Visualizing properties of Simulink models

Gregory N. Anderson[*], *Ashlie B. Hocking*[†], *John C. Knight*[*†]

[*]*Department of Computer Science, University of Virginia, Charlottesville, VA, USA. {gna9zh, jck}@virginia.edu*
[†]*Dependable Computing LLC, Charlottesville, VA, USA. ben.hocking@dependablecomputing.com*

Keywords: Simulink verification, Simulink semantics

Abstract

Model-based development of software using tools such as MathWorks Simulink® has become common in the engineering of safety-critical systems. When working with Simulink, engineers need to be able to assure that the subject models possess crucial properties such as: (1) safety properties are met, (2) the use of measurement units is consistent, (3) exceptions will not be generated, (4) the execution sequences of blocks is as planned, and (5) data types are inferred properly. In this paper, we describe: (a) an approach to the graphic decoration of Simulink models to display information determined by formal verification, and (b) a system, SimulinkDec, that implements the decoration. Block colouring, enhancements to block textual descriptions, and comments are used to present the information of interest to the user. SimulinkDec provides a graphical user interface that allows engineers to easily perform formal verification and display the results of that analysis.

1 Introduction

Safety-critical systems are often developed using model-based tools, such as MathWorks Simulink®, in which models are created using a custom notation via a graphic interface. A model can be examined using simulation, executable code for the model can be synthesized, test cases can be generated for the model, and the model can be checked with a static analyser [1] [2].

A variety of techniques can be used in the verification phase of development using Simulink including testing, inspection, and formal verification. In previous work [3], we have described a system, Simulink2PVS, for the formal verification of properties in Simulink models. The system operates by first translating a model into PVS [4], and then establishing properties of the model by stating the properties as theorems and attempting to prove the theorems.

As is common in analysis tools, Simulink2PVS produces a traditional, textual result. Because Simulink is based on a graphical interface, using this interface to display the results of verification would be convenient and quite natural. The visual nature of the Simulink system can be used to allow the engineer to quickly and conveniently see which properties are satisfied and which are potential sources of error.

In this paper we describe a system for decorating the graphic representation of Simulink models to communicate information about the verification of a model to an engineer. In Section 2, we discuss the semantics of Simulink. In Section 3, we review several types of properties of interest in Simulink models. In Section 4, we describe how the system decorates Simulink models and provide examples. Finally, in Section 5 we present our conclusions.

2 Simulink Semantics

2.1 Analogue vs. Digital Semantics

In practice, Simulink has two different semantics, the *analogue* semantics and the *digital* semantics. The former are the semantics that are presented by many elements of Simulink in the original graphic notation. Integration, for example, can be included in a model, and the semantics of integration are those defined by mathematics. In terms of application domains that involve continuous time, such as control systems, this is a convenient approach.

In the end, however, in order to be used in a practical system, any model will be converted to computer software by some means. There is no digital software implementation of concepts such as integration, and so the software generated for a model will use an approximation, for example a summation for integration.

2.2 Formal Semantics

Formal verification of a notation such as Simulink requires that there be a formal specification of the semantics of the notation. In order to support our verification goals with Simulink2PVS, we have developed a formal semantic definition of much of the Simulink notation, including a large number of the functional blocks and the order of execution of elements of a model [3]. We note that, to the best of our knowledge, no formal specification of Simulink semantics has been published by the manufacturer.

This semantic definition is of the digital semantics, i.e., the form of the model that will execute on a digital computer. With the formal semantics, we have a model in logic of each portion of Simulink that allows us to describe formally what a model will actually do when run. That we have defined the digital semantics is intentional, because we are interested in proving properties of models that hold for the system that will run on the intended target computer platform.

The importance of specifying semantics formally can be seen when one realizes that there are a few Simulink blocks that do not have an intuitive or agreed-upon behaviour in some circumstances, for example, the *Saturation Dynamic* block shown in Fig. 1.

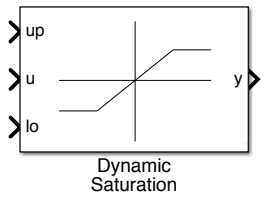

Fig. 1. Simulink *Saturation Dynamic* block

The *Saturation Dynamic* block takes three inputs. Under normal circumstances, the block is defined by the function:

$$y = \begin{cases} u & lo \le u \le up \\ lo & u < lo \\ up & u > up \end{cases}$$

However, Simulink does not guarantee that $up \ge lo$, and if $up < lo$, there is no information describing the expected behaviour of this block.

