain;
- It sorts the blocks according to the execution order.

After this phase, the blocks are executed in a single loop corresponding to a single step of the model.

Many optimizations are performed during the code generation phase, such as loop unwinding, introduction of auxiliary variables, elimination of variables by expression propagation, simplifications of expressions, etc. Some of these optimizations cause a simple proof of equivalence by induction to fail, as we will demonstrate in Chapter 4.

2.2 Examples of SIMULINK Blocks

The block is an entity which defines a relation between its inputs and outputs. The block's functionality can vary, depending on the blocks parameters and inputs. It can also be influenced by other blocks. The number of blocks inputs and outputs can vary also. Each input and output has a dimensionality and it can be a scalar, vector or a matrix signal.[1] We will give a representative example for most of these cases.

The Gain block (Fig. 2.2) multiplies the input by a constant value which is specified by the Gain parameter. The Gain can be a scalar or a vector.

Gain	Input	Output
5	7	35
5	[3,7]	[15,35]
[5,4]	7	[35,28]
[5,4]	[3,7]	[15,28]

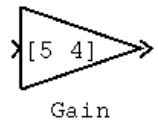


Figure 2.2: A Gain block and sample values.

The Sum block (Fig. 2.3) performs addition or subtraction on its inputs. This block can add or subtract scalar or vector inputs. If the block has a single input signal, then it collapses its elements into a scalar by summing or subtracting them. The operation of the block is specified with the "List of signs" parameter, which is a

[1] TVS does not support matrix signals.

list of Plus (+) and minus (-) signs, and indicates the operations to be performed on the inputs. If there are two or more inputs, then the number of + and - characters must equal the number of inputs. For example, "+-+" requires three inputs and configures the block to subtract the second (middle) input from the first input, and then add the third input. All non-scalar inputs must have the same dimensions. Scalar inputs are expanded to have the same dimensions as the other inputs.

List	Input1	Input2	Output
+	[5,7]	—	12
++	[5,7]	[3,4]	[8,11]
++	[5,7]	3	[8,10]
+-	[5,7]	3	[8,4]

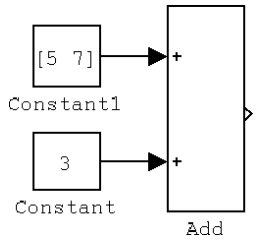


Figure 2.3: An Add block and sample computations.

The Mux block (Fig. 2.4) combines its inputs into a single vector output. An input can be a scalar or a vector signal. The Mux block's "Number of Inputs" parameter allows to specify input signal names and sizes as well as the number of inputs in the following formats:

- Scalar - Specifies the number of inputs to the Mux block. When this format is used, the block accepts scalar or vector signals of any size.
- Vector - The length of the vector specifies the number of inputs. Each element specifies the size of the corresponding input. For example, [2 3] specifies two input ports of sizes 2 and 3, respectively.

The Mux block enables the user to merge outputs of blocks.

The Subsystem block. (Fig. 2.5) A subsystem is a set of blocks that have been replaced by a single block called a Subsystem block. As a model increases in size and

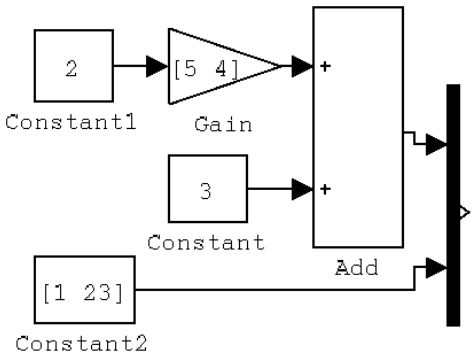


Figure 2.4: A Mux block. The *number of inputs* parameter is *2*

complexity, it can be simplified by grouping blocks into subsystems. Using subsystems helps to reduce the number of blocks displayed in the model window, allows to keep functionally-related blocks together, and enables to establish a hierarchical block diagram. The whole SIMULINK model, composed of blocks and subsystems is placed in a single system block referred to as a *root* system.

A subsystem can be executed *conditionally* or *unconditionally*. A conditionally executed subsystem may or may not execute, depending on the value of an input signal, called the control signal. Such a subsystem executes while the control signal is positive. When such a subsystem is disabled, the output signals are still available to other blocks. The user can decide whether these signals are held at their previous values or are reset to their initial (default) values. The same option is given to the user with respect to state variables of such a subsystem.

The Outport block. The outport block in a subsystem represents its outputs. A signal arriving to an Outport block in a subsystem flows out of the associated output port on that Subsystem block. For example in Fig. 2.5), the Outport block SubOut1 on the right represents the output port SubOut1 of the subsystem on the left.

