Platform-Independent Debugging of Physical Interaction and Signal Flow Models

Mehdi Dadfarnia
Engineering Laboratory
National Intitute of Standards & Technology
Gaithersburg, USA
mehdi.dadfarnia@nist.gov

Raphael Barbau
Engisis, LLC
Bethesda, USA
raphael.barbau@nist.gov

***Abstract*—Systems engineering tools are used to organize development activities of a wide variety of engineers, many of whom develop discipline-specific simulation models. To increase the efficiency of this process, systems modeling tools have been extended to represent physical interactions and signal flows that can be translated to simulation tools and executed. Sometimes these simulation models fail to execute or they produce unexpected execution results. It is helpful to identify causes of these problems in earlier stages of system model development, before they propagate to fully-developed simulation models. Debugging physical interaction models is difficult because their execution is bidirectional between system components. This paper gives an overview of debugging procedures for physical interactions and (unidirectional) signal flows in platform-independent system models that integrate with domain-specific simulation models. These procedures identify system model causes of simulation execution failure or incorrect simulation results.**

Keywords—SysML, debugging, modeling, simulation, lumped parameter, equation-based languages

I. Introduction

The increasing complexity of modern engineered systems and products requires integrating systems modeling and simulation tools to improve efficiency of design processes [1]. These tools capture behavioral and structural aspects of systems or products that are checked (analyzed) without prototyping and experimenting on real systems [2]. Systems engineers use systems modeling tools to organize and coordinate analysis by a wide variety of engineers [3], many of whom develop their own equation-based simulation models [4].

Some simulations tools present graphical interfaces showing interconnection of components, corresponding to energy and information exchange between physical objects in the real system [4, 5]. They are referred to as *physical interaction and signal flow* models (also known as lumped parameter, one-dimensional, or network models) [6]. These models help manage system complexity by focusing on *what* systems should accomplish, rather than *how* [5]. Simulations of these models are experiments that answer questions about the systems being modeled without physically building them [4, 5].

Many simulation tools incorporate equation-based modeling languages for physical interaction and signal flow [7, 8, 9, 10]. System component specifications include ordinary and algebraic differential equations, while their interconnections generate additional equations between components. These models can represent a wide range of discipline-specific physical interactions between components (mechanical, electrical, etc.) as well as communication of numeric and Boolean information.

Systems modeling tools also focus on how components are interconnected and broken down into subcomponents (system structure) [11]. Interconnections between components reflect physical interactions and information exchanges. This organization enables systems engineers to coordinate others in specialized engineering disciplines, focusing on subsets of components and interconnections that require their expertise [12]. However, unlike simulation tools, system models are not strictly equation-based.

System models are often developed separately from simulation, leading to inconsistent specifications of overlapping aspects of the system [6]. To minimize these inconsistencies and enhance model interoperability, we developed a publicly-available extension to the Systems Modeling Language (SysML) [13] (the most widely used graphical modeling language for systems engineering) that facilitates platform-independent integration of SysML with physical interaction and signal flow simulation tools (SysML Extension for Physical Interaction and Signal Flow Simulation, SysPhS) [6, 14]. We developed software to translate these extended SysML models into simulation files that run on two widely-used simulation platforms [15].

Despite the increased ease of use, efficiency, and integration provided by the extension and translator, it is often hard to identify (debug) the cause of errors in the models. Determining the cause of errors is a critical step in correcting models. This paper is concerned with identifying causes of failure to:

- Execute simulation models translated from system models,
- Get expected results from simulation execution.

The debugging procedures presented[1] focus on system models written in SysML that are translated to simulation, rather than on particular kinds of simulation models. The procedures are independent of simulation platform (language or tool), aiming to fix errors in system models before they spread to simulation models on multiple platforms. Failures of translation from system models to simulation due to incorrect usage of the SysPhS extension or errors in the translator are not considered.

Integrating SysML models of physical interactions and signal flows with simulation tools enables compilation, simulation, and validation of the SysML model. System models can be checked for failure to compile and simulate a translated model or failure to generate expected results from

The work of Raphael Barbau was supported by U.S. National Institute of Standards and Technology grant award 70NANB14H249 to Engisis, LLC.

[1] Author Raphael Barbau developed these techniques.

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

the simulation run-time. Although the causes of these two kinds of errors might be similar, debugging techniques in system models differs from those used in equation-based modeling languages.

Section II categorizes errors and debugging techniques for each kind of error. It also surveys literature on debugging techniques for system models and simulation tools for equation-based modeling. It describes SysML and SysPhS features that are important to debugging physical interactions and signal flows, using examples from a cruise control system. Section III gives an overview of the proposed debugging procedures. Section IV concludes with a discussion of our findings, and an outlook for future work.

II. Background

A. *Types of Errors and Debugging Techniques in Physical Interaction and Signal Flow Models*

Several types of errors can cause failures in physical interaction and signal flow models in systems and equation-based modeling languages. This paper focuses on identifying errors that cause simulation to fail or generate unexpected results during execution. The type of failure influences the debugging procedures required.

Errors that cause failure to simulate arise from model structure. These show the modeler's design does not properly support simulation. The underlying system of equations is inconsistent, including overconstrained (more equations than variables) or underconstrained (fewer equations than variables). It also includes equations that would divide by zero, functions being called outside of their real domain (such as the square root of a negative number), and other erroneous symbolic transformations.

