Proving Model Equivalence in Model Based Design

Ashlie B. Hocking, John Knight, M. Anthony Aiello
Dependable Computing LLC
Charlottesville, VA USA

Shin'ichi Shiraishi
Toyota InfoTechnology Center, U.S.A., Inc.
Mountain View, CA USA

***Abstract*—We introduce the concept of constrained equivalence of models in model-based development and present a proof technology for establishing constrained equivalence for models documented in MathWorks Simulink. We illustrate the approach using a simple model of an automobile anti-lock braking system.**

Keywords—Formal verification, model-based design, Simulink, equivalence proof

I. Introduction

In model-based software development, a model of a required software component is built, analyzed, and used to create an implementation of the software component. The model is developed as part of an overall systems engineering process in which the complete system is designed. The ability to modify a software component model easily, as part of systems engineering during refinement of the complete system design, is one of the major benefits of model-based software development.

Once a system design has reached the point that engineering moves to detailed design and manufacturing, the *development* model of the software component is refined into a *production* model. This refinement usually requires that the software component model be changed to accommodate differences such as limited precision data, target platform hardware characteristics, and more efficient algorithms. The refinement into the production model is often undertaken by different engineers from those who created the development model, frequently in different companies (a system developer and a supplier for example).

From the system perspective, preparation of the production model raises the following question: is the production model equivalent to the development model? Where different organizations are involved, answering this question is made more difficult by reluctance to share important engineering artifacts because they contain proprietary information.

Typically, equivalence of models is established by testing, leading to both high cost and doubt about the adequacy of the coverage of the test cases selected. In this paper, we introduce the concept of *constrained equivalence* in model-based development and present a *proof* technology for establishing constrained equivalence of models documented in MathWorks Simulink [1].

II. Constrained Equivalence

Development models rarely take account of the practical limitations forced by target platforms that must be addressed by production models. For example, development models might use 32-bit integers as a convenience (although not requiring that precision) where the target platform used for the production model only supports 16-bit integers. This difference means that the two models will not be identical. In general, such differences are common in engineering. Dealing with them by proof in model-based development is novel, to the best of our knowledge.

We have defined the concept of *constrained equivalence* for models. The definition of constrained equivalence is:

> **Constrained equivalence**: Two models exhibit constrained equivalence if either (a) all valid inputs to the first model produce the same output in the second model to within a specified tolerance, or (b) inputs to the second model that are within a specified tolerance of the inputs to the first model produce the same output. Predefined scaling factors and offsets might be used in determining whether two factors are the same.

In the following examples, n is an integer (perhaps from a production model) and x is a real number (perhaps from a development model). As an example of the first type of constrained equivalence, consider the following functions:

$$A(n) = \begin{cases} (n-32)\times 18, & n > 32 \\ (32-n)\times 18, & n \le 32 \end{cases}$$

$$B(x) = \begin{cases} (x-32)\times 1.8, & x > 32 \\ (32-x)\times 1.8 + 0.001, & x \le 32 \end{cases}$$

In this example, for all integers (i.e., the domain of A):

$$|A(n) - 10\times B(n)| \le 0.001$$

so we say there is a constrained equivalence between these two functions, where we assume that the scaling factor of 10 for the output of A is predefined.

As an example of the second kind of constrained equivalence, consider the following functions:

$$A(n) = \begin{cases} true, & n > 31416 \\ false, & n \le 31416 \end{cases}$$

$$B(x) = \begin{cases} true, & x > 3.14159265358979 \\ false, & x \le 3.14159265358979 \end{cases}$$

In this example, $A(n) = B(n \div 10000)$ when $n \neq 31416$, and $A(n) = B(n \div 10000 - 7.34641\text{E-}6)$ when $n = 31416$, so for all n there exists an m such that $\text{abs}(n \div 10000 - m) < 1\text{E-}5$ and $A(n) = B(m)$, where we assume that the scaling factor of 10,000 for the input to A is predefined.

III. Proof System

To prove constrained equivalence, the subject models have to be expressed in a language providing formal semantics. Once expressed in such a language, equivalence can be stated as a theorem and a proof attempted.

