Translation Validation: From Simulink to C

Michael Ryabtsev[1] and Ofer Strichman[2]

[1] Computer Science, Technion, Haifa, Israel
michaelr@cs.technion.ac.il
[2] Information Systems, IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. Translation validation is a technique for formally establishing the semantic equivalence of the source and the 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.

1 Introduction

Translation Validation [8,7,5] is a formal method for validating the semantic equivalence between the source and the target of a compiler or a 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 (frequently this mapping can be derived automatically). Based on the semantics of the source and target languages, it then builds a verification condition that 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. It has the advantage of being less sensitive to changes in the code generator – such changes invalidate correctness proofs of the code generator – and simpler, since verifying the code generator amounts to functional verification of a complex program. Further, when the code generator is not open-source this is the only option.

In this work we present TVS (Translation Validation tool for Simulink). TVS applies translation validation to the (commercial) Real-Time Workshop (RTW) code generator [3], which translates Matlab/Simulink models to optimized C code.[1] We will briefly describe Simulink in the next section. In Sect. 3 we will describe the verification condition generated by TVS, and in Sect. 4 we will highlight various technical issues such as automatic generation of invariants that are needed for the proof and abstraction based on uninterpreted functions. We must assume here, owing to lack of space, that the reader is familiar with such proof techniques.

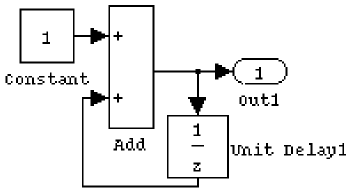
[1] TVS only handles 'classic' Simulink models. Simulink also allows integration of state diagrams in the model, called STATEFLOW, which TVS does not support.

2 Simulink

Simulink [4,3], developed by *The MathWorks*, is a software package for model-based design of dynamic systems such as signal processing, control and communications 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 block's *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 figure. These two functions appear within a template program that includes all the necessary declarations.



```
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;
}
```

Fig. 1. A Simulink diagram representing a counter (the `Unit Delay1` block represents a state element), and the corresponding generated code. The variable UnitDelay1_DSTATE is a state variable, which is initialized to a constant stored in UnitDelay1_X0 (by default this value is 0).

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: 1) it evaluates the block's parameter expressions; 2) it determines signal attributes, e.g., name, data type, numeric type, and dimensionality (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; 3) it flattens the model hierarchy by replacing subsystems with the blocks that they contain; and 4) 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 Sect. 4.

3 The Verification Condition

When considering reactive control systems, code equivalence can be stated in terms of system outputs equivalence, given the same input stream. The output in such systems in each step depends on the current inputs and the values of the state variables. The proof of equivalence is based on induction, assuming we have a correct mapping between the state variables and inputs in both programs.[2]

The inductive verification condition with which TVS attempts to establish the equivalence between the two transition systems can be formulated as follows. For a transition system M, let $init(M)$ be its initial-state predicate over M's state-variables, and let $TR(M)$ be its transition relation over M's inputs, outputs, current and next-step state variables. Denote by $inp(M)$, $out(M)$ and $state(M)$, the sets of M's input, output and state variables respectively (these sets are assumed to be disjoint). We use v' to denote the next-state version of a state variable v. Let S and T be two transition systems corresponding to the source and target of the code generator, and let map be a mapping function between their variables. The verification condition corresponding to the base case is:

$$\bigwedge_{s \in state(S)} s = map(s) \wedge init(T) \Rightarrow init(S) \wedge Inv \ , \tag{1}$$

where Inv is an invariant over T's variables that is possibly needed in order to strengthen the induction claim.[3] This condition ensures that the initial states of the target are legitimate initial states in the source, as well as the validity of the invariant in the initial step.

The verification condition corresponding to the step is:

$$\begin{aligned} &Inv \wedge \textstyle\bigwedge_{i \in inp(S)} i = map(i) \wedge \bigwedge_{s \in state(S)} s = map(s) \wedge TR(S) \wedge TR(T) \quad \Rightarrow \\ &Inv' \wedge \textstyle\bigwedge_{s \in state(S)} s' = map(s') \wedge \bigwedge_{o \in out(S)} o' = map(o') \ . \end{aligned} \tag{2}$$

This condition ensures that if 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. It also guarantees the propagation of the invariant.

[2] Incorrect mapping of the variables leads to incompleteness. Yet the process is incomplete due to other reasons as well. Automatic variable mapping is possible when the code generator preserves variable names in the translation.

[3] The need for such an invariant was already recognized in the original translation validation article [6].