Errors that cause the simulation to produce unintended results arise from the meaning of the model. These reflect discrepancies between desired system behavior and simulation execution. Although some errors can be identified automatically (such as variable values outside bounds) depending on the simulation tool being used, these errors can also be found manually after trying to validate the simulation results. These errors can come from incorrect equations, incorrect parameter or initialization values, and incorrect function calls from equations. Errors can also be due to integration errors in the equation solvers being used [1], which will not be discussed in this paper.

The difficulty in debugging these errors in physical interactions is that observing ordered execution of command sequences or operations do not work in models that include bidirectional relationships. This fundamental difference means that debuggers for errors in physical interactions need to examine chains of variable transformations in the model.

Static debugging techniques identify errors that cause failure to compile simulation models to executable code. These techniques trace symbolic transformations through the model to identify erroneous sections [1]. Dynamic debugging techniques identify errors that cause simulation to produce unexpected results. These techniques involve interactive inspection of models during execution [1]. They must be used after static debugging techniques ensure the model can be compiled to executable code.

B. *State-of-the-art Debugging Techniques for Physical Interaction and Signal Flow Models*

Though methods for identifying errors in system specifications share many similarities [5, 16], models that include physical interaction require specialized debugging strategies because these interactions affect all components involved. This is in contrast to sequential execution of operations in most other modeling languages, including signal flow simulation, where each component only affects the ones executing after it. Since models of physical interaction always have two participants we call it bidirectional (sometimes known as "acausal" even though causality applies to all physics), while models of signal flow are unidirectional.

Debugging in bidirectional and unidirectional modeling requires different approaches [5]. In unidirectional models, an error found at some point in the execution implies that the cause of the error is in one of the components or operations executed prior to that point. Errors in bidirectional models do not come from a past sequence of executed operations or components, because they are not executed in sequence.

Identifying errors in bidirectional (physical interaction) models has received significant attention in the Modelica language community [17]. In [1, 18, 19, 20, 21], authors look at debugging models in equation-based languages, as well as scalability of such techniques for larger models. Earlier work focused on code instrumentation to provide traditional debugging mechanisms such as breakpoints and single-stepping [20], which are more useful for sequential languages. There is also work focused on determining whether a system of equations is balanced [18], as well as techniques to semi-automatically isolate data flow slices to find potential sources of failure [19].

In [1, 22, 23], authors integrate information from variable (symbolic) transformations from static debugging into a dynamic debugger at simulation run-time (transformations are mathematical operations on variables to give values to other variables). The debuggers have graphical interfaces for exploring variable simulation and traditional debugging techniques such as breakpoints and single-stepping. Techniques for tracing symbolic transformation are critical to debugging physical interaction and signal flow models, including those translated from system models.

Implementing these debugging techniques in system models is more difficult than in simulation because of the higher level of abstraction preferred in early stages of systems engineering processes. Unlike equation-based modeling languages, system models specify multi-disciplinary, conceptual models of a system and its components. Tracing symbolic transformations requires a more diagram-based procedure in these models. The authors in [24] describe the structure of a functional, system model debugger that integrates system models with an equation-based modeling language through a mapping between them. The debugger focuses on visualizing variables running through a simulation, and much less on symbolic transformations. This paper argues that some tracing of symbolic transformations is necessary in system models.

It is burdensome to coordinate changes between system models and equation-based simulation models. One way to coordinate changes is to fix errors in simulation tools and feed the corrections back into system models. Another way is

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

to debug system models before generating and experimenting with simulation models. This paper focuses on the latter approach. Although both are useful, applying equation-based model debugging techniques to system models at earlier development stages can verify and increase understanding of the relationships captured in system models before discipline-specific experts focus on parts of the systems in their own models and tools. It also helps fix errors in functional, higher abstraction, platform-independent system models before they spread to behavioral, lower abstraction, domain-specific simulation models.

Next, we give some background information about modeling physical interactions and signal flows in SysML that is essential for tracing symbolic transformations. In the subsequent section, we describe the the debugging techniques applied to SysML system models.

C. *Modeling Physical Interactions and Signal Flows with SysPhS*

The SysML extension and simulation translator we developed [14, 15] integrates SysML with physical interaction and signal flow modeling simulation by identifying modeling capabilities in common between simulation platforms, comparing those with SysML, and extending SysML with only those modeling capabilities that SysML does not have [6]. Development starts with system models in extended SysML, then translates them into simulation platforms. This means that any errors not due to usage of the extension, translator, or simulator's execution engine will be in the SysML model.

Static debugging requires tracing through variable transformations in the model, and dynamic debugging uses bookkeeping of values over simulated time to pinpoint errors. Tracing physical interactions and signal flows in SysML system models extended with SysPhS [14] is done through connectors. The role of connectors in modeling physical interactions and signal flows is briefly discussed in this section; more can be found in [6, 12, 14].