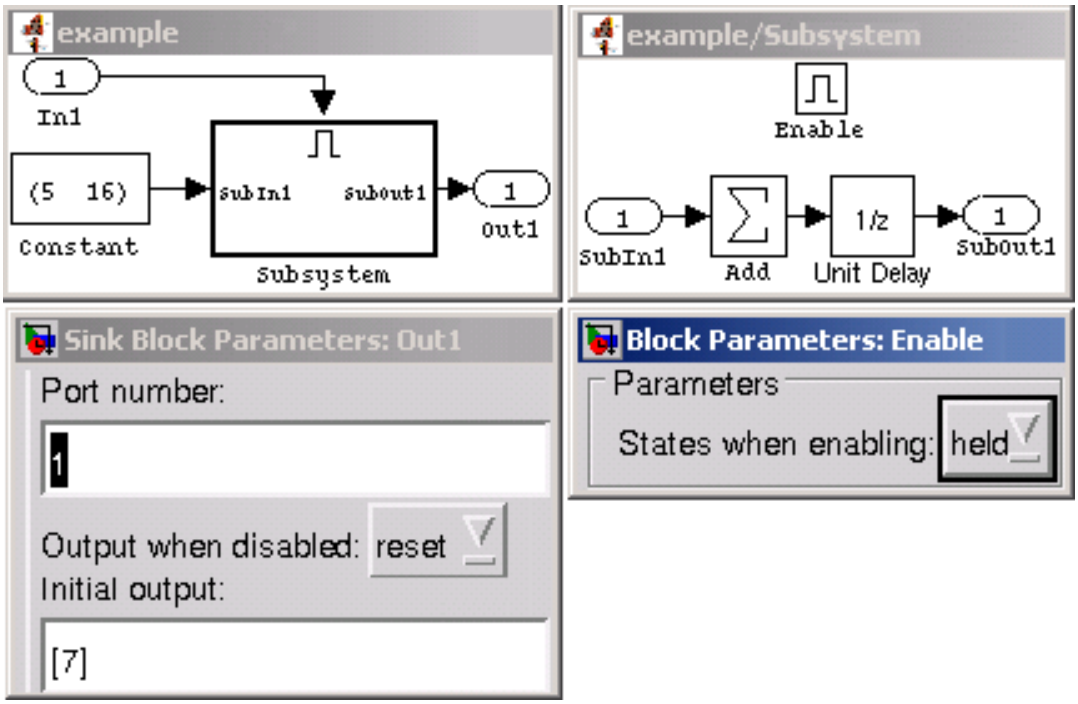


Figure 2.5: An Enabled subsystem. The subsystems control input is associated with the Enable block. This subsystem will output the value 7 when disabled and its states will be held at their previous values.

2.2.1 Block types

SIMULINK blocks can be categorized. This categorization was described in [7] and [8]. The generation of the verification condition in TVS is organized according to these categories.

SIMULINK blocks can be categorized into virtual and non-virtual. A virtual block is one that defines only the interconnections of signals and has no memory element (For example *Mux*, *Outport* and *Subsystem* blocks). Such a block has no explicit representation in the generated code. Non-virtual blocks normally represent some mathematical operation on their input values (for example the *Gain* and *Sum* blocks). A non-virtual block can be represented in the generated code by a variable or its operation can be propagated.

SIMULINK blocks can also be categorized into *transparent* and *opaque*. Opaque blocks are a subset of virtual blocks. Roughly explained, transparent blocks have a behavior, they can be simulated, and contribute to the system's behavior by changing signals values (e.g., the *Sum* block). Opaque blocks may not themselves influence signals values at all, but have an impact on other blocks behavior (e.g., the *Outport* block, which does not have an output but represents the subsystems output). Thus, the SIMULINK block semantics not only depends on the block type and its actual parameters, but also on whether it is located in the scope of some opaque block.

2.3 Textual Representation of the Model

There are two textual representations of the model:

- The *model.mdl* file, which is written in a Mathworks propriety markup language. The file contains the graphical model description and assignments to parameters of template blocks. SIMULINK allows not to specify blocks parameters that can be derived, i.e., propagated from other blocks automatically (for example – input signals types). Therefore, in the model file not all parameters are contained explicitly for all blocks. Blocks parameters can be defined in terms of Matlab workspace variables, but those values are also not included in the *model.mdl* file. Thus, the *model.mdl* file is tightly coupled with the Matlab environment.

- The *model.rtw* file, which is derived from model.mdl during code generation. It is an intermediate representation created by removing graphical information from model.mdl, and evaluating parameters of blocks. Although it contains more information, its format is not described in Mathworks' documentation and is difficult to understand by reverse engineering.

TVS, however, does not access these files directly. Instead it processes model.mdl in an indirect way with routines that are provided by the Matlab script language, which can be run from inside the Matlab environment. This approach has few advantages over reading model.mdl directly:

1. It gives us a direct access to the data without a need to develop a parser. This also makes TVS less sensitive to possible changes in the format of model.mdl in future versions.

2. The Matlab script language gives access to constants that are defined as part of the Matlab workspace, and are needed for generating the verification condition.

3. It allows easy access to information that is calculated and propagated through the blocks, like type information, which does not appear explicitly in model.mdl.

It should be noted that the information about the semantics of the blocks can only be obtained from the Mathworks manuals [8]. Thus, TVS translates the blocks according to our interpretation of their expected functionality, based on the informal description in the manuals.

Chapter 3

The Verification Condition

3.1 The Computational Model

The process of translation validation is based on the *synchronous transition systems* (STS) computational model. The description here follows closely the one in [12].

Synchronous transition system. Given a set of typed variables V, a state s over V is a type-consistent interpretation of the variables in V. Let Σ_V denote the set of all states over V.

A *synchronous transition system* $A = (V, \Theta, \rho)$ consists of:

V - a finite set of typed variables

Θ - a satisfiable assertion over the state variables, characterizing the initial states of system A

ρ - an assertion over A's inputs, outputs, current and next step variables, representing a transition relation.

ρ relates a state s to its possible successors s' by referring to both unprimed and primed versions of variables in V. Unprimed variables are interpreted according to s, primed variables according to s'.

A *computation* of A is an infinite sequence $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$, with $s_i \in \Sigma_V$ for each $i \in \mathbb{N}$, which satisfies $s_0 \models \Theta$ and $\forall i \in \mathbb{N}.\ (s_i, s_{i+1}) \models \rho$.

3.2 The "Correct Implementation" Relation

The notion of correct implementation used in translation validation is based on the general concept of refinement between synchronous transition systems.

