Science of Computer Programming 113 (2015) 29–50

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Signature required: Making Simulink data flow and interfaces explicit

Marc Bender, Karen Laurin, Mark Lawford, Vera Pantelic *, Alexandre Korobkine, Jeff Ong, Bennett Mackenzie, Monika Bialy, Steven Postma

McMaster Centre for Software Certification, Department of Computing and Software, McMaster University, Hamilton, ON, L8S 4L8, Canada

ARTICLE INFO

Article history:
Received 16 June 2014
Received in revised form 10 April 2015
Accepted 10 July 2015
Available online 30 July 2015

Keywords:
Simulink
Interfaces
Model transformation
Software engineering
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, as well as 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. Furthermore, outside of the model setting, signatures have proven to be an asset when exported and included in software documentation.

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 [1–3], 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 [4–6] to allow for division of tasks among developers, as well as ease maintainability, comprehensibility, verifiability, and module reuse. 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'

* Corresponding author. Tel.: +1 289 674 0250x59056.
E-mail addresses: bendermm@mcmaster.ca (M. Bender), laurink@mcmaster.ca (K. Laurin), lawford@mcmaster.ca (M. Lawford), pantelv@mcmaster.ca (V. Pantelic), korobka@mcmaster.ca (A. Korobkine), ongj2@mcmaster.ca (J. Ong), mackeb@mcmaster.ca (B. Mackenzie), bialym2@mcmaster.ca (M. Bialy), posmasm@mcmaster.ca (S. Postma).

http://dx.doi.org/10.1016/j.scico.2015.07.005
0167-6423/

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 a direct connection. Also complicating large models is the fact that 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 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 behavior. 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
- automatic extraction of the signature from existing models, and automatically updated as required
- detaches the interface from the subsystem, thus separating its internal behavior from its external manifestation.

Our efforts are motivated by the issues we have found with data flow in visual languages when modeling large complex systems, 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 [7,8]. The results of [7] 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 [8] 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 developers in data flow comprehension within Simulink.

Similar work has been proposed in [3]. 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.

This paper is an extended version of [9] that contains the following additional contributions. Building on the ideas introduced in [9], we define the *signature metric*, a software metric that employs signatures to evaluate quality of a Simulink model design. We demonstrate the applicability of the metric in evaluating modularity in large industrial models. Also, an alternative, more compact representation of a signature is presented that introduces the notion of "updates" — data stores and goto tags that may be (are) both read from and written to in the case of the weak (strong) signature. Furthermore, the discussion on applicability of signatures is significantly improved in this paper compared to [9], and an industrial automotive model is used to demonstrate the benefits of signatures. Most notably, while [9] contained only an outline of the application of signatures in test harnessing, this paper demonstrates a large improvement in testability for the example industrial automotive model when signatures are used to help identify the implicit interface and apply it in test harnessing. Not only does the testing coverage improve, therefore increasing the probability of finding a fault in a design, but also the testing effort (number of test cases) reduces significantly. The use of signatures in test harnessing has been fully automated. Additionally, while most of the discussion on the applicability of signatures in [9] focused on the inclusion of signatures in Simulink models, this paper describes how our automotive industry partner uses automatically generated signatures in their software development process for design description documentation.

The outline of the paper is as follows. Section 2 presents a careful analysis of Simulink data flow constructs and their behaviors. Section 3 offers a formal definition of signatures, along with a discussion of their properties and variants. Section 4 is devoted to using signatures in practice, and discussing their benefits. Section 5 defines a metric based on signatures for assessing modularity and discovering deficiencies in model design. Finally, in Section 6 we present conclusions and future work.

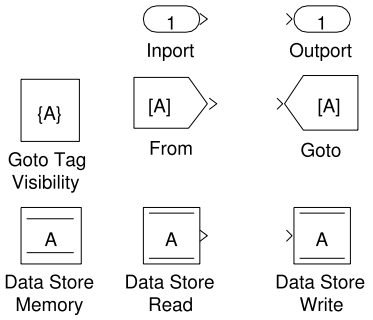


Fig. 1. Additional Simulink data flow mechanisms.

2. Data flow in Simulink

In this section, we present our analysis and criticism of data flow in Simulink. Simulink is used for model-based development and is integrated into Matlab's environment. 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 (2015a) and Simulink (8.5).

In order to model a complex system, the ability to decompose it into subsystems is required to make the system more comprehensible, maintainable, and to allow multiple developers to simultaneously work on different parts of the system. Simulink allows for systems to be embedded in another system, effectively creating a hierarchy of subsystems. Simulink's blocks 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.

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 Fig. 1). We now present some background information on these mechanisms. It is important to note that for simplicity we are only using virtual subsystems [10] 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, inports and outports 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) in which it is contained, 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 through the block's tag visibility parameter. The permitted scopes of 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.
- *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, as well as any other subsystem lower in the model hierarchy. These are identified by curly braces around the block name.
- *Global* – The from and goto blocks can be used anywhere in the model hierarchy. These are identified by the block name with no additional surrounding characters.