Modeling in SysML starts with system components and their interactions in an internal block diagram (IBD), as in Figure 1. It shows physical interaction and signal flows involved in automatic control of a vehicle's speed, between the vehicle, its components, the cruise controller, and the operating environment. Interactions are represented by SysML connectors, which show exchanges of physical substances or signals occurring between the ends of each connector. The ends are either parts, or ports (smaller rectangles appearing on rectangles) that are placed on parts, on connector properties, or on other ports. Part names appear in titles of the rectangles in IBDs, before the colons. Each part is a role in the model (such as *driver* in Figure 1) and is played (typed) by a kind of thing (such as *Person*), appearing in the title after a colon and represented by a SysML block. Ports are essentially parts of parts, playing roles of roles in a model. They are also typed by blocks to show the kind of thing playing each role.

Item flows on connectors are optional, but useful to represent the type of signal or conserved physical substance flowing between parts or ports. They appear in Figure 1 as labels of filled triangles on connector lines, such as the *LinearMomentum* label between the parts *gravVehicleLink* and *controlledVehicle*. One filled triangle on a connector indicates signal flow in the direction of the triangle is pointing, while two pointing in opposite directions indicate physical interaction.

Blocks that type parts or ports at the ends of connectors must have flow properties typed by the kind of signals or physical substances flowing. Flow properties or

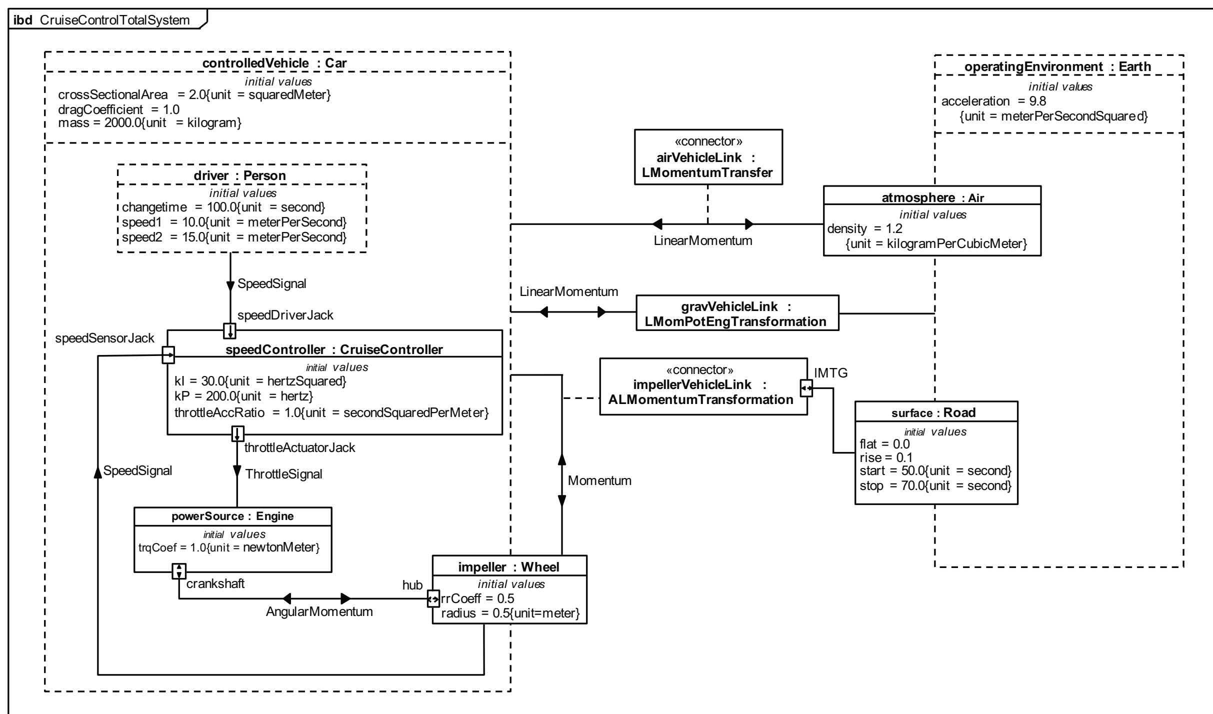


Fig. 1. Internal Block Diagram for a vehicle cruise control system, defined in SysML extended by SysPhS for physical interaction and signal flow

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

their types represent variables (either conserved or non-conserved, see below). Flow properties for signal flows are *in* or *out* (unidirectional) variables, typed by the kind of signal flowing. Flow properties for physical interactions are *inout* (bidirectional), typed by the physical substances flowing. These substances have variables for the substance's flow rate and potential to flow (one variable for each). Blocks that type parts or ports at the ends of connectors can be selected from libraries of reusable component interaction blocks or be specified by modelers. They can include more properties besides the flow properties, as well as multiple flow properties if the part or port typed by that block is involved with multiple types of signal flows or physical interactions.

The SysPhS extension includes a *PhSConstant* stereotype to specify that property values are to remain constant during each simulation execution. It also includes a *PhSVariable* stereotype to specify that they might vary during simulation. Flow properties for physical interaction are typed by blocks from a SysPhS library that have properties with *PhSVariable* stereotypes applied. Flow properties for signal flows have *PhSVariable* stereotypes applied as well. More details on defining blocks and their properties can be found in the SysPhS specification [14].