Let $A = (V_A, \Theta_A, \rho_A, E_A)$ and $C = (V_C, \Theta_C, \rho_C, E_C)$ be an abstract and concrete STS's, where $E_A \subseteq V_A$ and $E_C \subseteq V_C$ are externally observable variables (outputs in our case).

Given a subset of V variables $F \subseteq V$ and a state s over variables in V, denote by s^F a projection of s on F. An *observation* of STS $T = (V_T, \Theta_T, \rho_T, E_T)$ is any infinite sequence of elements which can be obtained by projecting each state in a computation of T to variables in E_T. That is, a sequence which has the form $s_0^{E_T}, s_1^{E_T}, \ldots$, for some $\sigma : s_0, s_1, \ldots$, a computation of T. We denote by $Obs(T)$ the set of observations of STS T. We say that system C *refines* system A, if $Obs(C) \subseteq Obs(A)$. That is, if every observation of system C is also an observation of A.

3.3 Proving Refinement by Simulation

The proof of equivalence is based on induction, assuming we have a correct mapping between the state variables and inputs in both programs.

As before, let $A = (V_A, \Theta_A, \rho_A, E_A)$ and $C = (V_C, \Theta_C, \rho_C, E_C)$ be a given abstract and concrete transition systems. For STS $T \in \{A, C\}$ denote by V_T^S the set of T's state variables, by V_T^I the set of its input variables and, finally, V_T^O the set of its output variables (V_T^I, V_T^S and V_T^O are disjoint). For a reactive control system the natural observation is its outputs. Therefore for STS T, $E_T = V_T^O$. Let $map : V_A \mapsto V_C$ be a mapping function. The verification condition corresponding to the base case is:

$$\bigwedge_{s \in V_A^S} s = map(s) \wedge \Theta_C \Rightarrow \Theta_A \ . \tag{3.1}$$

This condition ensures that the initial states of the target are legitimate initial states in the source. The verification condition corresponding to the step is:

$$\begin{array}{l} \bigwedge_{i \in V_A^I} i = map(i) \wedge \bigwedge_{s \in V_A^S} s = map(s) \wedge \rho_A \wedge \rho_C \quad \Rightarrow \\ (\bigwedge_{s \in V_A^S} s' = map(s') \wedge \bigwedge_{o \in V_A^O} o' = map(o')) \end{array} \ . \tag{3.2}$$

Validation of this condition ensures that the equality between the outputs of the source and the target is propagated through program computation. The condition states that when values of corresponding inputs are the same and the state variables in the current state are the same, then the transition relation implies that the outputs and states in the next step are equal.

The verification condition must hold over the set of all reachable concrete states. In practice it is very difficult to compute a characterization of the reachable concrete states set for even the simplest systems. Therefore we have to make the stronger requirement which is that the inductive condition holds with respect to all states satisfying some invariant Inv of C. If Inv is indeed a C-invariant then all C-reachable states must satisfy Inv and, therefore, inductiveness over all Inv-states clearly implies inductiveness over all reachable states[1]. The conditions that ensure this are:

$$\begin{array}{l} \Theta_C \Rightarrow Inv \\ Inv \wedge \rho_C \Rightarrow Inv' \end{array} \tag{3.3}$$

Example. An example of a small model, the generated code, and the full verification condition that is generated by TVS appears in Appendix B. It includes an abstraction with uninterpreted functions that is described in the next section.

[1] A too-loose invariant leads to incompleteness.

Chapter 4

Generating the Verification Condition

There are various technical issues that need to be resolved when generating the verification condition that was described in the previous chapter.

4.1 Generating the Source Transition Relation and Initial Condition.

Generating the source transition relation from the given model is done as follows. TVS iterates over all model blocks, and for each block generates its transition relation. The block transition relation depends on the block's type and its parameters. When the block is located in a conditionally executed subsystem its output value is also influenced by the value of the enabling condition. If the enabling condition evaluates to true, then the block has its normal output. If the enabling condition evaluates to false, the block's output is undefined. As describes in §2.2, conditionally executed subsystems have parameters that define what happens to the subsystem's state and output when the subsystem is disabled. These parameters influence the transition relation.

The initial condition is generated in a similar way. Only the blocks that have states or initial output participate in it.

4.2 Generating the Target Transition Relation and Initial Condition

Generating the target transition relation from the C code that is generated by Real-Time Workshop is rather straightforward since it is restricted to a subset of the C language that forbids unbounded loops in the *model_step* function.

The type of C expressions used in this code are naturally translated to Yices' format. For example C's:

$$a = b + c; \tag{4.1}$$

will be translated into the formula (all formulas in this section are given in Yices's format [3]):

$$(= \quad a \quad (+ \quad b \quad c)) \tag{4.2}$$

However, C represents asynchronous computations whereas the transition relation should be represented by a formula for the purpose of proving it with a theorem prover. For example, Real-Time Workshop reuses local variables for holding intermediate calculation results, thus assigning them a value more than once. To cope with this problem, tvs uses Static Single Assignment (SSA) [2].

Static Single Assignment

Static single assignment form is a convenient formulation with which we can turn a program into a formula that represents the legal computations in this program. We derive SSA from loop-free code as follows. In each assignment of the form `x = exp;` the left-hand side variable `x` is replaced with a new variable, say $\texttt{x}_1$. Any reference to `x` after this line and before `x` is assigned again is replaced with the new variable $\texttt{x}_1$. In addition, assignments are guarded according to the control flow. After this

transformation, the statements are conjoined: the resulting equation represents the computations of the original program.