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 data store read and write blocks, there is the potential for the order of execution to change in different releases. Ref. [11] recommends the following ways to handle the order in which data store read and writes are executed in a system:

- Use function call subsystems 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 data flow 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 is 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 is comprised of 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 tags whose visibilities 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 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 wherever 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, 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 to

- improve comprehensibility of models by reducing the need to examine the system hierarchy to understand data flow,
- provide ubiquitous information about 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, and
- 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, and
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. Union and difference operations have equal precedence. Intersection has higher precedence than union and difference; all three operations are left-associative. $\bigcup_{a \in S} f(a)$ is the set of all $f(a)$ s.t. $a \in S$. Cardinality of set S is denoted as $|S|$. 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 [10], 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).*

- ***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.
- ***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.
- ***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
- $DS^d(S) = \{ds \mid ds^d \in S\}$
- $DS^r(S) = \{ds \mid ds^r \in S\}$

[1] Technically, a subsystem as described 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.

- $DS^w(S) = \{ds \mid ds^w \in S\}$
- $TG^d(S) = \{tg \mid tg^d \in S\}$
- $TG^r(S) = \{tg \mid tg^r \in S\}$
- $TG^w(S) = \{tg \mid tg^w \in S\}$

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) *only* read from (*only* written to) in that subsystem. The 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 which identifies *actual* inputs and outputs of a subsystem
2. The view that shows its *potential* inputs and outputs

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

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 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 first view, is a useful *approximation* of a subsystem's actual inputs and outputs 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 strong and weak 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 strong signature is constructed from the bottom up, such that the signature of a subsystem also reflects its childrens' behavior.

Definition 3.4 *(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:

$$P^I_S = PO^r(S)$$

$$D^I_S = \Big(\bigcup_{S' \in Ch(S)} Sig^s(S'){\downarrow_{D^I}}\Big) \cup DS^r(S) \backslash DS^d(S)$$

$$T^I_S = \Big(\bigcup_{S' \in Ch(S)} Sig^s(S'){\downarrow_{T^I}}\Big) \cup TG^r(S) \backslash TG^d(S) \backslash T^O_S$$

$$P^O_S = PO^w(S)$$

$$D^O_S = \Big(\bigcup_{S' \in Ch(S)} Sig^s(S'){\downarrow_{D^O}}\Big) \cup DS^w(S) \backslash DS^d(S)$$

$$T^O_S = \Big(\bigcup_{S' \in Ch(S)} Sig^s(S'){\downarrow_{T^O}}\Big) \cup TG^w(S) \backslash TG^d(S)$$

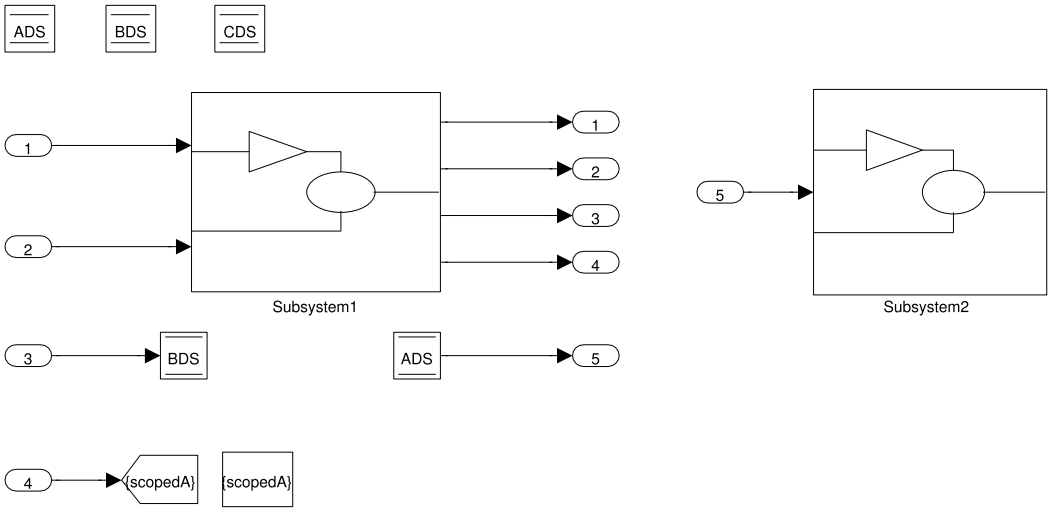


Fig. 2. Top level system before signature extraction/inclusion.

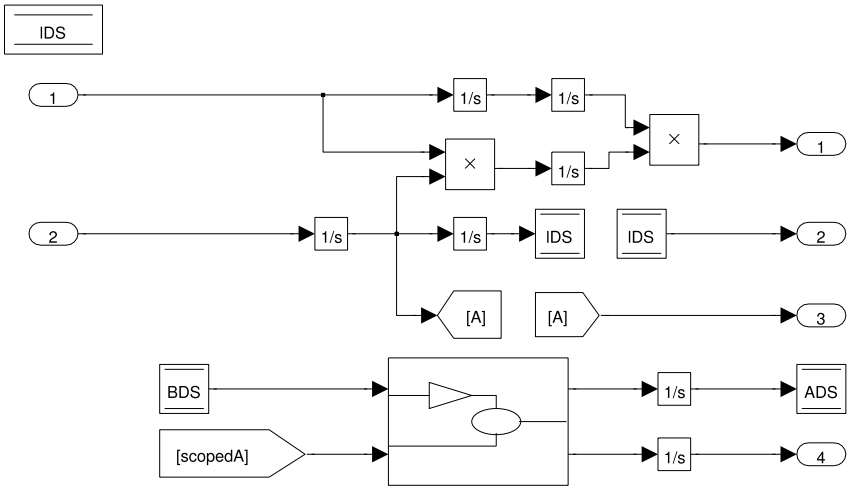


Fig. 3. Subsystem1 before signature extraction/inclusion.

$$D_S^M = DS^d(S)$$

$$T_S^M = TG^d(S)$$

Remarks 3.5.