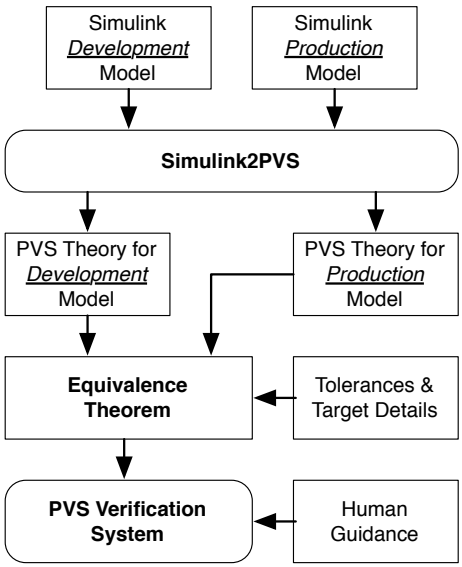

Figure 1: Constrained equivalence proof system

The overall structure of the verification system is shown in Figure 1. The proof system uses the PVS [2] language to define formally the semantics of Simulink [1]. PVS specifications of many of the Simulink blocks have been defined, and a translator, Simulink2PVS, has been built that translates a Simulink model into PVS theories.

The verification process begins by translating both models to PVS theories. These theories, which are formal specifications of the Simulink models, are then combined with a definition of constrained equivalence tailored to the models. The result is a theorem asserting constrained equivalence of the models. This theorem is submitted to the PVS verification system, and a proof of constrained equivalence is attempted, usually with human guidance in selecting the proof strategies.

Although the primary goal of developing the proof system was to allow proofs of constrained equivalence, the generation of PVS specifications of the models allows a wide variety of other analyses to be performed. All of the signals and data in the Simulink models are represented in PVS as PVS types; PVS type checking is immediately available. PVS is a functional language that requires that all functions be total. The PVS verification system generates *type correctness conditions* (small lemmas known as TCCs) that must be proved to ensure compliance with the type rules. The PVS type rules are meaningful in the original Simulink models, e.g., that the type of a numeric divisor does not include zero, and so ensuring compliance establishes useful properties of the models.

In addition to proofs of constrained equivalence and proofs of crucial type rules, other theorems can be stated and proved about a Simulink model, once the model has been translated to PVS. For example, a controller model for a microwave oven might be quite complex and describe control of the microwave source in detail, but proof of an important safety property, such as the source never operates while the door is open, is extremely useful and easily stated in PVS.

IV. Exploratory Study

A. Subject Models

To assess the feasibility of constrained equivalence proofs and to obtain a preliminary evaluation, we conducted an exploratory study using a model of an automobile anti-lock brake system (ABS) controller. The model we used, shown in Figure 2, is derived from an ABS model published by MathWorks [3]. This model serves as the development model in the study and relies upon a bang-bang controller that is published separately by MathWorks, shown in Figure 3. The ABS logic is only valid when the driver is depressing the brake pedal.

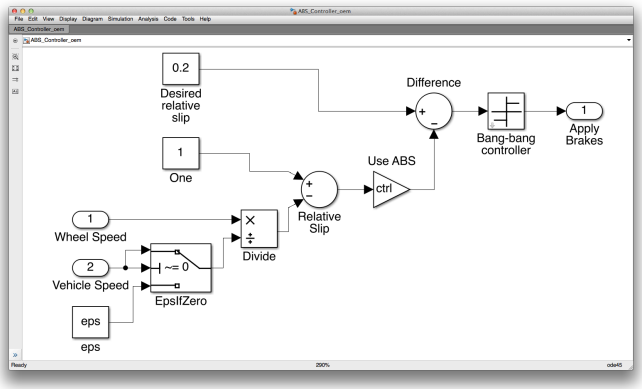

Figure 2: ABS controller development model

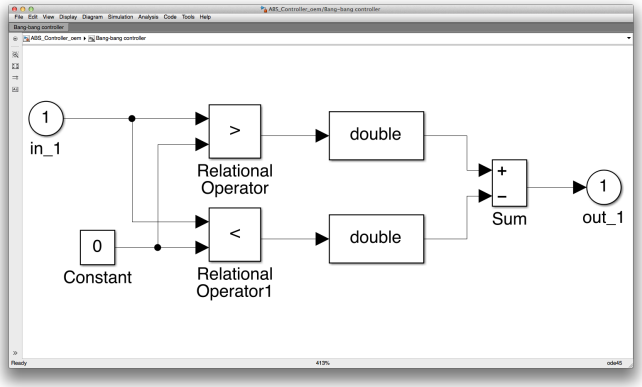

Figure 3: Bang-bang controller

The development model calculates the relative slip between the wheels and the vehicle. The definition of wheel slip is:

$$1 - \frac{Wheel\ Speed}{Vehicle\ Speed}$$

The system applies the brakes if the slip is less than 0.2 and releases the brakes if the slip exceeds 0.2. The data type specified in the model for wheel speed input is `Inherit: auto` which means that nothing is known about the actual type. Thus, in the development model, we have to assume that this input can be any real number.

An important element of the development model is the division block and the associated check for a zero divisor. The latter block is present to deal with the special case when the vehicle is at rest.