Example 1 *The following C code fragment demonstrates SSA.*

$$\begin{aligned} &a\_tmp = 3 * 5; \\ &b = a\_tmp; \\ &a\_tmp = a\_tmp + 1; \end{aligned} \tag{4.3}$$

Here the variable a_tmp has three values: an initial value before it is assigned, a value after the first assignment and a value after the second assignment. If translated straightforward, the code will result in the following transition relation, which is contradictory.

$$\begin{aligned} &(and \\ &\quad (= \quad a\_tmp' \quad (* \quad 3 \quad 5)) \\ &\quad (= \quad b \quad a\_tmp') \\ &\quad (= \quad a\_tmp' \quad (+ \quad a\_tmp' \quad 1)) \\ &) \end{aligned} \tag{4.4}$$

The SSA representation solves this problem:

$$\begin{aligned} &a\_tmp\_\_1 = 3 * 5; \\ &b = a\_tmp\_\_1; \\ &a\_tmp\_\_2 = a\_tmp\_\_1 + 1; \end{aligned} \tag{4.5}$$

Note that here all variables are assigned at most once. We can now conjoin the assignments and gain a formula that represents computations that can be mapped to computations through the original program.

Example 2 *Now consider the following program, which contains* if *statements:*

$$\begin{array}{l} if(v)\{ \\ \quad a = b; \\ \}else\{ \\ \quad a = c; \\ \} \\ d = a; \end{array} \tag{4.6}$$

The question is which a will be assigned to d? The correct conversion into SSA form is:

$$\begin{array}{l} if(v\_\_0)\{ \\ \quad a\_\_0 = b\_\_0; \\ \}else\{ \\ \quad a\_\_1 = c\_\_0; \\ \} \\ a\_\_2 = v\_\_0?a\_\_0 : a\_\_1; \\ d\_\_1 = a\_\_2; \end{array} \tag{4.7}$$

TVS does not explicitly generate the SSA form but computes the transition relation directly using the SSA rules.

4.3 Variable Mapping

To map between the source and target variables, TVS relies on REAL-TIME WORKSHOP's naming convention that defines the variable names in the C code, based on their block name and their usage. For example, the variable representing the output of the block named "block1" can be named rtb_block1, etc. Currently TVS can fail to find automatic mapping in a few complex cases. In general, giving blocks different names, which is also advised by the SIMULINK modelling methodology, solves this problem. Another solution for this problem is discussed in the future work section.

Another problem is that mapping of state variables is not always trivial. As we mentioned in section 2.2, the subsystem can hold its output on the previous value when it is disabled. Thus, the subsystem output acquires a state. However, the subsystem is a virtual block (section 2.2.1) and thus does not have a matching variable representing its output in the C code. For storing the subsystems last output value, Real-Time Workshop uses the variable representing the last non-virtual block in the path to the subsystems Outport block. Thus, to map a conditionally executed subsystem outputs, tvs traces the last non-virtual block in the path to each subsystems Outport block and maps it accordingly.

4.4 Invariants

While variables in a synchronous model are *volatile* (i.e., they have an undefined value in a step in which they are not updated), the corresponding global variables in the generated C code preserve their values between steps. Indeed, Real-Time Workshop generates code that updates the variables only as needed, while relying on their previous value otherwise. This fails the inductive proof.

An example of this case is Real-Time Workshop's handling of Simulink's mechanism of enabling and disabling subsystems. A subsystem is disabled if some predicate *condition* holds, and then its output is defined to be either maintained from the previous step or reset to some default value, depending on the user's choice. Real-Time Workshop introduces a variable $MODE$ that is set to *Enabled* if *condition* holds and to *Disabled* otherwise (this variable is needed separately from *condition* because some updates are required only when the mode changes).

The left and right code segments in Fig. 4.1 correspond to such a subsystem (i.e., here translated to C according to Simulink's semantics) and the code generated by Real-Time Workshop, respectively. Both represent part of the computation of the step. Real-Time Workshop declares `output` as global, and updates it only in the step function. It exploits this fact for avoiding an update when the value

```
if (condition)                    if (condition) {
  MODE = ENABLED;                   if (MODE == DISABLED) MODE = ENABLED;
else {                            } else if (MODE == ENABLED) {
  MODE = DISABLED;                    MODE = DISABLED;
  output = 1;                         output = 1;
}                                 }
...                               ...
```

Figure 4.1: An example of a code that fails the induction condition without an invariant. The code on the left represents the model behavior, and the code on the right, the C code behavior.

should remain the same. Thus, while it is being assigned its default value of 1 in the SIMULINK model in every step in which the subsystem is disabled, it performs such an assignment in the generated C code only when it becomes disabled. This clearly invalidates an inductive argument as to the value of MODE and `output`, which forces us to add invariants.

For each subsystem that has an enabling condition TVS adds an invariant of the form

$$\text{MODE} = \text{DISABLED} \vee \text{MODE} = \text{ENABLED}$$

and for each output *output* in such a subsystem that is configured to be reset to a default value *val* ($val = 1$ in the example above) it also adds an invariant of the form

$$\text{MODE} = \text{DISABLED} \rightarrow \text{output} = val\ .$$

4.5 The use of Abstraction

The model may use arbitrary functions, whereas the code generator refers to all functions as discrete and over a finite domain. We may attempt to decide the verification condition while referring to the variables as ranging over this finite domain (integers, float...), but this solution has little chance to scale. A standard solution in the translation validation literature is to abstract the functions, using *uninterpreted functions*. For example, instead of proving:

$$a_1 = (x_1 * x_2) \wedge a_2 = (y_1 * y_2) \wedge z = a_1 + a_2 \Rightarrow (z = (x_1 * x_2) + (y_1 * y_2)) \ ,$$

prove an abstracted formula:

$$a_1 = f(x_1, x_2) \wedge a_2 = f(y_1, y_2) \wedge z = g(a_1, a_2) \Rightarrow (z = g(f(x_1, x_2), f(y_1, y_2)) \ .$$

