Signature Required: Making Simulink Data Flow and Interfaces Explicit

Marc Bender[1], Karen Laurin[1], Mark Lawford[1], Jeff Ong[1], Steven Postma[1] and Vera Pantelic[1]
[1]*Department of Computing and Software, McMaster University, Hamilton, ON, Canada*
{*bendermm, laurink, lawford, ongj2, postmasm, pantelv*}*@mcmaster.ca*

Keywords: Simulink, interfaces, model transformation, refactoring, software engineering, software, data flow

Abstract: Model comprehension and effective use and reuse of complex subsystems are problems currently encountered in the automotive industry. To address these problems we present a technique for extracting, presenting and making use of signatures for Simulink subsystems. The signature of a subsystem is defined to be a generalization of its interface, including the subsystem's explicit ports, locally defined and inherited data stores, and scoped gotos/froms. We argue that the use of signatures has significant benefits for model comprehension and subsystem testing, and show how the incorporation of signatures into existing Simulink models is practical and useful by discussing various usage scenarios.

1 INTRODUCTION

Model-based development using visual programming languages has become a commonly used method for the development of embedded software. In particular, Simulink has been widely adopted for the development of control software in the automotive industry. While the use of model-based development has many advantages, which have been discussed at length in the literature (Schatz et al., 2002; Rau, 2000; Rau, 2002), many of the visual languages currently used for embedded software development lack some of the traditional software engineering features developers have come to expect and depend on.

It has become an accepted view in software engineering that system development requires modularization and information hiding (Rau, 2001; Meyer, 1992; Parnas, 1972) to allow for division of tasks among developers, as well as ease of maintainability, comprehensibility, verifiability, and reuse of modules. The focus of this paper is to bring the basic self-documentation components of traditional programming languages to Simulink. A traditional imperative programming language such as C uses function prototypes, variable declarations, and other such mechanisms to aid in the understanding and maintainability of the code. Importantly, the strict variants of the languages require that these mechanisms all be defined in specific parts of the code. Traditionally, in C-like languages the interface to a module has been defined in a header file. Simulink does not have any conventions that can be drawn upon as a parallel to the mechanism of module interface declarations in C header files that aid developers' understanding. This is a weakness of some visual programming languages, and Simulink in particular, which we will discuss at length in this paper.

We will focus on using the Simulink subsystem as the closest analogue to a module, but we will add structure to what is required in a subsystem in order to provide a more complete understanding of the subsystem to a developer. This leads to the question, what comprises a complete understanding of the interface to a subsystem in Simulink? We feel that the interface of a subsystem in Simulink comes down to the data flow into and out of the given subsystem, as in visual languages the data flow is an important component to understanding the purpose of the system.

In practice, we have found that it can be difficult to identify data flow in Simulink. The simple approach of connecting blocks using signal lines works for simple models, but as models grow in complexity, this becomes inadequate and difficult to maintain. Simulink includes other mechanisms such as from/goto pairs and data store memory blocks, which allows the passing of data without direct connection between. Also complicating large models is the fact they contain significant hierarchies of subsystems. Data flow using only input and output ports becomes inadequate for multi-level hierarchies, thus Simulink provides cross-hierarchical data flow using data store memory blocks and from and goto blocks that can be accessed at different levels, depending on the scope defined. As we have not found in literature a comprehensive analysis

of data flow in Simulink, we provide a brief summary of Simulink data flow in Section 2.

Using the Simulink mechanisms that are available to aid the developers with the flow of data without using directly connected signals presents challenges to understanding, navigating, documenting and maintaining production-scale Simulink models. Upon opening an arbitrary subsystem, it can be very difficult to determine its expected context and behaviour. There is no approach that has been widely adopted for discovering or presenting a subsystem's context. In this paper we present an approach which addresses this problem. We introduce the notation of signatures for subsystems in Simulink, which is an embedded presentation of the interface and the context of the subsystem. Our proposed signature provides the following main features:

- a data flow legend for each subsystem to ease comprehension
- the signature can be automatically extracted from existing models, and automatically updated as required
- detaches the interface from the subsystem, thus separating its internal behaviour from its external manifestation.

Our efforts are motivated by the issues we have found with data flow in visual languages when modeling large complex system, and the lack of attention that has been paid to these issues in the literature. There have been studies done that compare the use of a visual programming language to a textual programming language (Cox et al., 2004; Green and Petre, 1992). The results of (Cox et al., 2004) show how presenting developers with both control and data flow information can aid in the comprehension of Boolean expressions from code fragments. However, the study performed by (Green and Petre, 1992) discusses the fact that visual programming languages are not in fact easier to read than textual programming languages, due to the fact that it is harder to simply scan a visual program, the way one would scan a code fragment. This conclusion supports our argument for the need for a subsystem signature to aid the developers in data flow comprehension within Simulink.

Similar work has been proposed in (Rau, 2002). In that paper Rau proposes a pattern for strong interfaces in Simulink in order to improve typing for inputs and outputs. In order to achieve this interface, Rau proposes that developers follow a particular design pattern for subsystems. The differences between the potential use of typing in our proposed signatures for Simulink subsystems and typing in Rau's strong interfaces are discussed in Section 4.

The remainder of the paper is structured as follows. Section 2 presents a careful analysis of Simulink data flow constructs and their behaviours. Section 3 offers a formal definition of abstract signatures, along with a discussion of their properties and variants. Section 4 is devoted to using signatures in practice, providing a concrete application of signatures (i.e., used in Simulink models) and discussing their uses and benefits. Finally in Section 5 we present ongoing and future work and the conclusion is Section 6.

2 DATA FLOW IN SIMULINK

In this section we present our analysis and criticisms of data flow in Simulink. Simulink is used for Model-Based Development and is integrated in Matlab. For the purposes of the analysis performed, Matlab version 7.13 (2011b) and Simulink version 7.8 are used, however, this analysis should also apply to the most recent versions of Matlab and Simulink (2013b).

In order to model a large complex system, the ability to decompose a system into subsystems is required to make the system comprehensible, maintainable and allow multiple developers to simultaneously work on different parts of system. Simulink allows a system to be embedded in another system, effectively creating a hierarchy of subsystems. Blocks in Simulink represent these embedded subsystems as well as built-in basic functions performed by the system. The blocks are connected by signals, which represent the data.