Connectors also imply a mathematical relationship between variables of flow properties defined by the blocks typing parts or ports at each end of a connector. After connectors are translated to simulation, those tools generate equations for them (differently for physical interaction and signal flow, see below). Parts and ports may also perform mathematical manipulations on flow property variables, specified by the blocks that type them, using SysML's parametric diagrams. These diagrams equate (bind) variables of their equations to flow properties and other properties from their containing blocks. Equations in these diagrams are called constraints and their variables are called constraint parameters.

In physical interactions, when connected ports and parts are typed by blocks that have the same *inout* flow property (and no connector properties are involved, see below), the values of conserved variables (flow rates of conserved substances) for the same flow property must add to zero during simulation, while their paired non-conserved variables (potential to flow) must be the same. Connectors can be augmented by a property of the containing block that represents physical substances transferred between parts or ports in the system and not at the boundary of any object. A connector property can also represent transformation of one type of physical substance – and equivalently one set of flow property variables – to another. The transfers and transformations represented by connector properties are specified with equations in parametric diagrams. More about connector properties can be found in [6, 12, 14].

In signal flows, when connected parts and ports are typed by blocks with the same flow property but opposite directions (one *in*, the other *out*), the flow properties (which are non-conserved variables in this case) must have the same value during simulation. Multiple connectors cannot have the same *in*-flow property at their ends because signals flowing into it would conflict.

Lastly, connectors linking to a port or part with no flow properties, such as the one between parts *gravVehicleLink* and *operatingEnvironment* in Figure 1, are structural relationships that enable physical interaction, but across which not physical substances flow. In Figure 1 these are the connectors directly to the earth and road, which are necessary for the car to store/consume potential energy and to propel itself, respectively, but are assuming to be immovable themselves.

III. Debugging Methods

The SysPhS extension enables translation from system models to equation-based models. If an equation-based model fails to compile or simulate correctly, the cause can be identified by tracing through chains of connectors between components. This is the basis for static debugging techniques and facilitates dynamic debugging. Before overviewing the techniques, we discuss a procedure to simplify system models by distinguishing between physical interaction and signal flow connectors. Simplifying models beforehand makes debugging more straightforward and scalable.

A. Preprocessing: Simplifying Models

The system model is broken down into one for physical interactions and another for signal flows. This enables separate debugging of two simpler system models before replicating the resulting fixes in the complete model, a simpler task than debugging the entire model all at once.

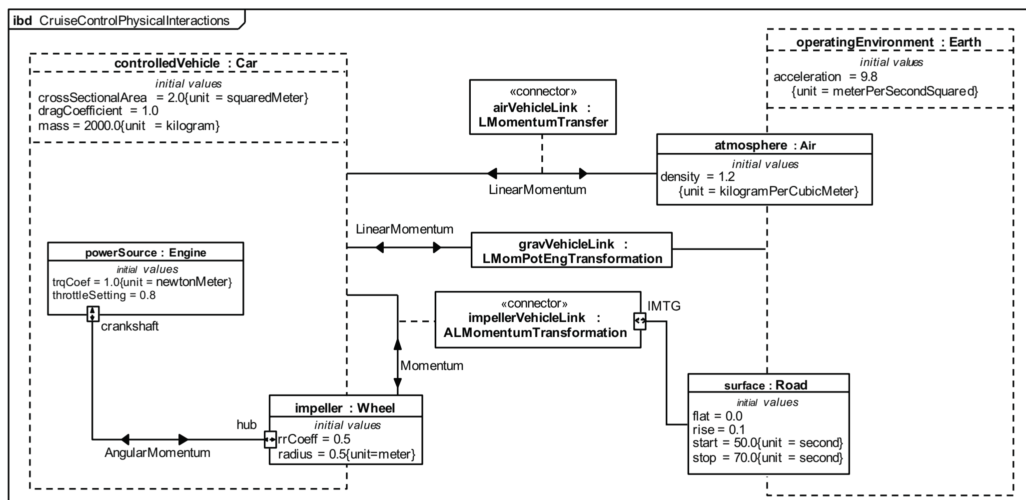


Fig. 2. Cruise control model with only physical interactions

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

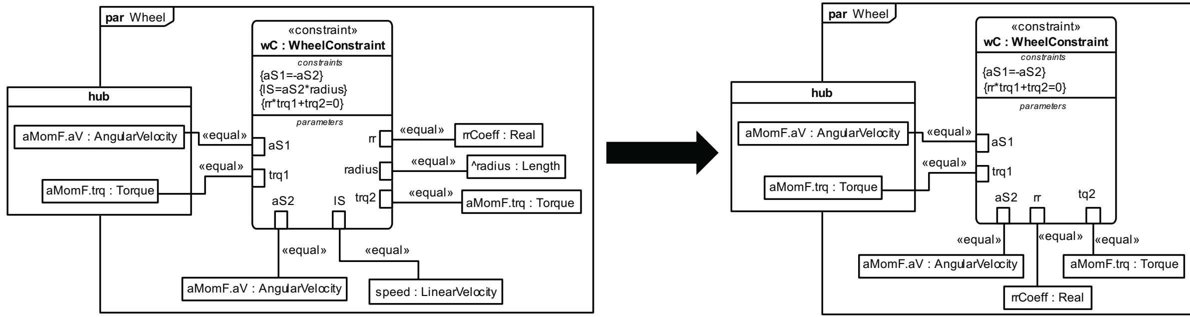


Fig. 3. Show two (2) parametric diagrams of the same system (before and after changes)

First, a model of physical interactions in the system is derived by eliminating all connectors in the original model's IBDs that do not represent physical interactions (saving a separate copy of the original model first). Any remaining parts or ports that are not at the end of the remaining connectors or do not possess a port that is at the end of a remaining connector are also eliminated. Following the example in Figure 1, Figure 2 presents an IBD with only the physical interactions in the original cruise control system.