In general this type of abstraction may introduce incompleteness, but in our case we know in advance the type of transformations done by the code generator, and change the verification conditions accordingly. Two examples of such modifications that we found necessary and TVS accordingly supports are:

1. *Commutativity.* This kind of abstraction prevents us from being able to prove an equivalence such as $a * b = b * a$, because the verification condition does not include information about commutativity. Indeed, Real-time workshop has optimizations that rely on commutativity. This problem can be solved by modifying the verification condition. Specifically in Yices, adding the statement

   $$(forall\ (a :: int\ b :: int)\ (=\ \ (f\ a\ b)\ (f\ b\ a))) \ ,$$

   solves this problem.[1] An alternative solution is to explicitly reduce the uninterpreted functions to equality logic using Ackermann's reduction [1], which gives more flexibility, including a solution to the commutativity problem. For example, given two uninterpreted functions $f(a, b), f(c, d)$ which we want to interpret as commutative, we can add the constraint

   $$(a = c \wedge b = d \vee a = d \wedge b = c) \Longrightarrow f_1 = f_2$$

   where f_1 and f_2 are new variables replacing the respective uninterpreted function instances.

[1] Using this construct leads to incompleteness in Yices, however, although rarely we encountered this in practice.

2. *Constant propagation.* Consider the following example taken from a real SIMULINK model (rather the equivalent expression that TVS generates from it) and the respective C code that was generated by REAL-TIME WORKSHOP:

$$\begin{aligned} & c\_m = 1 \quad \wedge \\ & state\_M = y\_m + x\_m * (c\_m * 1/100) \end{aligned}$$

in the source file, turned into:

$$state\_C = y\_c + x\_c * (1/100)$$

in the target code.

Even assuming we have a correct variable mapping where state_C is mapped to state_M, we cannot prove their equivalence if we replace multiplication with uninterpreted functions. To solve this problem, TVS applies constant propagation when generating the part of the verification condition that represents the source code.

The small example in Appendix B includes an abstraction with an uninterpreted function of the multiply function.

Chapter 5

Implementation

5.1 Tools

In our implementation we use three external tools:

CPP - A C pre-processor. We use it to combine the C code generated by Real-Time Workshop into a single file.

CTool - A C parser front end. CTool parses a C code and generates a data structure that represents a parse tree. tvs works with this parsing tree.

Yices - An SMT solver. We use it to prove the verification condition that tvs produces.

5.2 tvs Architecture

The architecture of the translation validation tool appears in Figure 5.1. The main module of tvs is written in C++. The module receives as input the Simulink model file *model.mdl*, the name of the verified subsystem, the generated C code and several more parameters configuring its execution. The C code has to be in one file (this limitation is imposed by CTool). Since RTW's output comes in several C files, we

use CPP to combine them into a single C file. Then the TVS execution continues as follows:

- The main module generates the target transition relation, from which the module derives the list of final variable names. The variable names (Sect. 4.2) are constructed from their original names and SSA indices.
- The main module calls the Matlab module (written in the Matlab script language). The main module transfers to the Matlab module the list of final variable names.
- The Matlab module generates the source transition relation and the mapping between the source and target variables (see Sect. 3.3).

The verification condition for the induction base is produced in a very similar way.

The resulting source transition relation, target transition relation and mapping are combined into a verification condition that is then checked with Yices.

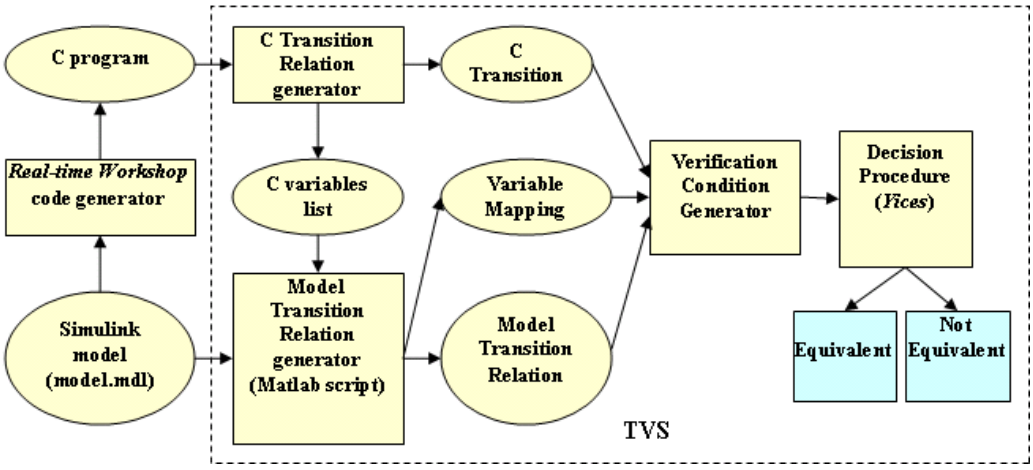


Figure 5.1: The architecture of the translation validation tool.

Chapter 6

Related Work

The Translation Validation approach was first proposed by Pnueli et al. [14]. Since then translation validation was successfully applied to several specification languages. Pnueli et al. in [12] applied it to translation from the Signal synchronous language, which is used in industrial applications for the development of safety-critical, reactive systems. Pnueli et al. in [10] developed the CVT tool for validation of a code generator of DC+, a common format for synchronous languages. Zuck et al. in [16] addressed the problem of validating SGI Pro-64 compiler suit for Intel Itanium systems.