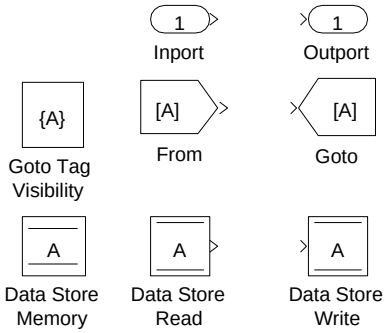

Figure 1: Additional Simulink data flow mechanisms

In a simple model, following these signals one can easily understand the system data flow. However, as models become more complex, it becomes much harder to follow the connected signals due to the introduction of subsystems, ports, froms and gotos, and data stores (See Figure 1). We now present some background information on these mechanisms. It is important to note that for simplicity we are only using

virtual subsystems (MathWorks, 2013) in our discussion of data flow analysis of Simulink. Non-virtual subsystems have exceptions as to when these constructs can be used and are therefore out of the scope of this paper. Also, for simplicity of presentation, we will not be discussing the Simulink Bus.

Subsystems create a hierarchy within the system. The result is that not all of the relevant information for a particular subsystem is readily available on that level in the hierarchy. Necessary information, for example signal types or the source of a signal, may come from further up in the hierarchy. In Simulink, in and out ports are used to show the incoming and outgoing data for a subsystem. Inports show the information that is being fed into the subsystem from outside and outports show what information the subsystem is passing back to the subsystem(s) that it is contained within or the surrounding environment.

From and goto blocks allow for connections to be made without using a directly connected signal. Information passed into a goto block is then propagated on to all corresponding from blocks. A single goto block may have multiple from blocks, but a from block may only receive data from a single goto block. The user can set the scope of the goto block in the block's parameters under tag visibility. The permitted scope of the goto blocks are:

- *Local* - The from and goto blocks are used within the same subsystem. These are identified by square brackets around the block name.
- *Global* - The from and goto blocks can be used anywhere in the mode hierarchy. These are identified by curly braces around the block name.
- *Scoped* - The from and goto blocks have limited visibility. In order to define the scope of a goto tag, a block called a goto tag visibility block must be used, which is of the same name as the corresponding from and goto blocks. The from and goto blocks may be used from within that subsystem and any subsystem lower in the model hierarchy.

Data stores are used in Simulink as memory. Through data store read and data store write blocks of the same name as the data store memory block, a developer can access this memory. The data store mechanism enables the transfer of data without having directly connected signals in the system. It also allows for multiple levels of a subsystem to use the same memory location without needing to pass it through ports. The scope of the data store is defined by the location of the data store memory block. The subsystem in which the data store memory block is defined and any subsystem below it in the hierarchy may access the data store through reads and writes. A global data store is defined as a signal object in the Matlab workspace, which allows the data store to be accessed by all models in the workspace. Global data stores are identified by the keyword "global" appearing in the block. There also may be multiple read and write blocks for a single data store memory block. This introduces issues with the order of access to a data store. There are three defined order of access errors associated with data stores:

- Read-Before-Write - A read occurs on the time step before a write has occurred, which introduces latency issues, as the model is reading stale data
- Write-After-Read - A write occurs after a read has already occurred on the time step, which can introduce issues as to whether the read has obtained the correct value required for execution
- Write-After-Write - A write occurs twice on the same time step with no read, which can introduce issues as data is lost

Without explicitly defining the order of executing of data store read and write blocks, there is the potential for the order of execution to change in different releases. (MathWorks, 2008) recommends following ways to handle the order in which data store read and writes are executed in a system.

- Use function call subsystems to be able to control the order the subsystems are executed in
- Set priorities in embedded atomic subsystems or model blocks
- Utilize diagnostics at compile and run time to detect order of access issues, data stores used in multiple tasks, and multiple data stores using the same name
- Use strong typing, which is inherited by the read and write blocks, to ensure there is no unexpected use of the data store.

While these Simulink dataflow mechanisms may be introduced into the model to reduce the number of blocks and signals that are visible for a given (sub)system with the goal of aiding comprehensibility and maintainability, they also introduce issues in understanding the data flow of a complex model. In the remainder of this section we will outline the issues we have encountered while working on large industrial models.

It is possible in Simulink to override a goto tag visibility or data store memory, by defining a new goto tag visibility or data store memory of the same name in a lower level subsystem. Then any access to the data, whether it be by a from or goto block for the goto tag or a read or write block for a data store memory, will be different depending on the level in the system

hierarchy. This can lead to errors or unexpected behavior if the developer was unaware of the multiple definitions in multiple levels of the hierarchy. Being able to differentiate within the subsystem where the definition of the scoped mechanism occurs, either at the current subsystem level or in a higher subsystem, could be of value to the developer, especially if they do not have to spend time searching for this information.

For a subsystem, we define the explicit interface as those items that are clearly dealing with the flow of information in or out of the given subsystem. Therefore the *explicit interface* for a system consists of all inports, outports and data flow mechanisms that are contained within embedded subsystems. The implicit interface are those mechanisms used to reduce the number of connected signals. The *implicit interface* contains the from and goto blocks (of all visibility types) and the data stores defined globally or defined in the subsystems higher in the hierarchy than the current subsystem that is being viewed. The *imposed interface* contains the data stores and goto tag whose visibility are defined within the current subsystem. The imposed interface is the definition of the data stores and gotos within the subsystem, but the actual use of these mechanisms may occur at a subsystem lower in the hierarchy.

When viewing a subsystem that is within a large complex system, the explicit interface is unclear, due to the fact declarations can be on multiple levels in a system hierarchy and are not all visible in one place. The numbered inport and outport blocks are inadequate for the developer to know where the data is coming from or going to on first glance. Another issue with ports is they can only be used (connected) once directly before requiring other mechanisms required to allow for signal branching. When it becomes necessary to branch the signal, we have found from the automotive industry code we are working with, developers will commonly feed the inport into a locally defined goto and use from blocks where the data is needed (we have incorporated this idea into our *concrete* application of signatures; see Section 4.2).

As with the explicit interface, the implicit and imposed interfaces are also unclear when viewing the given subsystem. The information for data that is defined globally or within a certain scope must be searched for within the entire system, and the Simulink 'find' function is not always adequate to preform this task. For a developer who is new to a given system, understanding the data flow can be a difficult and time consuming task. However, with a mechanism that can help guide the developer in understanding the data flow can make the task of understanding the subsystem quicker and easier. In the remainder of the paper we will present such a mechanism, the *signature* for Simulink subsystems.