Next, in each parametric diagram for the remaining parts or ports, eliminate equations (constraints) determining values of variables (constraint parameters) that are bound to (signal flow) *out*-flow properties. Eliminate part or port properties that are bound to variables on these *out*-flow equations as well. Replace any remaining equation variables bound to *in*-flow properties on the parts or ports by constant values, either by directly replacing the parameter with a constant value in constraints or by introducing a binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see [14] for value assignment examples). Figure 3 depicts a parametric diagram for a component in Figure 2, before and after these changes were made.

A separate system model for signal flows is derived by first eliminating all connectors in the original model's IBDs that do not represent signal flows (while saving a separate copy of the original model). Also eliminate any remaining parts or ports that are not at the end of the remaining connectors or does not possess a port that is at the end of a connector.

Next, in each parametric diagram for the remaining parts or ports, eliminate equations that play no role in determining values of variables bound to *out*-flow properties or equations that do not have any bindings to *in*-flow properties are eliminated. Part or port properties not bound to variables on the remaining equations. Of the remaining equations, some variables might be bound to physical interaction *inout*-flow properties on the parts or ports. These flow properties are replaced in simplification. If any equation variable bound to these flow properties determine the value of a variable bound to an *out*-flow property, then eliminate the *inout*-flow property and give a new constant value to its variable by binding to a *PhSConstant*-stereotyped property that has a constant default value or instance value (see [14] for value assignment examples). If any equation variable bound to these flow properties is determined by a variable in the same equation that is bound to an *in*-flow property, then eliminate the *inout*-flow property and give its variable a new binding to a new property with a *PhSVariable* (see [14] on applying variable- and constant-value stereotypes to properties in SysPhS).

The remaining sections present the debugging techniques. Static techniques find the cause of failures to compile and simulate translated models. This type of failure prevents generating a simulation run-time from the translated model. Once compilation succeeds, dynamic debugging techniques identify causes of failure to produce intended simulation behavior. The underlying theme for static debugging is tracing symbolic transformations in the model to find errors. Transformation tracing is also useful for dynamic debugging to better understand the model and sources of potential simulation-related errors.

B. Static Debugging for Failure to Execute Simulation

The failure of an equation-based model (translated from a system model) to compile and simulate in a simulation tool indicates a static error. These errors can be identified with static debugging techniques applied to the system model, which trace chains of symbolic transformations in the model. These transformations appear as mathematical relationships in constraint equations or implied by connectors. Specifically, tracing refers to tracking transformations of known and unknown variables through a model. Known variables are properties whose values are assigned a constant value or determined through mathematical relationships. Tracing is complemented by bookkeeping, which records the known or unknown status of these variables when operations apply to them in the model.

Static debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. For models with physical interactions, the first task is to identify the part, port, or connector property in IBDs where physical interaction will first occur or initiate other physical interactions in the system. Multiple parts and ports where physical interactions simultaneously occur can initiate further interactions, but any one can be arbitrarily picked to begin tracing. Tracing and bookkeeping of mathematical transformations start with properties associated to this selected part or port. Deciding which system component commences the physical interactions is easy in many cases. For example, the initiators of flow of electric charge in an electric circuit are the voltage sources or current sources. In the cruise control system represented in IBDs in Figures 1 and 2, the throttle in the engine physically initiates the car's interaction with the road and air (this happens on command from the driver, but the command is signal flow, not physical interaction).

When the initiator of physical interaction is not obvious, it can help to inspect the parametric diagrams of parts or ports in IBDs. Parametric diagrams contain bindings between

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

properties of the parts (or ports) to variables in the part's constraint equations. Look for parametric diagrams of parts that have a higher number of *PhSConstant*-stereotyped properties (with values given explicitly in the model) than *PhSVariable*-stereotyped properties (with values determined by mathematical relationships in the model), except for *PhSVariables* that give simulation time. The equation variables (constraint parameters) bound to *PhSConstant* or time properties are used in the part's equations (constraints) to determine values of other variables, which are bound to other properties used in the part's equations. To find an initiator, search for a part or port where most of its properties or properties of its ports are bound to constants or time values in its parametric diagram. The only properties without constant or time values should be flow properties, which can only have their values determined through connectors. Parts or ports initiating physical interactions have the fewest of these flow properties.

Tracing bindings and constraints in parametric diagrams helps understand and keep track of (bookkeep) which variables in the equations are known and unknown. Constraint equations show mathematical transformations between known variables, bound to properties with known values, and unknown variables, bound to properties with unknown values. Before simulation, the only known variables are the ones bound to *PhSConstant* properties, the variables bound to properties given (initial) values at the start of simulation, and properties that give simulation time values. These should lead to values assigned to all variables in the parametrics diagram of physical interaction-initiating parts. The status of these variables will change as tracing shows their values being assigned through constraints or connectors, which is recorded by bookkeeping.