The formal semantic definition provided by Simulink2PVS of the *Saturation Dynamic* block defined in PVS is shown in Fig. 2. An important feature of this definition is that the type definition for lo includes a pre-condition that the value of lo is less than the value of up. A lemma known as a *type-correctness condition* will be created where this block is used, and proving that the type-correctness condition holds will ensure that the block is used appropriately.

```
DynamicSaturate(up: real, u: real, lo: {v: real | v <= up}): real =
  IF u > up THEN
    up
  ELSE
    IF u < lo THEN
      lo
    ELSE
      u
    ENDIF
  ENDIF;
```

Fig. 2. Formal semantic definition of the dynamic saturation block

As well as missing or counterintuitive block semantics, Simulink blocks have a number of parameters that modify their behaviour and complicate their semantic definitions. For example, the data type of a block's output does not have to be explicitly defined, and can instead be inherited from its inputs. In particular, some blocks allow their output data type to be set to *Inherit: Inherit via internal rule*, where the internal rule is specific to each individual block. Semantically, this means each block might behave entirely differently from all others in terms of data types, and that the data type of an input will depend on the context of the block.

2.3 Units and Dimensions

The Simulink notation did not include unit information for variables in models until recently. Version 8.7 (R2016a) added limited support for units, with the *Inport*, *Outport*, and *Signal Specification* blocks being the only pre-existing blocks that support units, and with two new blocks (*Unit System Configuration* and *Unit Conversion*) added to help. Necessarily, unit analysis is limited to subsystem interfaces.

In view of the importance of unit consistency in models describing physical systems, Simulink2PVS supports a comprehensive and general units system. A simple annotation mechanism is used to indicate within the text of a block the units of relevant variables. All of the associated units' consistency rules are defined within the predefined theorems that Simulink2PVS provides. An example of a Simulink block with unit annotations is shown in Fig. 3.

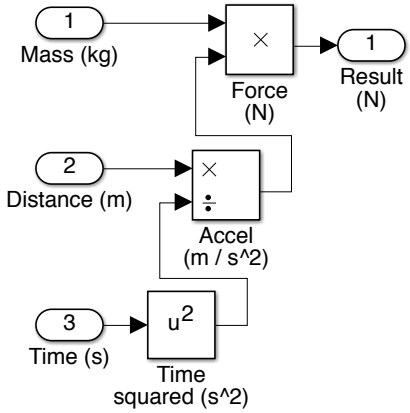

Fig. 3. A Simulink block with units annotations

The current version of Simulink2PVS (Version 3.0) can be used to prove that all the blocks in a system conform to the physical axioms associated with units. As a consequence of having the unit annotations, Simulink2PVS also proves the axioms associated with physical dimensions, for example that distance divided by time is a velocity.

3 Properties of Interest

In general, developers using Simulink are interested in many properties of their models. Their correct understanding of the semantics of the model are included in the list. The current properties that Simulink2PVS attempts to prove or identify are:

- **Safety properties**.
  Frequently, applications have important properties related to safety that derive from the requirements rather than being requirements themselves. For example, in a cruise control system, the system should never be engaged when the vehicle is under braking. Such properties should always hold.
- **Unit and dimensional consistency**.
  The use of measurement units should be consistent, and the physical dimensions in expressions should be consistent. For example, angles measured in degrees should not be mixed in expressions with angles measured in radians unless explicit conversions are performed.
- **Freedom from exceptions**.
  Exceptions, such as integer overflow, divide by zero, and array index out of bounds should never arise unexpectedly and be unhandled.
- **Planned block execution sequence**.
  Simulink's graphical notation suggests concurrency in

block execution. Nevertheless, in the digital semantics of Simulink, blocks are executed in a prescribed order. This order should match the user's expected order.

- **Block data type inheritance**.
  Simulink blocks allow their output data type to be set to *Inherit: Inherit via internal rule* (and other types of *Inherit* rules), where the internal rule is dependent on context and potentially specific to each individual block. The data type inheritance should match the user's expectations. To the best of our knowledge, explicit rules for data type inheritance are not published by MathWorks.

Simulink2PVS analyses these properties in different ways. Explicit properties (such as safety properties) are encoded as PVS theorems, and implicit properties (such as correct use of measurement units and freedom from exceptions) are encoded as PVS type-correctness conditions that must be proven to hold.