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.

Beyond the abstract definition of strong signature presented here, we can actually express the signature in Simulink and include it in the subsystem itself. Fig. 2 shows the top level of a system, from which we are going to concentrate on the Subsystem1 on the left. Fig. 3 shows the mentioned subsystem, for which we are going to present the extracted strong signature. For this example, it is assumed that in Subsystem1 there is no access to data stores, gotos or froms in the child subsystem. The subsystem Subsystem1, with its strong signature included on the left, is shown in Fig. 4. All figures have been created using the Signature Tool we developed, which will be presented in Section 4, and then exported from Matlab.

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

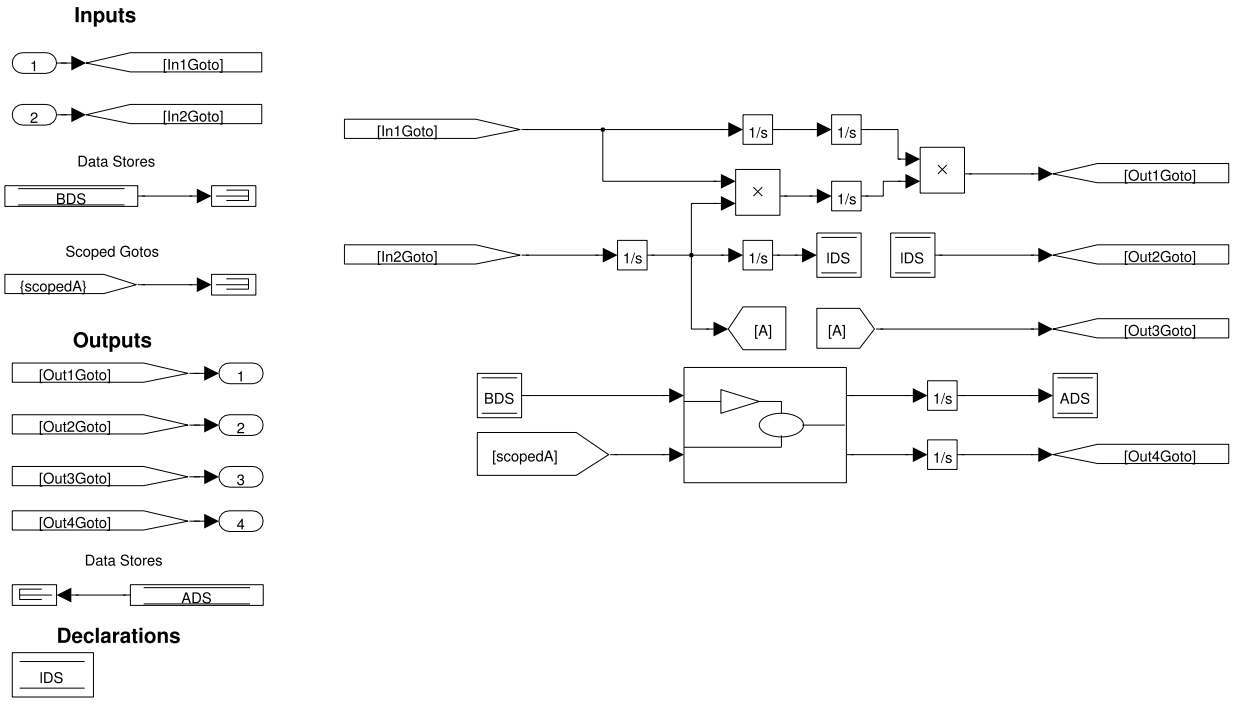


Fig. 4. Subsystem1 after strong signature extraction/inclusion.

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

Definition 3.6 *(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 defined as follows. Firstly, if S is the top-level subsystem, then we set

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

Otherwise,

$$P^I_S = PO^r(S)$$
$$D^I_S = Sig^w(Pa(S))\downarrow_{D^I} \cup DS^d(Pa(S)) \backslash DS^d(S)$$
$$T^I_S = Sig^w(Pa(S))\downarrow_{T^I} \cup TG^d(Pa(S)) \backslash TG^d(S)$$
$$P^O_S = PO^w(S)$$
$$D^O_S = Sig^w(Pa(S))\downarrow_{D^O} \cup DS^d(Pa(S)) \backslash DS^d(S)$$
$$T^O_S = Sig^w(Pa(S))\downarrow_{T^O} \cup TG^d(Pa(S)) \backslash TG^d(S) \backslash TG^w(Pa(S))$$
$$D^M_S = DS^d(S)$$
$$T^M_S = TG^d(S)$$