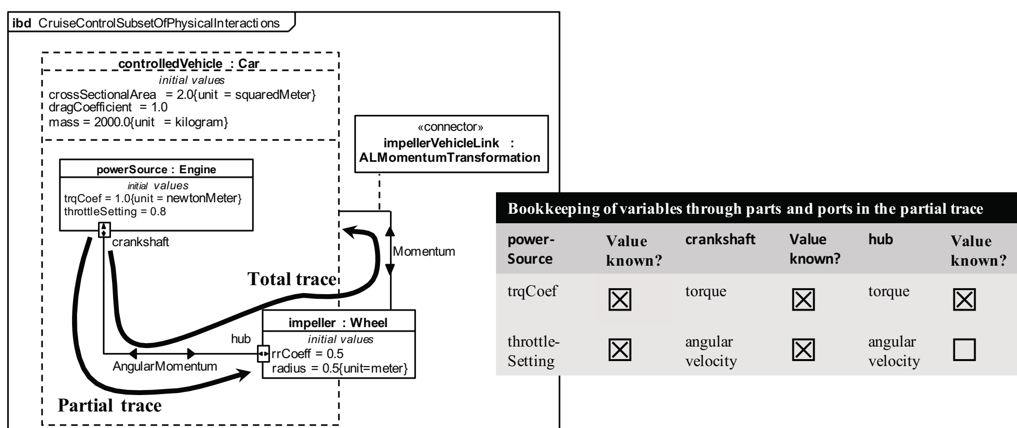
Physical interaction flow properties on the current part in the debugging process link to flow properties on parts or ports at the other end of the linking connectors. Trace along these connectors to find out whether values are assigned to these flow properties leads to parametric diagrams of other parts, ports, and connector properties linked to the current part. Repeat the same methods of tracing and bookkeeping in these other parametric diagrams to determine whether values are assigned to unknown variables and to find flow properties that lead to new connectors and parametric diagrams. The trace must go through all connectors and parametric diagrams of the system's parts, ports, and connector properties. Figure 4 shows an example of tracing and bookkeeping value assignments between a physical interaction-initiating part and another part. Bookkeeping of the total trace would complete the tracking of value assignments.

A system model will compile and simulate when translated if it a) uses all the constraint equations and connectors in the model for mathematical transformations between known and unknown variables and b) has all its property values determined by simulation of mathematical transformations. If tracing and bookkeeping identifies a constraint equation or connector that is not used, the system is overconstrained. In this scenario, the modeler must choose whether unused equations or connectors should be removed or a new property should be included and related to them. If an unknown property is not defined by any mathematical constraint or connector, then the system is underconstrained. In this scenario, the modeler must choose between using this property in a new equation or eliminating the property. Tracing and bookkeeping of equations also helps spot constraint equations that involve a division by zero and functions called outside their domains. Once corrections to the model are made, they are replicated in the original system model.

If there is a complementary model of signal flows, repeat the process of tracing and bookkeeping in a similar fashion, but start tracing from all parts that do not have *in*-flow properties or do not own ports that have *in*-flow properties. The *in*-flow property on these parts indicate that they receive unidirectional signals from another part in the model, so they cannot be the initiator of signal flows. Corrections in this model should likewise be reproduced in the original, complete model of the system. Translate the corrected SysML model and test on simulation platforms to determine if more debugging is needed.

C. Dynamic Debugging for Unexpected Simulation Results

Failure of an equation-based simulation model (translated from a system model) to produce expected results when executed indicates a dynamic error. The simulation model is able to compile and simulate, but produces variable values that deviate from modeler expectations. These errors can be identified with dynamic debugging techniques applied to



Bookkeeping of variables through parts and ports in the partial trace

power-Source	Value known?	crankshaft	Value known?	hub	Value known?
trqCoef	☒	torque	☒	torque	☒
throttle-Setting	☒	angular velocity	☒	angular velocity	☐

Fig.4. Shows initiating physical interaction component, direction of traces, bookkeeping of variables, and value assignment that occur through the partial trace (for brevity)

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

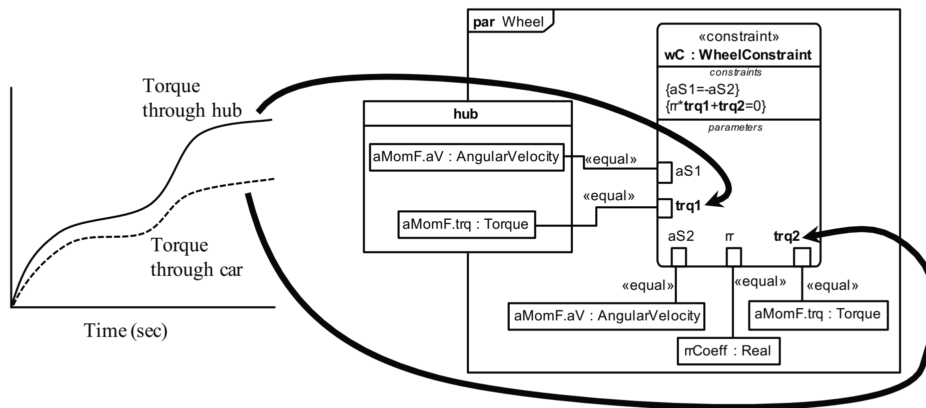


Fig. 5. Relationship between simulation variables and flow properties in the system IBD

the system model. These techniques examine executed simulations to understand exactly when signals and conserved substances flow through the system and what their characteristics are. They focus on simulation results for variables involved in the static traces of flow properties linked by connectors in the previous section. This showed how variables characterizing flow of physical substances and signals during simulation are related via transformations in the system model (mathematical operations via constraint equations and connectors). Though dynamic debugging can be performed without prior static debugging, fixing static errors first ensures the simulation model will compile and execute, and static tracing improves understanding of how variables change during simulation.