3 SIGNATURES

We begin by presenting an abstract formal definition of signatures. A subsystem signature is, essentially, just a representation of the interface of a Simulink subsystem. Thus, a signature comprises a set of inputs, a set of outputs and a set of declarations. What makes signatures useful is that they contain not only the explicit interface (i.e., ports) of a subsystem, but also its implicit interface (data store reads/writes and non-local froms/gotos) and its imposed interface (data store declarations and scoped visibility tags). As such, signatures have the potential to provide a complete view of the cross-hierarchical data flow in a Simulink model.

The primary goals of signatures are

- Improve comprehensibility of models by reducing the need to examine the system hierarchy to understand data flow,
- Provide ubiquitous information about the subsystems' implicit interfaces, empowering developers to use them more effectively,
- Support automatic signature extraction from existing Simulink models, in order to minimize overhead in using signatures,
- Open the door to providing stronger control over interfaces, by using signatures to, e.g., restrict data flow.

In this section, signatures are defined by set-theoretic means, and studied from a theoretical point of view. Our approach is to

1. Provide abstract definitions of subsystems and of signatures
2. Give inductive definitions of signatures of subsystems
3. Define the notion of consistency for signatures
4. Show how signatures and consistency checking allow us to restrict and verify interfaces.

Section 4 looks at how to use signatures in practice.

3.1 Preliminaries

In what follows, we will use the following notation. Sets will be written in the usual way, with $\{a_1, \ldots, a_n\}$ meaning the set containing the n elements a_i; $a \in S$ means that a is a member of S; $S_1 \subseteq S_2$ means that all

elements of S_1 are contained in S_2; $S_1 \cup S_2$ is the set containing all the elements of S_1 and S_2; $S_1 \backslash S_2$ is the set of elements in S_1 but not in S_2; and $\bigcup_{a \in S} f(a)$ is the set of all $f(a)$ s.t. $a \in S$. Tuples, that is ordered sets, are as usual written $(a_1, \ldots, a_n)$; we will define a relation '$\sqsubseteq$' on tuples below.

We define a subsystem, for our present purposes, as an abstraction of the usual notion of a Simulink subsystem. We see subsystems as merely the set[1] of ports, froms, gotos, visibility tags, data store reads, data store writes, data store declarations, and subsystems contained within. Admittedly, abstracting away subsystems' internal signal flow gives a somewhat impoverished view of subsystems, but this is precisely the idea behind signatures: to simplify data flow analysis of Simulink models by focusing on their cross-hierarchical interconnections.

Note also that we only consider normal virtual subsystems (MathWorks, 2013), in order to simplify the treatment of data flow. Atomic (nonvirtual) subsystems, referenced models, masked subsystems, etc. all affect the implicit interface in ways that, though interesting, detract from the presentation of the basic ideas which we are focused on in this paper. A full treatment is left to future work.

We also avoid global froms, gotos and data stores by "faking" them in an obvious way. Global tags are replaced by scoped tags, with a corresponding visibility tag placed in the top-level subsystem; global data stores are replaced by normal data stores, and the corresponding declaration is moved to the top-level subsystem.

Definition 3.1 (Identifiers)**.**

- $\boldsymbol{PO}$ is the set of all port identifiers (essentially just natural numbers), ranged over by po.
  - po^r represents an input port, and po^w is an output port.
- $\boldsymbol{DS}$ is the set of all data store names, ranged over by $ds_1, ds_2, \ldots$,
  - For any ds, we write ds^d for its declaration, ds^r for its read, and ds^w for its write.
- $\boldsymbol{TG}$ is the set of all scoped tag names, ranged over by $tg_1, tg_2, \ldots$.
  - For any tg, tg^d is its visibility tag, tg^r its from and tg^w its goto.

Definition 3.2 (Subsystem elements)**.** For a subsystem S, define

- $Ch(S) = \{S' \mid S' \in S\}$. (the set of all subsystems contained in S — S's children in the model hierarchy)
- $Pa(S)$ as the S' s.t. $S \in S'$, if it exists, undefined otherwise. (the parent of S)
- $PO^r(S)$ as the set of input ports of S
- $PO^w(S)$ as the set of output ports of S
- $DS^d(S) = \{ds \mid ds^d \in S\}$
- $DS^r(S) = \{ds \mid ds^r \in S\}$
- $DS^w(S) = \{ds \mid ds^w \in S\}$
- $TG^d(S) = \{tg \mid tg^d \in S\}$
- $TG^r(S) = \{tg \mid tg^r \in S\}$
- $TG^w(S) = \{tg \mid tg^w \in S\}$

With subsystems and their related properties defined, we can now define signatures.

Definition 3.3 (Signatures)**.** Let $P \subseteq \boldsymbol{PO}$, $D \subseteq \boldsymbol{DS}$ and $T \subseteq \boldsymbol{TG}$. A signature Σ is a tuple (I, O, M) (input, output, and imposed) where

- $I = (P^I, D^I, T^I)$
- $O = (P^O, D^O, T^O)$
- $M = (D^M, T^M)$

Intuitively, the inclusion of data stores and scoped tags in the inputs (outputs) of a subsystem's signature indicates that those data stores and tags can be (or are) read from (written to) in that subsystem. Inclusion of data stores or tags in the imposed interface is meant to indicate that those data stores or tags are declared in the subsystem.

Armed with the above definitions, we can now examine how to associate signatures with particular subsystems.

3.2 Subsystem signatures

For a given subsystem S, we would like to define a signature $Sig(S)$ which describes the interface of S in the most useful way possible. We have found that there are two complementary and equally important views of a subsystem's interface:

1. The view that shows *potential* inputs and outputs of a subsystem
2. The view which identifies its *actual* inputs and outputs

For a subsystem, we call the first view its *weak signature*, written $Sig^w(S)$, and the second view its *strong signature* $Sig^s(S)$.