The SIMULINK model is difficult for formal reasoning, because the SIMULINK software package is not accompanied by a formal definition of the semantics. Several works addressed this problem by translating the SIMULINK model into a more formal specification language. [7] introduce the ClawZ tool that translates a SIMULINK model into a Z specification for further validation. The first version of ClawZ was developed in 1999. In its translation, ClawZ relies on *model.rtw* and therefore does not have the advantages of using the Matlab script language – listed in § 2.3 – as TVS does. ClawZ supports a wider range of SIMULINK blocks and configuration than TVS does.

A similar idea is followed by Tripakis et al. in [15]. The main objective of this project is to allow the translation of SIMULINK/Stateflow into the synchronous programming language Lustre, allowing its associated compilers, model-checkers and abstract interpretation tools to be applied to SIMULINK designs. The project is divided

into two, more-or-less independent tools, Simulink2Lustre and Stateflow2Lustre and is capable of translating SIMULINK charts with Stateflow components. Simulink2Lustre supports some SIMULINK library subset that seems to be more or less similar to the library TVS supports. Also, like TVS, Simulink2Lustre begins from the *model.mdl* file. However, because it does not use Matlab interface to access the file, it needs to perform such steps as type inference and clock inference by itself. For the same reason Simulink2Lustre, like ClawZ, is restricted to a limited number of Matlab revisions.

The purpose of [6] , very similar to TVS, is to develop an automated tool that verifies correctness of the generated code, however their method differs significantly. They use comments placed by REAL-TIME WORKSHOP into the generated code for improved readability, to convert the code back into a model file, which is then compared by one of the off-the-shelf tools ([4],[5]) against the original model. This comparison is very weak, in the sense that it fails even when there is a small syntactic difference. Typically, the original model cannot be recovered out of the generated code without using some external help (for example comments). Consequently, the authors had problems translating back constructs more complex than simple blocks. The authors propose as future work to extract a graph representation of the model both from the model.mdl file and from the generated C code. They suggest to use the 'dot' format (a format for describing graphs), with which it is possible to perform graph comparison in order to verify functional equivalence.

Chapter 7

Summary, Status and Experiments

We introduced TVS, a translation validation tool for the code generator Real-Time Workshop, which generates optimized C code from Simulink models. TVS works according to the classic induction-based proof strategy introduced already a decade ago in [14, 11]. Each language, however, poses a new challenge since the verification condition depends on the semantics of the operators used in this language. Further, each code generator poses a challenge owing to its particular optimizations that may hinder the inductive argument. We presented several such problems and the way TVS solves them.

Currently TVS supports a basic subset of Simulink blocks, their various configurations and combinations. This imposes a limitation on the models for which the verification condition can be generated fully automatically by TVS. For models with blocks that are not supported, TVS emits a warning and generates a partial verification condition which can then be changed manually. Adding support for new blocks in TVS is simple given the semantics of the block. Since the vast Simulink block library is expressed by a basic set of C constructs, no additional work in developing the target transition relation generator is expected.

Experiments We tested TVS on various small models, and one large industrial example. The large model that we verified automatically (with some minor manual

modifications listed in Appendix A) with TVS is called "rtwdemo_fuelsys", which is a model of a fuel injection controller that is distributed together with Matlab. This model has about 100 blocks, and the generated C code for the step has about 250 lines. The generated verification condition in the format of Yices has 790 lines. Yices solves it in about a second. Small perturbations of the verification condition that invalidates it cause Yices to find counterexamples typically within 10 seconds.

Appendix A

Manual Modifications to the rtwdemo_fuelsys Example

Following is a list of manual modifications that we performed in order to be able to prove the rtwdemo_fuelsys example.

1. The presence of a stateflow unit in the model and the fact that TVS does not yet support it, causes a problem to isolate the rest of the model in the C code for the sake of translation validation. We solved this problem by changing the model such that all the components other than the stateflow subsystem are grouped in one subsystem. The functionality of the model is unaffected by this change.

2. Two blocks that appear in the rtwdemo_fuelsys example and TVS does not yet support are 'lookup table' and 'discrete transfer function'. We replaced the former with a different block of the same cardinality ('add') and removed the latter, which has cardinality of 1.

3. Block and subsystems can be enabled or disabled by testing a numeric condition on some input signal. A block is enabled, according to the SIMULINK manual, if this condition is greater than 0. When the enabling condition is a Boolean

signal, however, Real-Time Workshop generates code that checks whether it is *different* than 0, which is a semantic preserving optimization (checking equivalence to 0 is cheaper than checking whether a variable is greater than 0). This optimization fails the verification condition unless some modification to the verification condition is performed. We see two options for solving this problem. The first option is to perform the same optimization on the model side, and hence obtain the same effect on both sides. The second option is to leave the 'greater-than' sign interpreted, and add an invariant that such variables are equal to either 0 or 1. We currently support the second solution.

4. Problems in mapping inputs. In the rtwdemo_fuelsys example Real-Time Workshop replaces the name of some of the inputs (specifically the inputs In5 and In6) with the names of the outputs of the stateflow subsystem ('fail_state' and 'fuel_mode', respectively) which generate these inputs for the subsystem that we check. It prints the original names in a C comment adjacent to the declaration of these variables. Our current solution is to manually map these variables. A 'quick-and-dirty' solution would be to read the comments. A better solution is probably possible with further reverse-engineering, i.e., investigating in which cases Real-Time Workshop performs such changes.