4 Decorating Simulink

4.1 Concepts

Our approach to the decoration of Simulink models takes advantage of the graphical interface that is used to design models. A number of properties defined in this interface can be used to convey information to the engineer, including:

- **Colour**.
  Simulink makes little use of colour in the notation by which models are defined. Despite this, colours can be set for a number of items including the foreground and fill colours of blocks and the line colour of signals.
- **Font**.
  As with colour, Simulink makes little use of available fonts. The size, weight, style, and actual font used in displayed text can be set to a wide variety of values.
- **Time**.
  The presentation of Simulink models during development is static. Various forms of simple motion can be introduced on different language elements.
- **Additional material**.
  Any elements of the Simulink notation can be added to a model as can user interface elements such as dialog boxes.

Combinations of the above display properties achieve decoration of Simulink models to indicate the diagnostic information in which we are interested.

An important design aspect of the approach to decoration is that the decoration is applied to the subject model within the Simulink system itself. Thus, the user only needs to deal with one primary interface. Model development and analysis take place through this common graphic display. The decorating system merely adds a small control panel in a separate window.

4.2 System Structure

The structure of the system that implements the Simulink decorating scheme is in Fig. 4. The system includes MathWorks Simulink, Simulink2PVS, the PVS theorem prover, and custom software, SimulinkDec, that integrates all the various parts.

PVS relies upon a sophisticated type system, and many defects in a model yields type-correctness conditions (TCCs), i.e., conditions implied by the type system, that could not be discharged. These unproven TCCs are traced back to the block that generated them in order to identify the element in the subject model that has to be decorated.

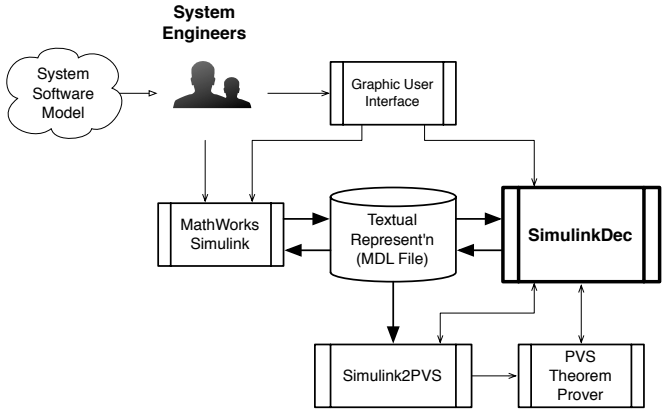

Fig. 4. Overall structure of the Simulink decoration system

SimulinkDec acts as a controller, first calling Simulink2PVS to generate a semantic definition of the subject model. The PVS specification is then combined with several theorems that define correctness properties (such as unit matching and freedom from division by zero) within the PVS type system. Finally, SimulinkDec invokes the PVS theorem prover to prove as many of theorems and type-correctness conditions as possible.

SimulinkDec uses its own simple interface to communicate with the user. The interface allows the user to select a model to be analysed, specify the types of analyses required, and set decoration parameters. After the decoration is added, the user can return the model to its original state through the interface by clearing the decorations.

4.3 Safety Properties

Safety properties of interest are application specific and are included in Simulink models as *Assertion* blocks [2]. Simulink2PVS recognizes *Assertion* blocks in models and translates them into PVS theorems. To indicate that an *Assertion* block should be interpreted as a safety property, the block's name includes the word Consequent.

Theorems that cannot be proved indicate a possible problem. Failure to prove a theorem could be because the theorem is false or because of the use of ineffective proof strategies. The automatic approach used by SimulinkDec cannot distinguish these two cases, and so the user is alerted to the situation in either case.

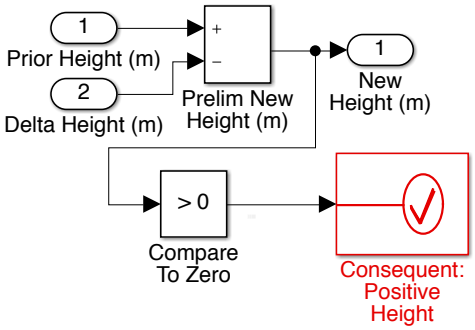

Fig. 5. Calculation for new height of water with failed property

An example is shown in Fig. 5. This example comes from a control system designed to manage the height of water in a supply tank used for cooling. The safety property of interest is that the height of the water not drop to zero, and this is included in the model as an assertion. Based on analysis of the complete model, Simulink2PVS cannot prove that this safety property will be true, and so the *Assertion* block is decorated by colouring it red.

4.4 Units and Dimensions

In order to take advantage of unit and dimensional analysis, Simulink2PVS requires that every block in a model include appropriate unit information.