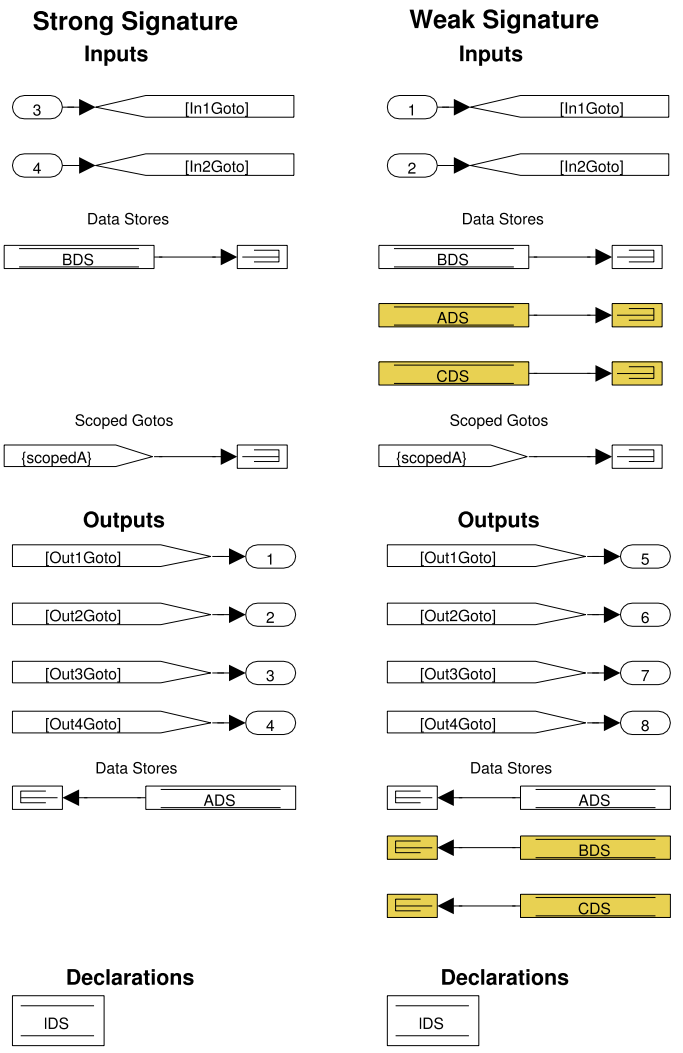


Fig. 5. Strong and weak signatures for Subsystem1.

Remarks 3.7.

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 behaviors for data stores (e.g., write-after-write) could potentially affect the weak signature; we do not explore this here.

Fig. 5 shows the weak signature of Subsystem1 from Fig. 2, side by side its strong signature. As expected, the data items of the weak signature subsume the data items of the strong signature (the highlighted blocks show the difference between the strong and weak signature). We now formally explore the connection between strong and weak signatures.

3.3. Signature consistency

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

Definition 3.8 *(Consistency).* If $I_1 = (P_1^I, D_1^I, T_1^I)$ and $I_2 = (P_2^I, D_2^I, T_2^I)$, then $I_1 \sqsubseteq I_2 \iff (P_1^I = P_2^I \wedge D_1^I \subseteq D_2^I \wedge T_1^I \subseteq T_2^I)$. Similarly, if $O_1 = (P_1^O, D_1^O, T_1^O)$ and $O_2 = (P_2^O, D_2^O, T_2^O)$, then $O_1 \sqsubseteq O_2 \iff (P_1^O = P_2^O \wedge D_1^O \subseteq D_2^O \wedge T_1^O \subseteq T_2^O)$. 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 behavior 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 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.* □

As mentioned before, Fig. 5 shows both the strong and weak signature for Subsystem1 from Fig. 2. The highlighted blocks show the difference between the strong and weak signature. Obviously, the strong signature of the subsystem is consistent relative to the subsystem's weak signature.

What is interesting about Theorem 3.9 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 that 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 which 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.

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

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.

3.4. Signatures with updates

An alternative representation of signatures which may be useful is one that introduces the concept of *updates*. Updates are data stores and tags that are (can be) both read from and written to. Next, we provide a formal definition.

Definition 3.10 *(Strong/weak signature with updates).* Given the definitions of a strong/weak signature $Sig^{s/w}(S) = ((P_S^I, D_S^I, T_S^I), (P_S^O, D_S^O, T_S^O), (D_S^M, T_S^M))$ as in Definitions 3.4 and 3.6, a strong/weak signature with updates is defined as follows:

$$Sig_U^{s/w}(S) = ((P_S^I, D_S^I \setminus D_S^O, T_S^I \setminus T_S^O), (P_S^O, D_S^O \setminus D_S^I, T_S^O \setminus T_S^I), (D_S^U, T_S^U), (D_S^M, T_S^M)),$$

$$\text{where } D_S^U = D_S^I \cap D_S^O \text{ and } T_S^U = T_S^I \cap T_S^O$$

To clarify, strong/weak signatures with updates "move" scoped tags and data stores appearing in both inputs and outputs into a separate set; the set of updates. As a result, those items no longer appear in the inputs and outputs of the signature. Fig. 6 demonstrates a weak signature representation which uses updates for the subsystem Subsystem1 as in Fig. 2. As one can notice in Fig. 5, all the data stores of the weak signature appear in both inputs and outputs — in other words, they could be read from or written to. Therefore, in Fig. 6, all the data stores appear as updates instead. This approach of including updates is beneficial in the visual representation of the signatures because it avoids the need to visually search through inputs and outputs, checking whether they appear both as reads and writes.