Dynamic debugging can be performed on complete system models, but is described here on simplified, complementary models of a system's physical interactions and signal flows. Behavior of conserved substances in physical interactions is characterized by their flow rate and potential to flow. Flow rate and potential to flow appear in simulation as variables translated from properties at the ends of connectors in the system model. This enables modelers to track simulation variables that correspond to properties in SysML system models. The SysPhS translator uses the names of association ends and constraint parameters in the resulting simulation models to facilitate this, but tracking simulation variables might require some familiarity with the equation-based simulation language. Lastly, like static debugging, dynamic debugging starts by tracing simulation variable transformations at points in the model that initiate physical interactions in the rest of the model. These points must be identified before debugging.

Physical interaction variables simulate flow of conserved substances only at their corresponding connector endpoint (part or port) in the system model. A more complete picture of symbolic transformations of these variables is seen by observing their values over simulated time and comparing them to other physical interaction simulation variables in the model. Graphical displays in simulation tools show these values, enabling comparison of simulated values to their intended mathematical relationships. The relationships are defined, correctly or not, through transformations (mathematical relationships between variables derived from connectors and parametric diagrams in the system model) of corresponding flow properties in the system model. To visualize these transformations, observe variables when their corresponding flow properties have not undergone more than one set of transformations (operations that occur on flow properties in the constraints of one parametric diagram or in the mathematical relationship implied by one connector). Compare simulation values of these variables with those of other physical interaction variables related to the same part or port in the system model, as well as simulation variables related to the other end of the variables' associated connectors.

Analysis of simulation variable results is performed in simulation runs that are sufficiently long for their values to reach a steady-state or a recognizable pattern of changes. Check that changes follow the mathematical transformations specified in corresponding constraint equations and connector links in the system model, which can be modified to produce better results. Figure 5 shows the relationship between simulated variable values over time and flow properties in the parametrics diagram and IBD.

Further simplification of system models can determine whether simulation results are valid, especially when physical interactions are highly complex. One way is to temporarily remove parts, ports, and connectors until modelers have high confidence in what they expect from variable behavior. Once this core model produces correct simulations, the removed parts and ports can be incrementally restored, simulated, and checked [16].

Validity of simulation results might also be determined by reaching consensus among modelers, users, and stakeholders on whether the simulation model is producing the correct results [25]. These techniques are out of the scope of this paper, but they include qualitative and quantitative model exploration, and comparison of simulation results to system behavior or alternative validated simulation results [25].

If there exists a complementary model of signal flows, repeat the process of inspecting simulation variables in a similar fashion. However, start tracing with all parts that do not have *in*-flow properties or do not own ports that have *in*-flow properties, as chosen during static debugging. Replace remaining parts in a complementary model of signal flows that only have *out*-flow properties or only have ports with *out*-flow properties have their flow properties by *PhSConstant*-stereotyped properties with pre-specified values before debugging.

Errors that are found by debugging are corrected in the system model, then tested by translating to simulation models and executing them. Translating and testing system models to multiple simulation platforms is more robust, because fixes

978-1-5386-8396-5/19/$31.00 ©2019 IEEE

sometimes work for one simulation platform and not others. For example, a function call in a parametric diagram is domain-specific, and this might need to be replaced with a more universal function call. It is also possible that some modeling capabilities in SysML, such as state machines or different ways of defining initial values, cannot be replicated on some simulation platforms (see [14] for more specific examples about translation differences between simulation platforms).

IV. Conclusions & Future work

This paper presents an overview of debugging procedures for physical interaction and signal flow models translated from system models to equation-based simulation languages. The procedures identify errors causing compilation and simulation of these models to fail, or to produce incorrect simulation results. The integration of system models with equation-based models facilitates interoperability between developers of these types of models. This is done in SysML extended by SysPhS [6, 12, 14, 15]. The debugging procedures for platform-independent system models help identify problems without debugging and correcting the domain-specific simulation model and then transferring those changes back into the system model. The procedures are categorized as static and dynamic. Static debugging traces symbolic transformations in the system model, and dynamic debugging uses results of simulations to check changes in variable values during simulation. These debugging procedures are performed on system models and intend to complement existing debugging techniques on simulation platforms.

The authors plan to improve the debugging processes with user-friendly interfaces to visualize aspects of model translation, particularly the mapping between components in system models (e.g. equations, parts, properties) and structures in simulation models resulting from translation. The interface can provide information to both system and simulation modelers about the names, locations, and number of times translated structures and variables appear in the simulation model. This facilitates communication between systems engineers and simulation tool experts, without being concerned about details in one another's models.