Any block that has no unit information has its name set in bold font and increased font size. An example is shown in Fig. 6.

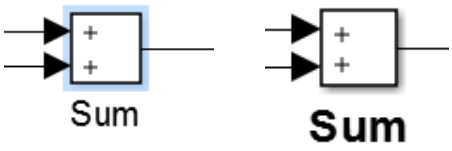

Fig. 6. Decoration of a Simulink block without unit annotation

Normally, unit information is embedded within block names. For example, the block shown in Fig. 6 might be named Sum (m) to indicate that the values being added represent meters.

Any blocks with mismatched units are decorated by colouring the foreground of the offending block red. The foreground is coloured rather than the background because the issue is a discrepancy between the two blocks linked by the coloured wire rather than inherent to the input port. Because the properties generated by Simulink2PVS generally deal with the relationship between blocks, we chose a colouring scheme that highlights the relationship between the blocks rather than the blocks themselves. With foreground colouring, the name of the block causing the issue can be highlighted so the engineer can find it immediately.

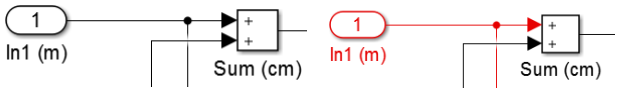

Fig. 7. *Inport* block In1 coloured red to indicate unit incompatibility with the Sum block

An example is shown in Fig. 7 with the original model fragment on the left and the decorated fragment on the right. The foreground of the *Inport* block is coloured to indicate a mismatch in units between it and the *Sum* block.

4.5 Unhandled Exceptions

Numerous computations can result in exceptions in code generated from Simulink, including integer overflow, divide by zero, and array index out of bounds. As an example, consider the Accel (m / s^2) block in Fig. 3. If the value to the *Inport* Time (s) is zero, then the division operation will have a divide-by-zero error. In the absence of a pre-condition on the *Inport* block (i.e., having its *Minimum* value set to a non-zero value such as 0.001), or special handling for the zero-condition, SimulinkDec will decorate the foreground colour of the Time squared (s^2) block with red, as shown in Fig. 8.

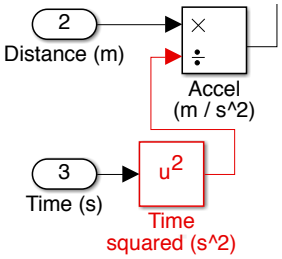

Fig. 8. Identification of potential divide-by-zero error

4.6 Block Execution Sequence

Decorating blocks one at a time in their execution order is used to indicate block execution sequences. The decoration consists of colouring the block and changing the font used for the block name to bold.

An example is shown in Fig. 9 where the first block in the sequence is decorated. Stepping through the rest of the process reveals that the complete sequence is Product, Sqrt, Sum, Sum1, and then Out1. By separating the colouring of the blocks temporally, we indicate the sequential separation of computations.

The sequence of block execution is defined by our semantic definition. Specifically, this semantic definition is outlined in pseudocode as:

```
if (block1.priority < block2.priority) then
    block1 executes before block2
else if (block2.priority < block1.priority) then
    block2 executes before block1
else if (block1.feedsInto(block2) and not
  block2.feedsInto(block1)) then
    block1 executes before block2
else if (block2.feedsInto(block1) and not
  block1.feedsInto(block2)) then
    block1 executes before block2
else if (block1.name < block2.name) then
    block1 executes before block2
else
    block2 executes before block1
end if
```

Informally, execution order is determined first by block *Priority* (an optional parameter), then by data dependency, and then by lexicographical ordering (block names are required to be unique in Simulink). For example, in Fig. 9, the Sum1 block depends on Product, Sum, and Sqrt, so it executes after these blocks (no priorities are set). The Product and Sum blocks do not depend on each other, so Product executes before Sum, due to lexicographical ordering. Similarly, Sum and Sqrt do not depend on each other, so Sqrt executes before Sum.

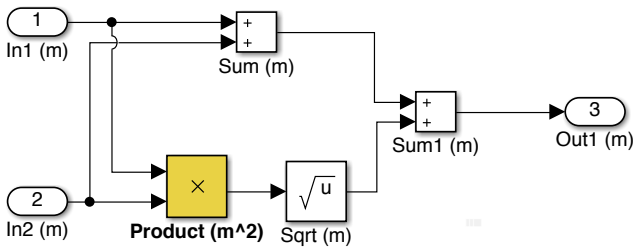

Fig. 9. Decoration indicating that the Product block is the first non-input block to be executed.