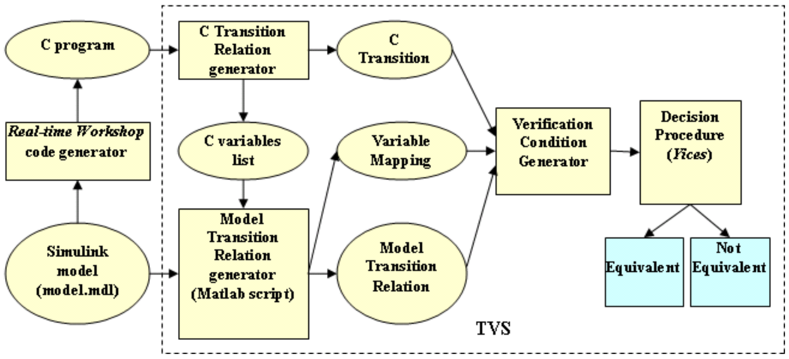

C program
Real-time Workshop code generator
Simulink model (model.mdl)
C Transition Relation generator
C variables list
Model Transition Relation generator (Matlab script)
C Transition
Variable Mapping
Model Transition Relation
Verification Condition Generator
Decision Procedure (Yices)
Equivalent
Not Equivalent
TVS

Fig. 2. The architecture of TVS

After the construction of the two verification conditions, it is left to check them with a decision procedure. For this purpose any one of the existing off-the-shelf SMT solvers can be used. Our tool invokes Yices [2].

The architecture of the translation validation tool appears in Figure 2.

4 Various Technical Issues

Generating the target transition relation. is rather straightforward because RTW uses a rather simple subset of the C language (e.g., no unbounded loops or dynamic memory allocation). However since C represents asynchronous computations whereas the transition relation is a formula, TVS uses Static Single Assignment (SSA) [1] – based on the observation that there are no unbounded loops in the target code – in order to cope with variables that have different values in the course of the computation of the step function. For example, the code `a = b; b = b + 1` is translated into $a_0 = b_0 \wedge b_1 = b_0 + 1$. Multiple assignments to the same variable in a single step (such as b in this example) are common in code generated by RTW owing to its various optimizations.

Generating the source transition relation. is more difficult for two reasons. First, the transitions entailed by each type of block should be defined according to the block's semantics. The large number of such blocks and the fact that the semantics are not formally defined (at least not in the public domain), makes this task difficult. Second, the *model.mdl* file, which represents the Simulink model, is not sufficient when taken out of Matlab's workspace. TVS uses the 'Matlab script language' for retrieving information missing from this file such as the signals' type and constants from the Matlab workspace, for evaluating various expressions, and for working with an internal data structure that represents the model.

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.

If not all state variables are updated explicitly at each step the inductive proof can fail. Indeed, RTW generates code that updates the variables only as needed, while relying on their previous value otherwise.

Consider, for example, the following code fragments. The code on the left is a representation of the model according to Simulink's semantics, whereas the code on the right was generated by RTW for the same model. In the generated code `MODE` is declared as global and is being updated only in the step function. It can therefore avoid an update when the value has not changed. As a result, while in the Simulink model this variable is assigned a value in every step, it performs an assignment in the generated C code only when the value of `condition` changes. This clearly invalidates the inductive argument.

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

In order to solve this problem TVS adds an invariant *Inv* (see Eq. (1) and (2)) of the form MODE = DISABLED $\vee$ MODE = ENABLED. Such cases are associated with conditional executions of blocks, and can be identified automatically.

Abstraction. Whereas the functions in the verification condition range over finite domains (integer, float...), deciding it without abstraction has little chance to scale. A standard solution in the translation validation literature is to make some or all the functions uninterpreted. For example, instead of proving:

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

we can attempt to prove the more abstract formula:

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

where f and g are uninterpreted. In general this type of abstraction may introduce incompleteness, unless 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 *commutativity* and *constant propagation*. As for the former, consider a case in which the code generator swapped the arguments of a multiplication expression, e.g., from $a*b$ to $b*a$. We need to declare the uninterpreted function that replaces the multiplication function as commutative. In Yices this is possible by adding a quantifier of the following form, for an uninterpreted function f:[4]

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

As for the latter, constant propagation is required in a case such as the following: whereas the blocks in the model correspond to two consecutive statements

[4] Using this construct leads to incompleteness in Yices. We rarely encountered this in practice.

`c = 1; y = c * x;`, in the target C code there is only the statement `y = x;` owing to constant propagation. This trivial propagation invalidates the proof if the multiplication is replaced with an uninterpreted function. TVS, therefore, performs constant propagation before the abstraction phase.

4.1 Summary 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 [6,7]. 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.

The largest model that we verified automatically (with some minor manual modifications listed in a technical report [9]) 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.

References

1. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
2. Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, SRI international (2006)
3. Mathworks, T.: Simulink 7 user-guide, `http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf`
4. Mathworks, T.: Simulink getting started guide, `http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_gs.pdf`
5. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI 2000 (June 2000)
6. Pnueli, A., Siegel, M., Shtrichman, O.: Translation validation for synchronous languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 235–246. Springer, Heidelberg (1998)
7. Pnueli, A., Siegel, M., Shtrichman, O.: 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)
8. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. Technical report, SACRES and Dept. of Comp. Sci., Weizmann Institute (April 1997)
9. Ryabtsev, M., Strichman, O.: Translation validation: From Simulink to C (full version). Technical Report IE/IS-2009-01, Industrial Engineering, Technion (2009)