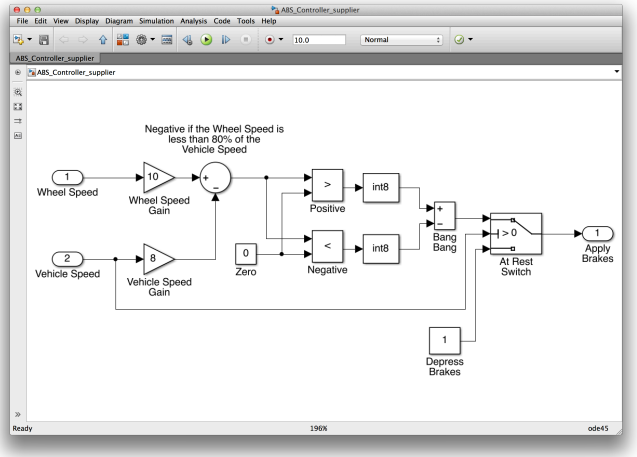

Figure 4: ABS controller production model

The production model, shown in Figure 4, was developed by the authors to investigate the reported proof technology and, although realistic, is purely *hypothetical*. The differences between the models are:

- There is no division in the production model, and therefore no need to protect against division by zero. Instead of the predicate:

$$0.2 - \left(1 - \frac{Wheel\ Speed}{Vehicle\ Speed}\right) > 0$$

or:

$$\frac{Wheel\ Speed}{Vehicle\ Speed} > 0.8$$

the production model uses the predicate:

$$10 \times Wheel\ Speed > 8 \times Vehicle\ Speed$$

These predicates are the same provided:

$$Vehicle\ Speed \neq 0$$

- The bang-bang controller has been replaced with its equivalent Simulink blocks. The assumption is that this controller would have been taken from a library for the development model but would be carefully customized for the production model.
- The production model has added logic for dealing with the case when the vehicle is at rest. In that case, the brakes should always be applied.

Visual comparison reveals immediately that the development and production models are quite "different" from each other — yet they should provide equivalent functionality.

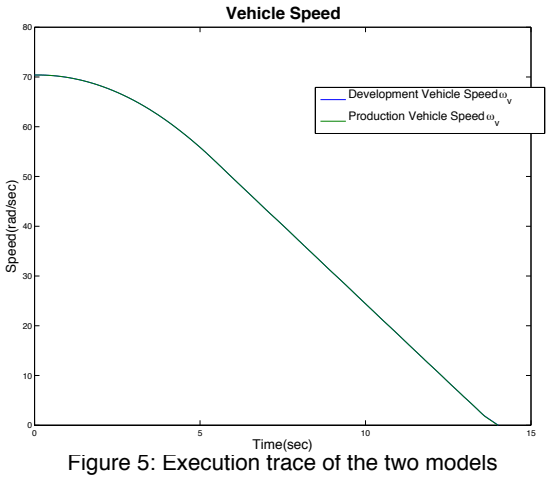

Figure 5: Execution trace of the two models

The functional difference between the two ABS controller models was examined using closed-loop simulation with a common vehicle model and is illustrated by the test outputs shown in Figure 5. This figure shows vehicle speed versus time produced by both controllers. While this test builds confidence, proof that the models are equivalent over *all* valid inputs is the goal of constrained equivalence.

B. *Developing The Constrained Equivalence Proof*

The steps to complete the proof of constrained equivalence are:

1. Translate both models to PVS theories using Simulink2PVS.
2. Create a predicate stating constrained equivalence between the two PVS specifications, including the application of necessary scaling. Quantification in the predicate is over the values of variables that the production model uses, i.e., integers or mappings to the integers.
3. Combine: (a) the PVS specifications of both models, and (b) the predicate of constrained equivalence.
4. Use the PVS verification system to attempt to prove the theorem of constrained equivalence.

Human guidance (but not human proof) might be required to decide which proof strategies PVS should attempt, if the proof system cannot complete the proof using its automated approaches.

C. *Proof for the Exploratory Study*

For the exploratory study, the predicate defining constrained equivalence between the models is:

```
Constrained_Equivalence: THEOREM
  FORALL (v_sys: ABS_Controller_production.sys_type,
            vSpeed: {i: nonneg_int32 | i <= 100000},
            wSpeed: {i: nonneg_int32 | i <= 100000}):
  f_Apply_Brakes(f_output(ABS_Controller_development.run
        (conv_sys(v_sys), vSpeed / 100, wSpeed / 100)))
                  =
  f_Apply_Brakes(f_output(ABS_Controller_production.run
        (v_sys, vSpeed, wSpeed)))
```

This predicate states that the application of the brakes by the two models is equivalent for vehicle and wheel speeds with integer values in the range 0 to 100,000. The divisions by 100 in the development model are scale factors necessary to align the speed measurement units. Integer values are meaningful, because the data supplied by speed sensors are discrete.