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 automatic extraction, interface specification, and consistency-checking. Beyond these, there are many more practical uses and benefits of signatures. Below, we explore some of these uses:

- 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 use signatures for software documentation purposes.
- How to apply signatures to a *concrete* application, i.e., included in Simulink subsystems, as shown in Section 3, 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 behavior or performance is incurred when a signature is included in a subsystem.

Further, some of the benefits of signatures will be illustrated on an industrial automotive model that implements the control of the gasoline engine in hybrid electric vehicles while it is starting/stopping. We will be focusing on applying signatures to the model's subsystem Compute Torque shown in Fig. 7. The contents of its subsystem Compute Offset are shown in Figs. 8 and 9; the contents of Compute Torque's ancestor systems (its context) are not shown. The names of the blocks in the models are obfuscated for confidentiality purposes.

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 *method* is needed to enforce good design practice at the level of interfaces. The systematic use of signatures provides just such a method.

Prescriptive signatures As an example, imagine a scenario where a model is under development, and a new subsystem needs to be created. Since the context of this subsystem is 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 to create the desired strong signature for the new 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.

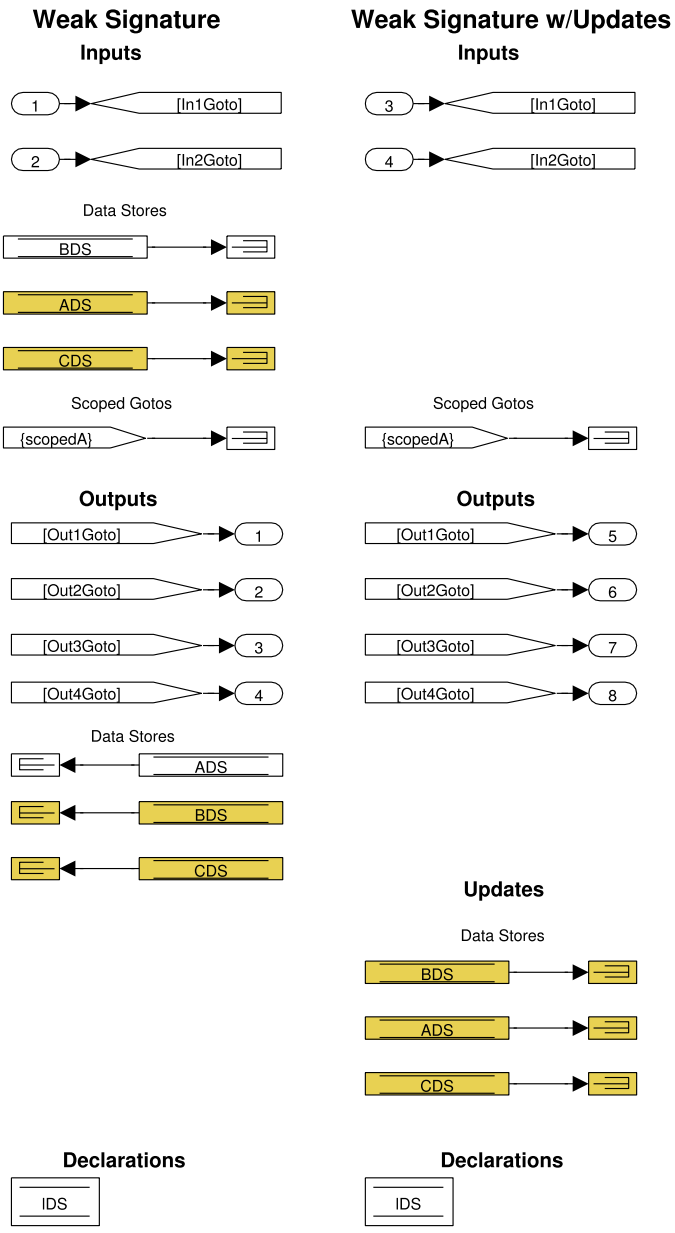


Fig. 6. Weak signature and weak signature with common I/Os shown in Updates for Subsystem1.

Classifying inputs Making the implicit interface visible empowers developers to more effectively use data stores and scoped tags. We can use these features much more effectively when they are easy to identify. For example, consider 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, 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

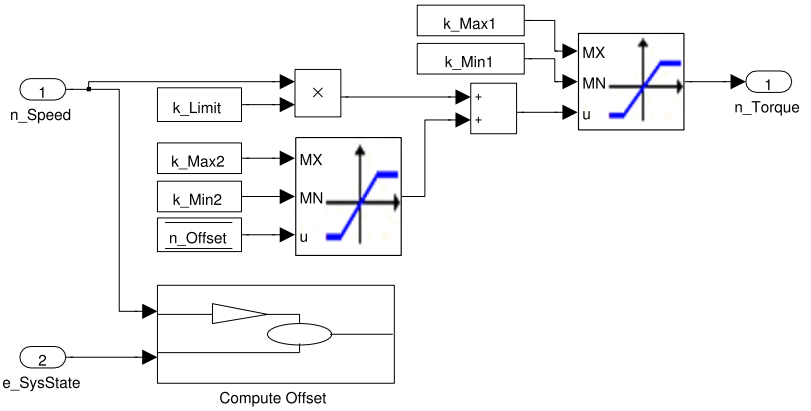


Fig. 7. Compute Torque subsystem.