For the weak signature, we wish to discover all of the data stores and scoped tags which are *accessible* to a given subsystem, that is those which are declared higher up in the model hierarchy; for the strong

[1] Technically, a subsystem as described here would be a multiset as, e.g., multiple data store reads might be present in a single subsystem. We can ignore this in our presentation because it is only the presence (or absence) of the various elements that we are interested in.

signature, we aim to enumerate those data stores and tags which are *accessed* in a subsystem or its children. Note that the question of whether or not these *are in fact accessed* during the execution of a model is difficult, and requires deep analysis of control and signal flow. What we aim to create, for the second view, is a useful *approximation* of actual inputs and outputs to a subsystem simply by checking for the presence or absence of read blocks and write blocks. This is one of the strengths of the signature approach: providing data flow analysis in a setting where semantics are not available, as is unfortunately the case for Simulink.

In what follows, we will provide inductive definitions of both weak and strong signatures, and then in the next subsection a consistency theorem connecting the two will be presented. First we define a convenient projection function on signatures. If $\Sigma = ((P^I, D^I, T^I), (P^O, D^O, T^O), (D^M, T^M))$, define

$$\Sigma{\downarrow_{P^I}} = P^I,\ \Sigma{\downarrow_{D^I}} = D^I, \text{ etc.}$$

The weak signature is constructed from the top down, reflecting the fact that it tells us about a subsystem's inherited context.

Definition 3.4 (Weak signature)**.** The weak signature $Sig^w(S) = ((P^I_S, D^I_S, T^I_S), (P^O_S, D^O_S, T^O_S), (D^M_S, T^M_S))$ is defines as follows. Firstly, if S is the top-level subsystem, then we set

$$\begin{aligned} P^I_S = D^I_S = T^I_S = P^O_S = D^O_S = T^O_S &= \{\} \\ D^M_S &= DS^d(S) \\ T^M_S &= TG^d(S) \end{aligned}$$

Otherwise,

$$\begin{aligned} P^I_S &= PO^r(S) \\ D^I_S &= Sig^w(Pa(S))\downarrow_{D^I} \cup DS^d(Pa(S)) \setminus DS^d(S) \\ T^I_S &= Sig^w(Pa(S))\downarrow_{T^I} \cup TG^d(Pa(S)) \setminus TG^d(S) \\ P^O_S &= PO^w(S) \\ D^O_S &= Sig^w(Pa(S))\downarrow_{D^O} \cup DS^d(Pa(S)) \setminus DS^d(S) \\ T^O_S &= Sig^w(Pa(S))\downarrow_{T^O} \cup TG^d(Pa(S)) \setminus TG^d(S) \\ &\qquad \setminus TG^w(Pa(S)) \\ D^M_S &= DS^d(S) \\ T^M_S &= TG^d(S) \end{aligned}$$

Remarks 3.5.

1. If a scoped *goto* is encountered, then the corresponding *output* is removed from the signature. This reflects the fact that the same goto cannot appear more than once (in the same scope).
2. All declarations result in new inputs/outputs on child subsystems (except if a scoped goto is in the same subsystem as its declaration).
3. Data stores always appear in both the inputs and outputs. This is due to the fact that we treat data stores in the most liberal way that Simulink allows. Disabling various behaviours for data stores (e.g., write-after-write) could potentially affect the weak signature; we do not explore this here.

The strong signature is constructed from the bottom up, such that the signature of a subsystem also reflects its children's behaviour.

Definition 3.6 (Strong signature)**.** The signature $Sig^s(S) = ((P^I_S, D^I_S, T^I_S), (P^O_S, D^O_S, T^O_S), (D^M_S, T^M_S))$ of a subsystem is defined as follows:

$$\begin{aligned} P^I_S &= PO^r(S) \\ D^I_S &= \Big(\bigcup_{S' \in Ch(S)} (Sig^s(S')\downarrow_{D^I}) \cup DS^r(S) \setminus DS^d(S) \\ T^I_S &= \Big(\bigcup_{S' \in Ch(S)} (Sig^s(S')\downarrow_{T^I}) \cup TG^r(S) \setminus TG^d(S) \setminus T^O_S \\ P^O_S &= PO^w(S) \\ D^O_S &= \Big(\bigcup_{S' \in Ch(S)} (Sig^s(S')\downarrow_{D^O}) \cup DS^w(S) \setminus DS^d(S) \\ T^O_S &= \Big(\bigcup_{S' \in Ch(S)} (Sig^s(S')\downarrow_{T^O}) \cup TG^w(S) \setminus TG^d(S) \\ D^M_S &= DS^d(S) \\ T^M_S &= TG^d(S) \end{aligned}$$

Remarks 3.7.

1. Scoped gotos (data store writes) result in outputs on the current subsystem *and all subsystems above it* until a visibility tag (data store declaration) is reached.
2. If a scoped tag is not included in the outputs (yet), and a scoped from is found, then the tag is placed on the *inputs*. This is done because the corresponding goto is expected to be found higher up in the model hierarchy.

With the two types of signature defined, we now explore the connection between them.

3.3 Signature consistency

To compare two signatures, we define the relation '$\sqsubseteq$' as follows:

Definition 3.8 (Consistency)**.** If $I_1 = (P^I_1, D^I_1, T^I_1)$ and $I_2 = (P^I_2, D^I_2, T^I_2)$, then $I_1 \sqsubseteq I_2 \iff (P^I_1 = P^I_2 \wedge D^I_1 \subseteq D^I_2 \wedge T^I_1 \subseteq T^I_2)$. Similarly, if $O_1 = (P^O_1, D^O_1, T^O_1)$ and $O_2 = (P^O_2, D^O_2, T^O_2)$, then $O_1 \sqsubseteq O_2 \iff (P^O_1 = P^O_2 \wedge D^O_1 \subseteq D^O_2 \wedge T^O_1 \subseteq T^O_2)$. Now, for signatures $\Sigma_1 = (I_1, O_1, M_1)$ and $\Sigma_2 = (I_2, O_2, M_2)$, we define

$$\Sigma_1 \sqsubseteq \Sigma_2 \iff I_1 \sqsubseteq I_2 \wedge O_1 \sqsubseteq O_2 \wedge M_1 = M_2.$$

When $\Sigma_1 \sqsubseteq \Sigma_2$, we say that Σ_1 is *consistent* relative to Σ_2.