In attempting to prove this theorem, we used proof by cases using the following cases:

1. the car is moving and the wheels are not slipping;
2. the car is moving and the wheels are slipping; and
3. the car is at rest.

We were able to prove the first two cases, but for the third case we determined that the two models were actually different. To see the difference, consider the following two additional lemmas:

```
Applies_Brake_When_At_Rest_Production: LEMMA
   FORALL (v_sys: ABS_Controller_production.sys_type):
   f_Apply_Brakes(f_output(ABS_Controller_production.run
         (v_sys, 0, 0))) = 1


Releases_Brake_When_At_Rest_Development: LEMMA
   FORALL (v_sys: ABS_Controller_development.sys_type):
       f_Apply_Brakes(f_output(ABS_Controller_development.run
         (v_sys, 0, 0))) = -1
```

These two lemmas state properties about when the brakes will be applied by the system. In each of these two lemmas, the first zero in the parameter list corresponds to vehicle speed and the second zero corresponds to wheel speed. The first lemma shows that the *production* model applies the brake (i.e., has a return value of "1") in this case, while the *development* model releases the brake (i.e., has a return value of "-1"). In essence, proving these lemmas has identified a flaw in the *development* model, because the vehicle's brakes *should* be applied when the car is at rest and the wheels are not turning.

Because of this flaw in the development model, proving that these two models are equivalent for all valid inputs to the production model is impossible, and so instead we show that the two models are equivalent when the car is in motion:

```
Constrained_Equivalence_When_Moving: THEOREM
   FORALL (v_sys: ABS_Controller_production.sys_type,
         vSpeed: {i: pos_int32 | i <= 100000},
         wSpeed: {i: nonneg_int32 | i <= 100000}):
   f_Apply_Brakes(f_output(ABS_Controller_development.run
         (conv_sys(v_sys), vSpeed / 100, wSpeed / 100)))
                =
   f_Apply_Brakes(f_output(ABS_Controller_production.run
         (v_sys, vSpeed, wSpeed)))
```

This theorem is the same as the original constrained equivalence theorem, except that its inputs to the vehicle speed are only allowed to be positive integers, i.e., the case when the vehicle speed is zero is no longer considered. As this input range is covered by the first two cases, the proof by cases can be reused to prove this theorem.

V. Related Work

A variety of tools have been developed to check properties of models. The MathWorks Simulink Design verifier can perform static analysis of models including detection of dead code and code that could raise exceptions, and can also generate test cases.

Reactis is a test generation system for Simulink models [5]. SimCheck extends Simulink's type system to include annotations, dimensions, and units that can be associated with ports and links [6]. This extended type system is similar to the PVS type-system and can capture constraints that can be checked statically.

In addition to the types of analyses mentioned above, several tools have been developed to support demonstration of conformance with the ISO 26262 standard [7].

VI. Conclusion

In model-based development, different models of the same component are often used in development and production. Frequently, these models are built by different organizations. In such circumstances, establishing confidence that the production model of a software component is equivalent to the development model is a critical component of system verification. A proof of equivalence where equivalence is defined precisely can help to establish confidence without having to rely entirely on testing.

In the project reported in this paper, we have created a formal semantic model of part of MathWorks Simulink in PVS and implemented a translator from Simulink to PVS. In order to establish the equivalence of a development model and a production model, both models are translated into PVS theories and an equivalence theorem constructed. The theorem is proved using the PVS verification system.

We have illustrated this technique and demonstrated feasibility using a published model of an antilock breaking system.

Acknowledgments

We gratefully acknowledge the financial support for the research reported here from Toyota InfoTechnology Center, U.S.A., Inc.

References

[1] Simulink: Simulation and Model-Based Design, MathWorks, http://www.mathworks.com/products/simulink/

[2] SRI International, PVS Specification and Verification System, http://pvs.csl.sri.com/index.shtml

[3] MathWorks, Modeling an anti-lock braking system, http://www.mathworks.com/products/simulink/examples.html?file=/products/demos/shipping/simulink/sldemo_absbrake.html

[4] MathWorks, Simulink Design Verifier, 2012

[5] Reactive Systems, Inc., "Testing and validation of Simulink models with Reactis," November 2010.

[6] P. Roy and N. Shankar, "SimCheck: An expressive type system for Simulink," in NASA Formal Methods Symposium, 2010, pp. 149–160

[7] International Organization for Standardization, ISO 26262-1:2011, Road vehicles -- Functional safety