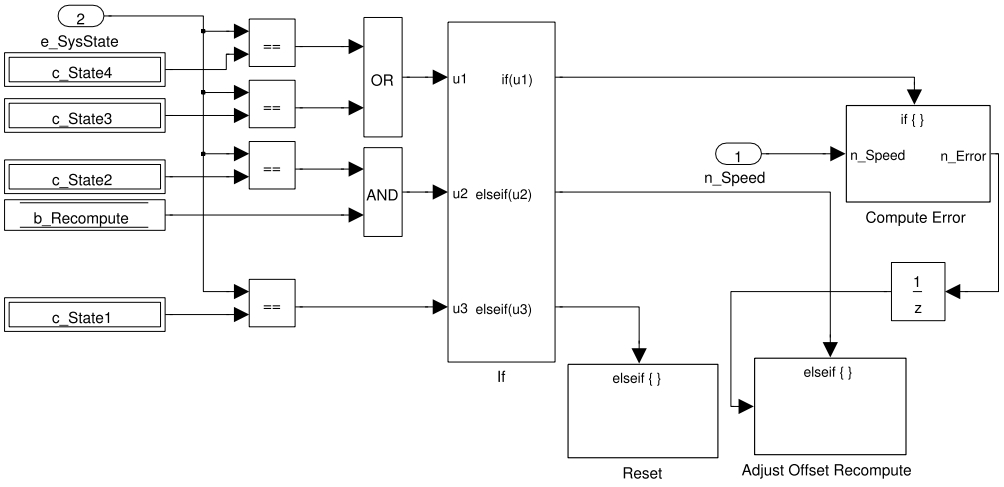


Fig. 8. Compute Offset subsystem of Fig. 7.

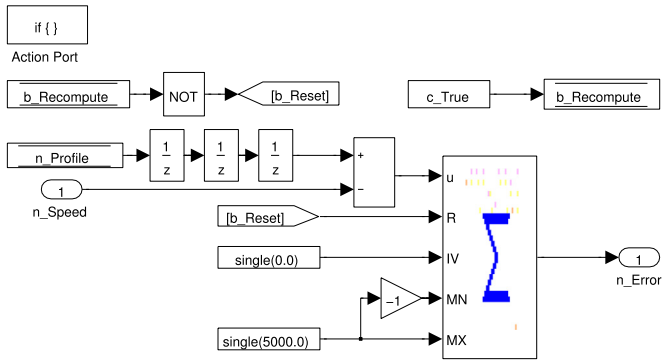
identify whether tags and data stores are read from or written to. Finally, the actual signature for the subsystem would be somewhere in between, restricting access to context as fits the situation. Just because a tag is not read in a subsystem does not mean that it cannot (should not) be read.

From an information hiding point of view [6], 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.

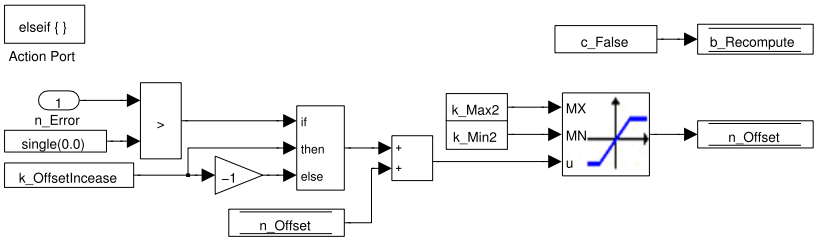
An interesting usage of signatures in the context of refactoring is *rescoping*. In the course of examining some large-scale industrial Simulink models, we came across a situation where many data stores were being defined at, or near, the top-level of the subsystem hierarchy. This essentially amounts 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. This application of signatures is discussed further in Section 5.

Documenting software Software documentation is seldom given the importance it deserves. It is a highly neglected topic in both industry and academia today. Software documentation is often poorly written, incomplete, or inconsistent, in part because of the additional expense and manual effort required to create and maintain it. Even if the original software has adequate related documentation, schedule pressures often result in the documentation not being kept up-to-date. While Simulink models are sometimes described as "self-documenting", we have found that in practice, for large industrial systems, additional documentation is needed and can be as important as the model itself to the developers. Producing proper software documentation does not come without challenges: it is not easy and it is resource-intensive, and, therefore, expensive.

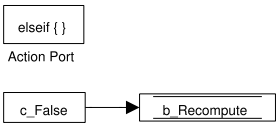
Given those challenges, an automation of even a part of documenting process may prove very cost-effective. Signatures can be used to automatically document the complete interfaces of subsystems and can be automatically extracted into an external document that serves as a part of the design description. The Signature Tool is capable of extracting strong and



(a) Compute Error subsystem.



(b) Adjust Offset Recompute subsystem.