Roughly speaking, $\Sigma_1 \sqsubseteq \Sigma_2$ means that the input/output behaviour of Σ_1 does not "exceed" that of Σ_2. Some additional remarks are in order as to how the above relation is defined. First of all, notice that if $\Sigma_1 \sqsubseteq \Sigma_2$, then Σ_1 and Σ_2 must have an identical set of ports. This is due to the fact that adding and removing ports from a subsystem is a nontrivial operation, one that cannot (at present) usefully be expressed using a signature.[2] On the other hand, the inclusion or omission of data stores and scoped tags in the implicit interface is less constrained. (Similarly, the set of declarations in a subsystem is somewhat malleable, although we do not go into detail about this here; for our present development weak and strong signatures are identical.)

Before presenting the consistency theorem between weak and strong signatures, the concept of validity of subsystems must be introduced. A subsystem is *valid* whenever

- All from tags and goto tags are in the scope of visibility tags; similarly all data store reads and data store writes are in the scope of data store declarations.
- There is exactly one goto tag in the scope of each corresponding visibility tag.

These restrictions are more than reasonable as any Simulink model that violates them will result in an error when performing a simulation or code generation.

Theorem 3.9. *$Sig^s(S) \sqsubseteq Sig^w(S)$ for any valid S.*

Proof. By induction on the height of S. The proof is straightforward. □

What is interesting about the above result is that it does not hold for *some* subsystems that are not valid. Specifically, if there are two identical scoped gotos in two subsystems where one is above the other in the model hierarchy, then the theorem does not hold. However, the proof *does* go through if the two gotos are in separate subsystems with a common ancestor. This shows that simply computing the signatures for subsystems can automatically discover errors before compile time.

Let us explore this idea further to show how signatures can be useful as an analysis tool. Say we (manually) define some signature Σ for a subsystem S. Then by computing $Sig^s(S)$ and $Sig^w(S)$, we can validate Σ:

- If $\Sigma \not\sqsubseteq Sig^w(S)$, then Σ is *incompatible* with the context of S. This can mean, for example, that Σ refers to an undeclared data store.
- If $Sig^s(S) \not\sqsubseteq \Sigma$, then S does not *satisfy* Σ; e.g., S writes to a data store that is not in the outputs of Σ.

So signatures can be employed as *interface specifications* for subsystems, which can be automatically checked for consistency. As such, a signature can be used to enforce encapsulation by restricting access to data, for example, by forcing a data store to be read-only. Since Simulink provides no such facilities, signatures can be of significant benefit from a software engineering perspective.

Assume you are given a signature specification for a subsystem. If the given signature is not consistent with the generated weak signature, specifically there are extra data flow mechanisms in the weak signature that are not in the given signature, it shows the potential for the subsystem to access data flow mechanisms that may cause interference with other subsystems in the model hierarchy. If the given signature is not consistent with the generated strong signature, specifically if the strong signature contains data flow mechanisms that were not in the original given signature, then the subsystem has access to data flow mechanisms it should not. In this case, the designed system modularity has been broken. Using the generated strong and weak signatures can aid in the checking of a given signature and the implementation of the subsystem for consistency, as mentioned earlier.

The definitions and theorem above are simplified for this initial presentation, but they can be refined to make signatures even more expressive and useful. If we incorporate *types* into signatures, then checking signatures for consistency grows to encompass type checking.

The consistency-checking technique developed above demonstrates that signatures can be useful in practice. In fact, there are many practical ways in which signatures can be used; the next section explores these in depth.

4 USING SIGNATURES

With the abstract notion of signature fixed, we now adopt a more pragmatic viewpoint and explore the applications of signatures in practice. We have already touched on some practical benefits of signatures in the last section, namely interface specification, automatic extraction and consistency checking. Beyond these, there are many more practical uses and benefits of signatures. Below, we explore some of these uses.

[2] However, in future work, when signatures are extended to support *types*, the relationship between the ports in two signatures will be more complex.

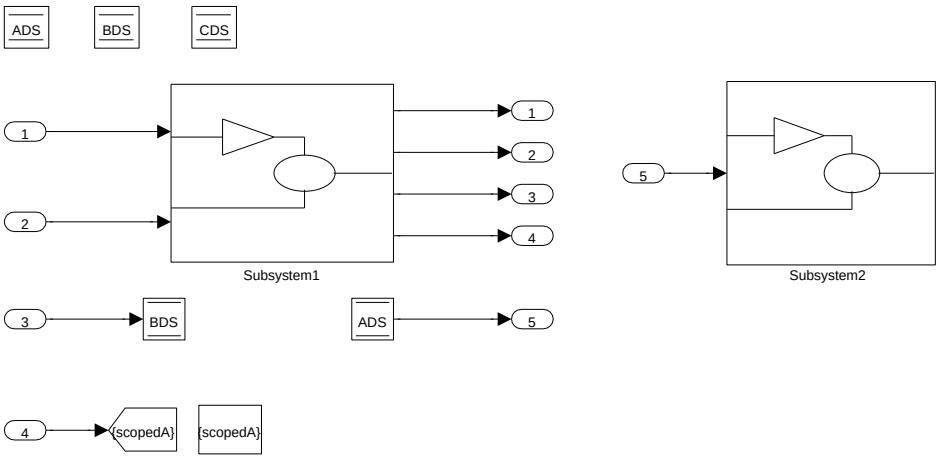

Figure 2: Top level system before signature extraction/inclusion

- How to incorporate signatures into a software engineering methodology;
- How to use signatures to (re)organize and classify parts of subsystem interfaces;
- How to use signatures to aid in a real-world Simulink refactoring/reverse-engineering effort;
- How to apply signatures to a *concrete* application, i.e., included into Simulink subsystems, and how to automate this process;
- How to use signatures to facilitate testing, model-in-the-loop simulation and instrumentation;
- How to make use of signatures to apply typing to subsystems.

We will also present evidence that no change in behaviour or performance is incurred when including a signature in a subsystem.

4.1 Using signatures for software engineering practices

As mentioned in Section 3, Simulink lacks built-in data flow management functionality, from the point of view of providing good subsystem interface management. Therefore, a *discipline* is needed to enforce good design practice at the level of interfaces. The systematic use of signatures provides just such a discipline.

Prescriptive signatures. As an example, imagine a scenario where a model is under development, and a new subsystem needs to be created. The context of this subsystem being fixed, we can automatically determine its weak signature by using the tool mentioned later in this section. This signature can then be refined by, e.g., removing data stores from the outputs if they are to be read-only from within the subsystem, or removing tags altogether if they are not relevant to the subsystem.