Acknowledgments

The authors thank Conrad Bock for helpful discussions on the contents of this paper. Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the U.S. National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

References

[1] A. Pop, M. Sjölund, A. Asghar, P. Fritzson, and F. Casella, "Integrated Debugging of Modelica Models," *Modeling, Identification and Control,* vol. 35, no. 2, pp. 93-107, 2014.

[2] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*, Morgan Kaufmann Publishers, 2014.

[3] A. L. Ramos, J. Vasconcelos Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1, pp. 101-111, 2012.

[4] P. Fritzson, *Introduction to modeling and simulation of technical and physical systems with Modelica*, John Wiley & Sons, 2011.

[5] S. Van Mierlo, "A multi-paradigm modelling approach for engineering model debugging environments," Ph.D. dissertation, Department of Mathematics and Computer Science, University of Antwerp, 2018.

[6] C. Bock, R. Barbau, I. Matei, and M. Dadfarnia, "An Extension of the Systems Modeling Language for Physical Interaction and Signal Flow Simulation," *Systems Engineering* vol. 20, no. 5, pp. 395-431, 2017.

[7] Controllab Products B.V., *Getting Started with 20-Sim 4.6*, 2017. Accessed on: Oct. 16, 2018. [Online]. Available: http:// www.20sim.com/downloads/files/20simGettingStarted46.pdf

[8] F.E. Cellier, *Continuous System Modeling*, Springer Science & Business Media, 1991.

[9] The MathWorks, Inc., *Simscape Language Guide*, *Version 4.5*, September 2018. Accessed on October 18, 2018. [Online]. Available: https://www.mathworks.com/help/pdf_doc/physmod/simscape/simscape_lang.pdf

[10] Open Source Modelica Consortium, *OpenModelica User's Guide*, February 18, 2016. Accessed on October 18, 2018. [Online]. Available: https://www.openmodelica.org/doc/OpenModelicaUsersGuide/OpenModelicaUsersGuide-v1.9.4-dev.beta2.pdf

[11] *IEEE Standard for Application and Management of the Systems Engineering Process,* IEEE 1220-2005, 2005.

[12] M. Dadfarnia, C. Bock, and R. Barbau, "An Improved Method of Physical Interaction and Signal Flow Modeling for Systems Engineering," in *14th Annual Conference on Systems Engineering Research (CSER),* Huntsville USA, March 2016.

[13] Object Management Group, *OMG Systems Modeling Language Specification, version 1.5*, May 2017. Accessed on October 19, 2018. [Online]. Available: http://www.omg.org/spec/SysML/1.5

[14] Object Management Group, *SysML Extension for Physical Interaction and Signal Flow Simulation Specification, Version 1.0,* July 2018. Accessed on October 2, 2018. [Online]. Available: https://www.omg.org/spec/SysPhS/1.0/

[15] R. Barbau, *SysPhS 1.0 OMG Release*, May 4, 2018. GitHub repository, accessed on November 2018. [Online]. Available: https://github.com/usnistgov/saismo/releases/tag/sysphs

[16] D. Krahl, "Debugging simulation models," in *37th Conference on Winter Simulation*, Olrando USA, December 2005, pp. 62-68.

[17] Modelica Association, *Modelica-A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.3, Revision 1*, July 2014. Accessed on October 18, 2018. [Online]. Availabe: https://www.modelica.org/documents/ ModelicaSpec33Revision1.pdf

[18] P. Bunus, and P. Fritzson, "A debugging scheme for declarative equation based modeling languages," in *International Symposium on Practical Aspects of Declarative Languages*, Portland USA, January 2002, pp. 280-298.

[19] P. Bunus, and P. Fritzson, "Semi-automatic fault localization and behavior verification for physical system simulation models," in *18th IEEE International Conference on Automated Software Engineering,* Montreal Canada, October 2003, pp. 253-258.

[20] A. Pop, and P. Fritzson, "A portable debugger for algorithmic modelica code," in *4th International Modelica Conference,* Hamburg Germany, March 2005, pp. 435-443.

[21] A. Asghar, A. Pop, M. Sjölund, and P. Fritzson, "Efficient Debugging of Large Algorithmic Modelica Applications," *IFAC Proceedings Volumes* vol. 45, no. 2, pp. 1087-1090, 2012.

[22] M. Sjölund, F. Casella, A. Pop, A. Asghar, P. Fritzson, W. Braun, L. Ochel, and B. Bachmann, "Integrated Debugging of Equation-Based Models," in *10th International Modelica Conference,* Lund Sweden, March 2014, pp. 195-204.

[23] M. Sjölund, and P. Fritzson, "Debugging symbolic transformations in equation systems," in *4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools,* Zurich, Switzerland, September 2011, pp. 67-74.

[24] A. Canedo, and L. Shen, "Functional Debugging of Equation-Based Languages," in *5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools,* Nottinham UK, April 2013, pp. 55-64.

[25] R. G. Sargent, D. M. Goldsman, and T. Yaacoub, "A tutorial on the operational validation of simulation models," in *Winter Simulation Conference (WSC),* Washington D.C. USA, December 2016, pp. 163-177.

978-1-5386-8396-5/19/$31.00 ©2019 IEEE