(c) Reset Subsystem.

Fig. 9. Subsystems of Fig. 8.

```
INPUTS
Inports: n_Speed, e_SysState
Data Store Reads: n_Offset, b_Recompute, n_Profile

OUTPUTS
Outports: n_Torque
Data Store Writes: n_Offset, b_Recompute
```

Fig. 10. Strong signature of Compute Torque subsystem exported to a text file.

weak signatures of a subsystem into a separate document. For example, the strong signature of the subsystem Compute Torque from Fig. 7 can be extracted into a text file as shown in Fig. 10.

Our industrial partner recognized the importance of the automatic production of subsystem interface specifications. The automation creates the documentation at virtually no cost, and allows it to be automatically kept up-to-date as the models themselves change. The initial prototype of our tool has been used by our industrial partner in their development and maintenance processes to insert subsystems interface specifications into design documentation.

4.2. Concrete application of signatures

As illustrated in Section 3, we can actually express the signature in Simulink and include it in the subsystem itself. For the industrial model from Fig. 7, its strong signature is extracted and included within the model, as shown in Fig. 11.

Data flow view The signature presented in Fig. 11 effectively augments the subsystem 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 signature effectively provides a *data flow view*, as defined by Quante [12]. In fact, in Quante's definition 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. For example, for the subsystem from Fig. 7, in order to understand the subsystem's data flow through inherited data stores, a developer has to manually investigate five subsystems (all subsystems from Figs. 7, 8, and 9), which is a time-consuming and error-prone process, even for the subsystems that are not overly complex such as the one from Fig. 7. Instead, the signature on the left of Fig. 11 explicitly identifies data store reads and data store writes.

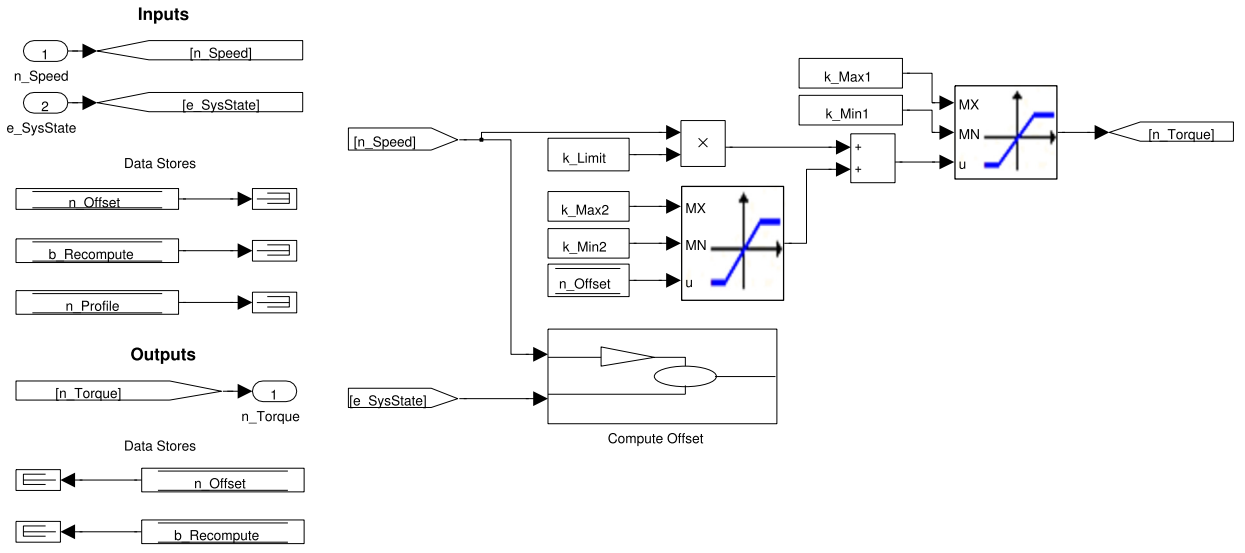


Fig. 11. Subsystem Compute Torque, with signature.

We are undertaking a study to demonstrate improved readability and comprehensibility of models when signatures are included — see Section 6 for details.

Automatic extraction of signatures As previously mentioned, we have developed a tool, the *Signature Tool*,[3] capable of automatically computing the weak and strong signatures of a Simulink model. The tool enables signature extraction on all subsystems of a model, or only user-specified subsystems. The weak signature is implemented in a recursive top-down algorithm on the system tree of a model, while the strong signature is implemented by a bottom-up recursive algorithm. The tool functionality also includes the option to generate the alternative view of signatures containing updates. Additionally, as stated earlier, extraction of signatures to external formats has significant benefits, so this functionality is now also included in the tool.

Since its prototype version described in [9], the tool has progressed to a more robust and mature stage, and has been used on large, real-world automotive examples. As a demonstration of its effectiveness, the tool was run on several substantial Simulink models each implementing sizeable vehicle-level features. The models were comprised of 800 blocks on average, with a hierarchical depth of 6, and each subsystem in a model was visited and instrumented with a signature interface. The signature extraction function runtime never exceeded one minute, even for the largest of models.

It is evident from our use, as well as in the view of our industrial partner, that this tool has the potential to save significant developer time and effort.