Creating a signature before developing a subsystem (a *prescriptive* signature) allows for fine-grained control of data flow throughout the model architecture, which was discussed in detail in Section 3.3. It provides a mechanism by which we can apply *information hiding* and encapsulation within a Simulink model.

Classifying inputs. Making the implicit interface visible empowers data stores and scoped tags. We can use these features much more effectively when they are easy to identify. For example, if we have a subsystem with some inputs which are dynamic, i.e., are changing throughout the execution of the subsystem, and some which are static, that is do not (or rarely) change. If we apply the discipline of using scoped tags for static inputs and ports for dynamic inputs, this significantly declutters the explicit interface of the subsystem. Without signatures, it could be argued that such an increased use of scoped tags is detrimental to the comprehensibility of the model. With signatures, scoped tags are no less visible than ports.

Refactoring and reverse engineering. When working with large under-documented models, it can be a daunting task to navigate and understand the hierarchy of subsystems they contain. But if we see subsystems as modules, then understanding the interfaces to subsystems is crucial to proper documentation of the model. Signature extraction gives us a good starting point for interface documentation,

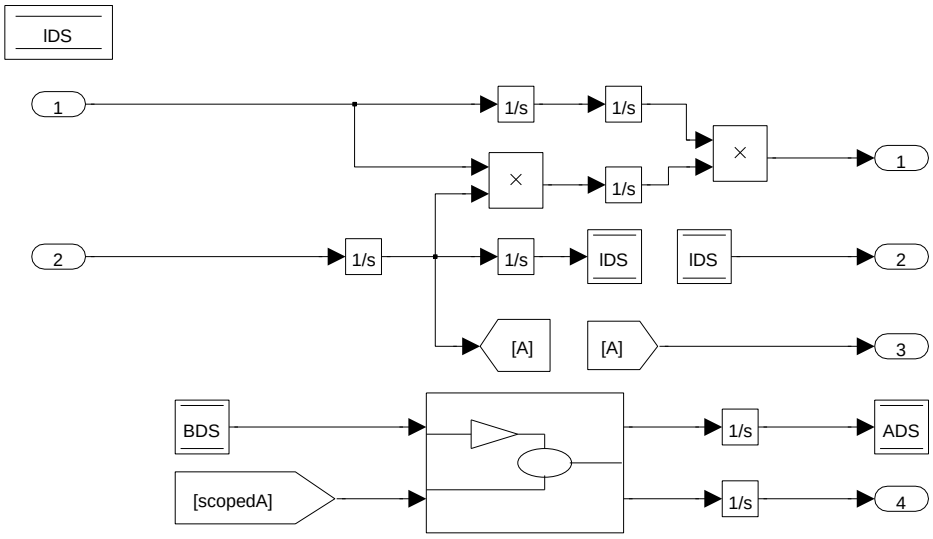

Figure 3: Subsystem1 before signature extraction/inclusion

without much overhead for the developer.

First, computing the weak signature gives an understanding of the *context* of a subsystem; by examining it, we can see if there are any unnecessary tags or data stores in the interface. Computing the strong signature for the subsystem can help identify whether tags and data stores are read from or written to. Finally, the actual signature for the subsystem would be somewhere in between, restricting access to context as fits the situation. Just because a tag is not read in a subsystem does not mean that it cannot (should not) be read. A good example of this is a state timer.

From an information-hiding point of view (Parnas, 1972), the signatures at various levels give a potential method for discovering module secrets, and also a mechanism to ensure that these secrets are not exposed by the interface.

An interesting usage of signatures in the context of refactoring is *rescoping*. In the course of examining some large-scale industrial Simulink models, we came across a situation where many data stores were being defined at or near the top-level of the subsystem hierarchy. This essentially amounts to programming with a large number of global variables, which is of course undesirable. By computing the weak and strong signatures for each of the subsystems and comparing them, we were able to "push down" declarations automatically to the lowest subsystem possible. The signatures also provided a clear view of the sizes of implicit interfaces of all of the subsystems, which, after the push-down operation, were substantially reduced.

4.2 Concrete application of signatures

Beyond the abstract definition of signatures given in Section 3, we can actually express the signature in Simulink and include it in the subsystem itself. Figure 2 shows the top level of a system, of which we are going to concentrate on the Subsystem1 on the left. Figure 3 shows the mentioned highlighted subsystem, for which we are going to present the extracted strong and weak signatures. All figures have been created using the automated signature extraction tool in Simulink and exported from Matlab.

The left hand side of Figure 4 presents the (strong) signature extracted from the original subsystem. Data store reads and scoped froms, fed into terminators, are included for each input and output in the implicit interface; declarations are grouped together at the bottom. Note that the signature goes beyond simply presenting the interface, but makes the additional step of feeding all input ports into local gotos, and feeding output ports from local froms. This step was taken in our implementation after we observed that our industry partner's models used this technique whenever a port needed to be used multiple times in a model; in fact, this was one of the initial motivations for the development of signatures in the first place.

The signature presented in Figure 4 effectively augments subsystems with a "data-flow legend". If applied systematically it serves as a form of self-documentation in Simulink; a rough analogy might be the use of function prototypes and header files in C. (A signatures effectively provides a *data flow view* as defined by (Quante, 2013); in fact, in his defini-