5. Problem in mapping multiple blocks with the same name. Simulink permits equivalent names for blocks as long as they are in different subsystems. Real-Time Workshop has two different options of handling this case. The first is to compute some unique string based on the environment of the subsystem. The computation of this string is not documented. Real-Time Workshop supports another naming convention, by which the number of the subsystem is appended to the block name, thus making it unique. We did not find a way, however, to retrieve the subsystem number from the .mdl file. A simple solution is to rename blocks so no two blocks will have the same name. It is perhaps also possible to trace back Real-Time Workshop's calculation method of the subsystem number and name the blocks accordingly.

6. Fractional constants, originating from constants declared as float and double, are printed in the source and the target with a different number of digits after the dot, which fails the verification condition. For the source TVS computes the values of the constants based on a Matlab function that returns 15 digits, whereas the code generator uses 17 digits. We currently fix this difference manually, although it can probably be solved automatically by either searching for the appropriate Matlab function that prints the fractions with the same precision as the code generator, or alternatively change the target of the code generator to fixpoint.

Appendix B

A Full Example

The initialization and step functions that are generated by Real-Time Workshop for the model in Fig. B.1 appear below.

```
/* Macros for accessing real-time model data structure */
/* Block signals (auto storage) */
typedef struct {
  real_T UnitDelay1; /* '<Root>/Unit Delay1' */
  real_T Product; /* '<Root>/Product' */
} BlockIO;
/* Block states (auto storage) for system '<Root>' */
```

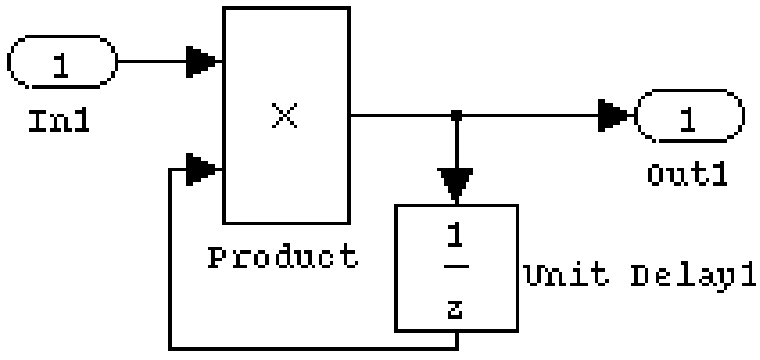


Figure B.1: A Simulink model example.

```
typedef struct {
  real_T UnitDelay1_DSTATE; /* '<Root>/Unit Delay1' */
} D_Work;
/* External inputs (root inport signals with auto storage) */
typedef struct {
  real_T In1; /* '<Root>/In1' */
} ExternalInputs;
/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T Out1; /* '<Root>/Out1' */
} ExternalOutputs;
/* Parameters (auto storage) */
struct Parameters {
  real_T UnitDelay1_X0; /* Expression: 0
                         * '<Root>/Unit Delay1'
                         */
};

/* Parameters (auto storage) */
typedef struct Parameters Parameters;
Parameters rtP = {
  0.0                   /* UnitDelay1_X0 : '<Root>/Unit Delay1'
                         */
};
/* Block signals (auto storage) */
BlockIO rtB;
/* Block states (auto storage) */
D_Work rtDWork;
/* External inputs (root inport signals with auto storage) */
ExternalInputs rtU;
/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs rtY;

/* Model step function */
void example_step(void)
```

```
{
  /* UnitDelay: '<Root>/Unit Delay1' */
  rtB.UnitDelay1 = rtDWork.UnitDelay1_DSTATE;
  /* Product: '<Root>/Product' incorporates:
   *  Inport: '<Root>/In1'
   */
  rtB.Product = rtU.In1 * rtB.UnitDelay1;
  /* Outport: '<Root>/Out1' */
  rtY.Out1 = rtB.Product;
  /* Update for UnitDelay: '<Root>/Unit Delay1' */
  rtDWork.UnitDelay1_DSTATE = rtB.Product;
}

/* Model initialize function */
void example_initialize(void)
{
  /* block I/O */
  {
    rtB.UnitDelay1 = 0.0;
    rtB.Product = 0.0;
  }
  /* external inputs */
  rtU.In1 = 0.0;
  /* external outputs */
  rtY.Out1 = 0.0;
  /* InitializeConditions for UnitDelay: '<Root>/Unit Delay1' */
  rtDWork.UnitDelay1_DSTATE = rtP.UnitDelay1_X0;
}
```

The verification condition generated by TVS.

Base condition:

```
;==============OUT==================
(reset)
```

```
;==============DECLARATION==================
(define-type double real)
(define-type real_T double)
(define-type boolean int)
(define-type boolean_T int)
(define-type int_T int)
(define-type uint8 int)
(define-type uint_T int)
(define-type uint8_T int)
(define-type uint32_T int)

(define UnitDelay1_DSTATE__0::real_T)
(define Out1__0::real_T)
(define UnitDelay1_X0__0::real_T)
(define UnitDelay1__0::real_T)
(define Product__0::real_T)
(define In1__0::real_T)
(define UnitDelay1__1::real_T)
(define Product__1::real_T)
(define UnitDelay1_DSTATE__1::real_T)
(define In1__1::real_T)
(define Out1__1::real_T)
(define example_In1_out0'::(tuple double))
(define example_Product_out0'::(tuple double))
(define example_Unit_Delay1_out0'::(tuple double))
(define example_Unit_Delay1_state::(tuple double))
(define example_Unit_Delay1_state'::(tuple double))
(define example_out0'::(tuple double))

(define mult::(-> double double double))
(assert (forall
```

```
        (x::double y::double z::double)
        (= (mult (mult x y) z) (mult x (mult y z)))
      )
)
(assert (forall (x::double y::double) (= (mult x y) (mult y x))))
(assert (forall (x::double) (= (mult x 1) x)))

(assert
   (not
      (=>
         (and
            ;===========TARGET INITIAL CONDITION================
            (= UnitDelay1_X0__0 (/ 00 10))
            (and
               (and
                  (= UnitDelay1__1 (/ 00 10))
                  (= Product__1 (/ 00 10))
               )

               (= In1__1 (/ 00 10))
               (= Out1__1 (/ 00 10))
               (= UnitDelay1_DSTATE__1 UnitDelay1_X0__0)
            )
            ;==============Mapping=================
            (= (select example_Unit_Delay1_state' 1) UnitDelay1_DSTATE__1 )
         ) ;END and
        ;==============THEN==================
        ;==============Source Base Transition=================
        (and
             (= (select example_Unit_Delay1_state' 1) 0)
        ) ;END and
```

```
      )
   )
)
(check)
```