4.7 Inherited Output Type

Any block that inherits the data type of its output rather than defining the data type explicitly is decorated by colouring its foreground yellow as shown in Fig. 10.

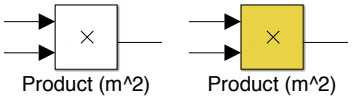

Fig. 10. Block with *Inherit: Inherit via internal rule* output data type before and after indicative decoration.

As noted in Section 2.2, inherited data types require assumptions to be made based on the context of the blocks, including how large a result might be, which removes the ability to verify that the output does not overflow the declared type, as the type is *assumed* to be large enough to hold the output. Thus, the system warns the engineer of any blocks that do not explicitly define their output types.

4.8 Semantic Traps

In Simulink, the semantics of a particular block can be influenced by multiple parameters, and some of these parameters are confusing or ambiguous. As an example, consider the *Integer rounding mode*, which takes one of seven possible values: *Ceiling*, *Floor*, *Zero*, *Convergent*, *Nearest*, *Round*, and *Simplest*. The supplied documentation defines the first six of these rounding modes clearly, but determining from simple examination which rounding modes are used on which blocks is not easy. Furthermore, the *Simplest* rounding mode is ambiguous, since it depends on "[generating] code that is as efficient as possible" [5].

To assist the engineer in dealing with this circumstance, SimulnkDec provides an option for decorating blocks with annotations indicating their rounding method. An example is shown in Fig. 11.

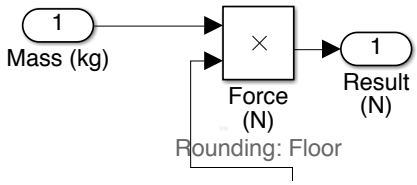

Fig. 11. Force block annotated with its *Integer rounding mode*.

The difficulty with the *Saturation Dynamic* block noted in Section 2.2 is also a semantic trap with Simulink. This trap is tackled initially within Simulink2PVS. The formal semantics for the block allow Simulink2PVS to try to prove the precondition included in the semantics. If the proof is successful, the trap is avoided. If the proof fails, the user's attention is brought to the block by the associated TCC failure decoration.

5 Conclusions and Future Work

In this paper, we have presented a system for the decoration of Simulink models with information gained from a formal verification system. The verification system defines a formal semantics for Simulink that is used to allow proofs of a wide variety of properties of models. It also supports a simple annotation mechanism that allows physical units to be specified generally and unit consistency checked.

These capabilities yield diagnostics about potential defects in models that are made available to the user via decorations of the subject Simulink model. The decoration system also informs the user about various aspects of the semantics of Simulink that might be unfamiliar.

In future work, we plan to extend the decoration system in a number of ways. Decoration is being applied to other semantic traps, including potentially confusing or ambiguous parameters, as they are discovered. We are also enhancing the types of decoration used to display block execution sequences. We plan to add analysis of floating-point computations together with decoration to show the user the extent of rounding error. Finally, we plan on creating an open-source repository containing our PVS definition of the Simulink semantics

so that interested parties[1] can contribute to an agreed-upon understanding of the semantics.

Acknowledgment

This research was sponsored in part by Toyota InfoTechnology Center, Mountain View, CA.

References

[1] MathWorks, "Simulink Design Verifier," 2016. [Online]. Available: http://www.mathworks.com/products/sldesignverifier/. [Accessed 15 June 2016].

[2] A. B. Hocking, M. A. Aiello, J. C. Knight and N. Aréchiga, "Proving Critical Properties of Simulink Models," 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE), pp. 189-196, January 2016.

[3] A. B. Hocking, J. C. Knight, M. A. Aiello and S. Shiraishi, "Formal Verification in Model Based Development," Society of Automotive Engineers, 2015.

[4] SRI International, "PVS Specification and Verification System," [Online]. Available: http://pvs.csl.sri.com.

[5] The MathWorks, Inc., "Rounding Modes for Fixed-Point Simulink Blocks," 2016. [Online]. Available: http://www.mathworks.com/help/fixedpoint/ug/rounding-modes-for-fixed-point-simulink-blocks.html. [Accessed 16 Jun 2016].

[6] A. B. Hocking, M. A. Aiello and J. C. Knight, "Static Analysis of Physical Properties in Simulink Models," in 26th IEEE International Symposium on Software Reliability Engineering, Gaithersburg, MD, 2015.

[1] Other companies developing static analysis tools and code generators for Simulink have already shown an interest in this idea.