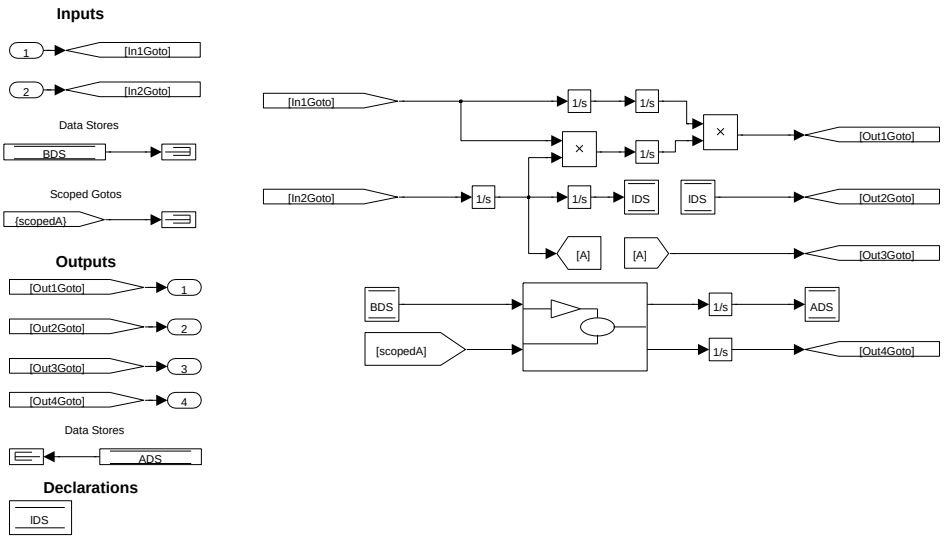

Figure 4: Subsystem1 after strong signature extraction/inclusion

tion of data flow view, he points out that "such views can be very helpful for tracking data flows through a system — especially when there are additional hidden dependencies.) The signature allows for all implicit and explicit data flow to be found in one place for a subsystem, instead of opening multiple subsystems to understand the data flow for a single subsystem. We are undertaking a study to demonstrate improved readability and comprehensibility of models when signatures are included — see Section 5 for details.

Figure 5 shows both strong and weak signature, side by side, for the subsystem given by Figure 3. For this example, it is assumed in Subsystem1 there are no access to data stores, gotos or froms in the child subsystem. The highlighted blocks show the difference between strong and weak signature. The blocks that are included in the weak signature show what the subsystem has access to, but is not necessarily using. The strong signature shows what the subsystem is actually using.

Automatic extraction of signatures. The current prototype of the signature tool is a MATLAB function, that is executed from the command line. It annotates a loaded Simulink model with its signature. Weak signature is implemented in a recursive top-down algorithm on the system tree of a model, while the strong is implemented bottom-up recursive algorithm. The prototype tool has been used on real-world examples, but due to the fact that the automotive industry code we are working with is proprietary information that we are not allowed to disclose, we were not at this time able to publish a real world use case.

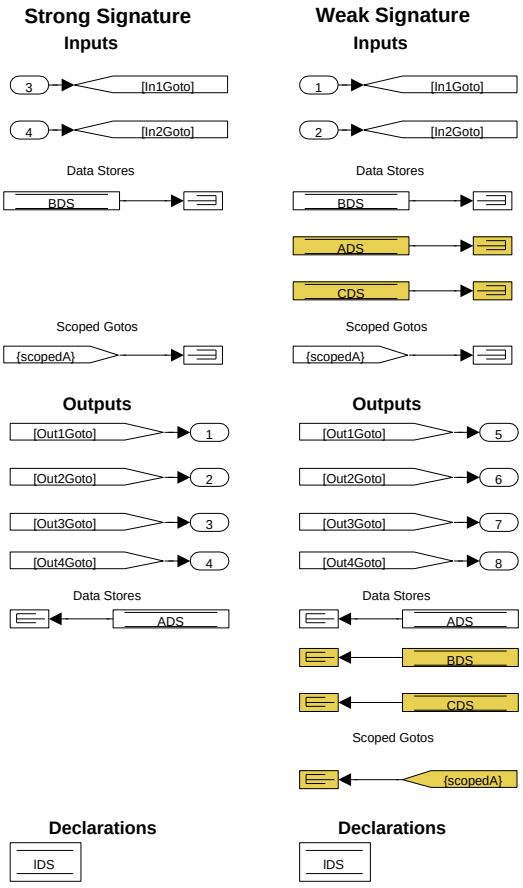

Figure 5: Strong and Weak Signatures for Subsystem1

Harnessing. Including the signature in a subsystem provides us with a "plug-in" interface to that subsystem, in that signatures make it easy to attach various things to subsystems. We can take advantage of this when developing various kinds of *harnesses* for subsystems, e.g., testing, model-in-the-loop, and instrumentation. Importantly, the handling of the implicit interface when developing a harness is significantly improved.

There is a considerable lack of proper consideration of implicit data flow in modern testing tools. More precisely, when generating testing harnesses for a subsystem, some testing tools neglect to mimic data passed in/out of the subsystem by data store reads/writes. We looked into two major commercial automatic test generation tools[3] to explore cases when a data store defined outside a subsystem is being read from or written to in the subsystem. Both tools support *subsystem extraction*: a subsystem together with its necessary execution context is extracted into a new model, and tests can then be generated for the new model. The two tools deal with data store reads and data store writes in different ways. The first tool extracts the subsystem, and for both data store reads and data store writes adds only definitions of data stores (data store memory blocks) to the model, therefore not taking into account the data flow through inherited data stores. The other tool also adds the necessary definitions to the extracted model, but does mimic the data flow through a data store read: for a given data store that is being read in the subsystem, an inport is added that writes into the data store. Further, any constraints being defined for the data store (minimum and maximum values) are assigned to the new inport. When it comes to a data store write, on the other hand, this tool treats it similarly to a data store read: it adds an inport that writes into the data store, and then effectively grounds this input in the generated test harness.

The signature can address the lack of implicit data flow consideration in test generation, since it explicitly presents the subsystem's interface, therefore clearly identifying implicit data flow in and out of the subsystem so that it is not overlooked in test harness generation. Therefore, the signature can be used for an automatic test harness generation that would account for all the data passed through the subsystem. One important note about harnessing using signatures is that the harness is *separated* from the rest of the subsystem, in that it appears entirely on the left-hand side. The main benefit of this is that the inclusion of the harness does not obscure the actual subsystem, and that in the case where we only have a harnessed subsystem at our disposal it would be easy to remove said harness to recover the original subsystem.

Also, adding the strong typing information to the signature (as presented next in this section) further enhances testability: the typing information can be used to focus test generation only to data of interest.

Typing. Inclusion of the signature in a subsystem has another major benefit: we can use the patterns presented in (Rau, 2002) to incorporate *strong typing* into subsystems' interfaces. By using masked subsystems or bus objects that ensure that signals are of a particular type, Rau presents a set of design patterns for Simulink that enforce types on a subsystem's ports. But there are obvious drawbacks to his approach. First, the application of his design patterns presents overhead for developers since they essentially need to be programmed by hand. Second, his typing only applies to the *explicit* interface of a subsystem, i.e., its ports. By applying the essential elements of his approach, attached to the signature as opposed to included as a pattern, we can apply strong typing to the implicit interface as well. Typing in this manner is quite clean as well, as it is completely contained within the signature and thus does not visually affect the rest of the subsystem.