Step condition:

```
;==============OUT=================
(reset)
;==============DECLARATION==================
(define-type double real)
(define-type real_T double)
(define-type boolean int)
(define-type boolean_T int)
(define-type int_T int)
(define-type uint8 int)
(define-type uint_T int)
(define-type uint8_T int)
(define-type uint32_T int)

(define UnitDelay1_DSTATE__0::real_T)
(define Out1__0::real_T)
(define UnitDelay1_X0__0::real_T)
(define UnitDelay1__0::real_T)
(define Product__0::real_T)
(define In1__0::real_T)
(define UnitDelay1__1::real_T)
(define Product__1::real_T)
(define Out1__1::real_T)
(define UnitDelay1_DSTATE__1::real_T)
(define example_In1_out0'::(tuple double))
```

```
(define example_Product_out0'::(tuple double))
(define example_Unit_Delay1_out0'::(tuple double))
(define example_Unit_Delay1_state::(tuple double))
(define example_Unit_Delay1_state'::(tuple double))
(define example_out0'::(tuple double))

(define mult::(-> double double double))
(assert (forall (x::double y::double z::double)
            (= (mult (mult x y) z) (mult x (mult y z)))
      )
)
(assert (forall (x::double y::double) (= (mult x y) (mult y x))))
(assert (forall (x::double) (= (mult x 1) x)))

(assert
   (not
      (=>
         (and
           ;===========TARGET TRANSITION RELATION================
           (= UnitDelay1_X0__0 (/ 00 10))
            (and
               true
               (= UnitDelay1__1 UnitDelay1_DSTATE__0)
               (= Product__1 (mult In1__0 UnitDelay1__1))
               (= Out1__1 Product__1)
               (= UnitDelay1_DSTATE__1 Product__1)
            )
            ;===========SOURCE TRANSITION RELATION================
            ;==============Mapping Before Step=================
            (= (select example_In1_out0' 1) In1__0 )
            (= (select example_Unit_Delay1_state 1) UnitDelay1_DSTATE__0 )
```

```
            ;==============Transition Relation=================
            (= (select example_Product_out0' 1)
               (mult
                   (* 1 (select example_In1_out0' 1))
                   (* 1 (select example_Unit_Delay1_out0' 1))
               )
            )
            (=
                (select example_Unit_Delay1_state' 1)
                (select example_Product_out0' 1)
            )
            (=
                (select example_Unit_Delay1_out0' 1)
                (select example_Unit_Delay1_state 1)
            )
            (= (select example_out0' 1) (select example_Product_out0' 1) )

          ) ;END and
        ;==============THEN==================
        ;==============Mapping After Step==================
        (and
             (= (select example_Unit_Delay1_state' 1) UnitDelay1_DSTATE__1 )
             (= (select example_out0' 1) Out1__1 )

        ) ;END and
       )
   )
)
(check)
```

Bibliography

[1] W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM.

[3] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI international, 2006.

[4] EnSoft. *SimDiff - Automatically find differences between Simulink models.* http://www.ensoftcorp.com/SimDiff/.

[5] ExpertControl. *ecDIFF - Graphical differencing for Simulink models.* http://www.expertcontrol.com/en/products/ecdiff.php.

[6] D. D. Graaf, C. Kleinheksel, S. Kothari, K. Korslund, and B. Miller. C2mdl - model based software verification. Technical report, Iowa State University, Ames, Iowa, 2008.

[7] R. Jones. Clawz - the semantics of simulink diagrams. Technical report, Lemma 1 Ltd, 2003.

[8] T. Mathworks. *Simulink 7 user-guide.* http://www.mathworks.com/ access/helpdesk/help/pdf_doc/simulink/sl_using.pdf.

[9] T. Mathworks. *Simulink getting started guide*. http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_gs.pdf.

[10] A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation: From DC+ to C. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, volume 1641 of *Lect. Notes in Comp. Sci.*, pages 137 – 150, 1998.

[11] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT)-automatic verification of a compilation process. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1999.

[12] A. Pnueli, M. Siegel, and O. Shtrichman. *Translation Validation: From SIGNAL to C*, volume 1710 of *LNCS State-of-the-Art Survey*, pages 231–255. Springer-Verlag, 1999.

[13] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. Technical report, Sacres and Dept. of Comp. Sci., Weizmann Institute, Apr. 1997.

[14] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *4th Intl. Conf. TACAS'98*, volume 1384 of *Lect. Notes in Comp. Sci.*, pages 151–166. Springer-Verlag, 1998.

[15] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time simulinkf to lustre. *Trans. on Embedded Computing Sys.*, 4(4):779–818, 2005.

[16] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizingcompilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.