Behaviour preservation. An obvious question which arises when including signatures in a subsystem is whether we have changed the behaviour of that subsystem. To show that the behaviour has been preserved, we provide three arguments: first, by intuitive analysis of data flow and control flow in the signature; second, by using coverage testing to profile subsystems before and after signature inclusion; and third, by comparing the generated code before and after.

Our intuitive analysis follows (Schäfer et al., 2009): if name-binding, control flow and data flow are preserved, then we have a strong argument that behaviour has been preserved. By simply analyzing the patterns in the signature, we can make a simple argument that we have not changed the behaviour of a subsystem by including its signature. Ports are fed with local froms and gotos; as long as we do not choose conflicting names, these are just shorthand for directly connected signals so they cannot change behaviour. Data stores and scoped tags, whether they are inputs or outputs, are just included in their "read" form (data store read and scoped from respectively) and fed into terminators. Here, we can make an easy argument that data flow has not changed, but it is trickier to argue that control flow has not been affected. The argument hinges on whether or not Simulink optimizes

[3]Educational licensing terms of these tools do not allow us to publish the names of the tools.

away a data store read which feeds into a terminator, or whether it results in the insertion of an extra computation step. In this case, however, control flow cannot "change" so much as incur a (tiny) performance penalty. Finally, as far as the declarations go, we are just moving data store declarations and visibility tags from within the existing subsystem into the signature. This is a cosmetic change which certainly should not affect either data flow or control flow.

To supplement the above intuitive argument, we have also used a testing tool[4] to perform coverage testing of a set of subsystems before and after signature inclusion. The results were identical in all cases. We also generated code before and after signature extraction; in this case, code was identical (with some cosmetic changes) for every part of the signature except for data stores. When including a data store in the signature in a case where the data store was *not actually read from or written to in that subsystem*, the generated code contained some extra instructions. Although our own inspection of this supplementary code leads us to believe that it is nothing more than bookkeeping code, we cannot be sure of its intent or effect in any context. We need to look at this in more detail in future work.

5 FUTURE WORK

In order to show the benefits of the proposed signature for subsystems in Simulink, we plan to perform a comprehensibility study with software engineers from our industrial partner. As they already use Simulink for development, our field of test subjects will already be familiar with Simulink and data flow mechanisms. We plan to present them with examples of hierarchical subsystems with and without signatures and ask them to identify data flow in the subsystem and perform small tasks that involve understanding the data flow of the overall system. Based on the time it takes to complete these tasks and feedback we receive from the software engineers after they have completed the study, we will be able to draw conclusions on the benefits of the signature for subsystems in Simulink. Similar studies have been performed with visual programming languages in (Cox et al., 2004; Green and Petre, 1992). We plan to use these studies as guides to help create our own.

Besides the study just mentioned, future work will also include incorporation of signature checking into Simulink, further elaboration of type checking using signatures, including signatures in a comprehensive software engineering methodology currently being developed at McSCert, and continuing our work on harnessing and testing using signatures. In particular, we would like to extend the work on slicing Simulink models presented by (Reicherdt and Glesner, 2012) by elaborating their treatment of data flow; in their paper only explicit signals are treated. Work on creating a metrics for modularity of the system using the difference between the weak and strong signatures will also be explored. Simply this metric would examine that if there is a large difference in size between the weak and strong signatures, it means the subsystem has access to many data flow mechanisms it is not using, creating the potential for a developer to access something incorrectly or to interfere with another subsystem in hierarchy. Work needs to be done on how to compute a numeric value to represent this metric.

[4]Educational licensing terms of the tool does not allow us to publish the name of the tool.

6 CONCLUSION

This paper has presented a novel approach to handling interfaces in Simulink, namely signatures. In our presentation of signatures, we have striven to present as complete a picture as possible: from the abstract theory of signatures to the concrete application of signatures in Simulink; motivated by discussion and examples. The inclusion of signatures in a Simulink subsystem gives the user one place to get the entire context of the subsystem, to aid in the user's comprehension of the subsystem's explicit and implicit data flow. Also, by examining changes to the strong and weak signatures during development, we can identify issues if system modular structure has been broken, as we have discussed in Section 3. We hope to have convinced the reader of the usefulness and practicality of signatures. Our industry partner has already expressed a keen interest in the work and we are currently collaborating closely with them to develop and deploy the technique in practice.

REFERENCES

Cox, A., Gauvin, S., and Smedley, T. (2004). Towards comprehensible control flow in visual data flow languages. In *International Conference on Distributed Multimedia System*.

Green, T. and Petre, M. (1992). When visual programs are harder to read than textual programs. In *Human-Computer Interaction: Tasks and Organisation, Proceedings 6th European Conference on Cognitive Ergonomics*.

MathWorks (2008). Best practices for data stores. `http://www.mathworks.com/support/solutions/`

attachment.html?resid=1-6F3I63&solution=1-5NM3AN. [Online; accessed 20-August-2013].

MathWorks (2013). Subsystem, Automic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem. http://www.mathworks.com/help/simulink/slref/subsystem.html. [Online; accessed 25-August-2013].

Meyer, B. (1992). Applying "Design by contract". *IEEE Computer*, 25(10):40–51.

Parnas, D. (December 1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058.

Quante, J. (2013). Views for efficient program understanding of automotive software. *Softwaretechnik-Trends*, 33(2).

Rau, A. (2000). Potential and challenges for model-based development: in the automotive industry. *Business Briefing: Global Automotive Manufacturing and Technology*, pages 124–138.

Rau, A. (2001). Decomposition and interfaces revisited. *Softwaretechnik-Trends*, 21(2):19–23.

Rau, A. (2002). On model-based development: A pattern for strong interfaces in SIMULINK. *Softwaretechnik-Trends*.

Reactive Systems (2012). Reactis. http://www.reactive-systems.com.

Reicherdt, R. and Glesner, S. (2012). Slicing MATLAB simulink models. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 551–561, Piscataway, NJ, USA. IEEE Press.

Schäfer, M., Verbaere, M., Ekman, T., and de Moor, O. (2009). Stepping stones over the refactoring rubicon. In Drossopoulou, S., editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer.

Schatz, B., Prestchner, A., Huber, F., and Philipps, J. (2002). Model-based development of embedded systems. Technical report, Technische Universitat Muchen, Germany.