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

There is a significant lack of proper consideration of implicit data flow in modern testing tools. More precisely, when generating testing harnesses for a subsystem, some testing tools do not deal with data passed in/out of the subsystem by data store reads/writes. We looked into two major commercial automatic test generation tools[4] 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/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, and then generates input test data for data store reads: 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 mishandles it by treating 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.

[3] Available for download at http://www.mathworks.com/matlabcentral/fileexchange/49897-signature-tool.
[4] Educational licensing terms of these tools do not allow us to publish the names of the tools.

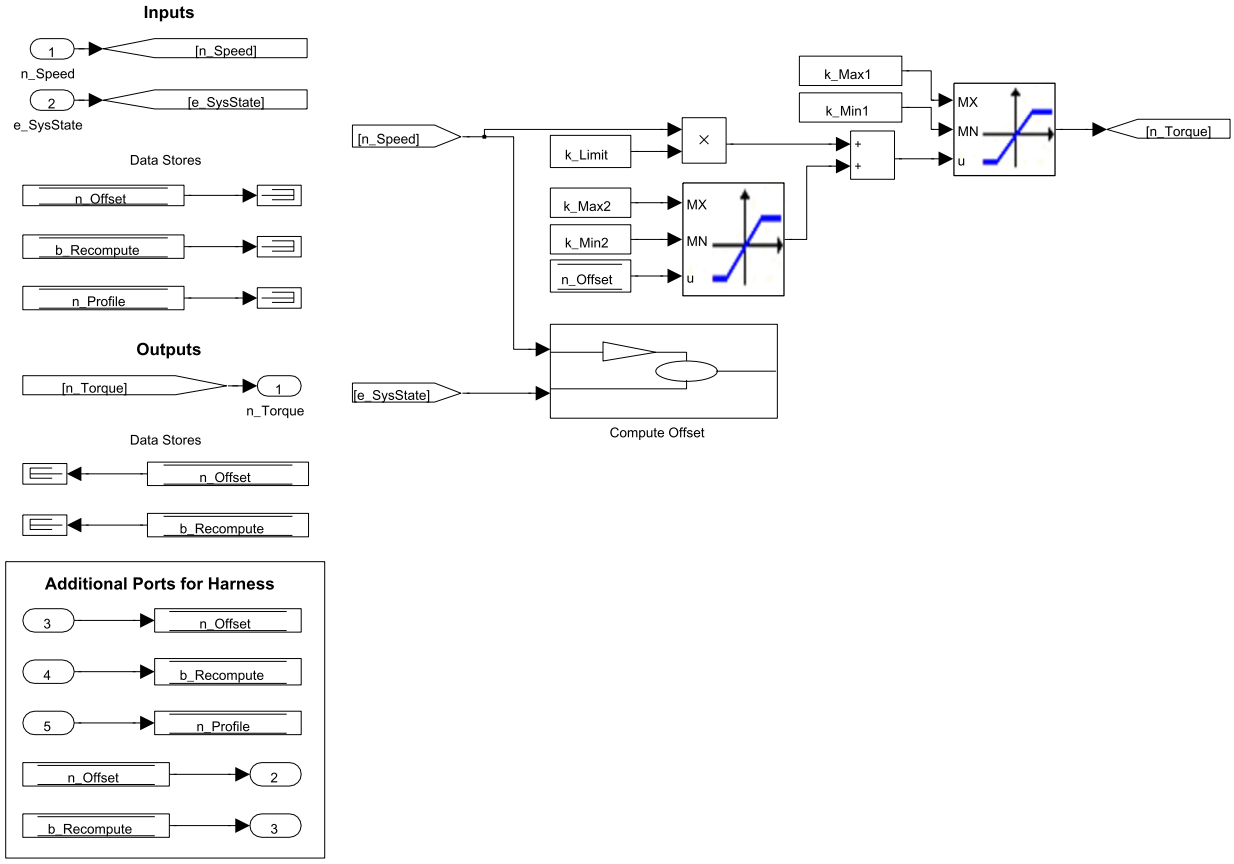


Fig. 12. Signatures help identify additional inports (3, 4, and 5), and outports (2 and 3) for unit testing of subsystem Compute Torque.

The strong 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. We will now demonstrate how signatures can help generate test harnesses that properly account for the implicit data flow. The aforementioned popular testing tool that does not account for the implicit data flow will be utilized to test subsystem Compute Torque from Fig. 7.

The testing tool is first used to extract the subsystem Compute Torque for testing: the tool extracts the subsystem itself, together with the subsystem's relevant context, including e.g., the triggering mechanism of its parent subsystem, and declarations of the inherited data stores that Compute Torque references. The context extracted by the tool is contained above the subsystem in the hierarchy, and is not shown here as it is not relevant for the discussion. The subsystem extracted by the tool for harnessing is exactly the one shown in Fig. 7. The subsystem contains two inputs and one output for testing, corresponding to the subsystem's inports and outports respectively, therefore not including the data flow through references to data stores n_Offset, b_Recompute, and n_Profile. In order to properly account for the implicit data flow through the subsystem via the data stores, the following process is proposed:

1. The subsystem's strong signature is included in the subsystem (Fig. 11).
2. The testing tool is used to extract the subsystem, including its context.
3. The Signature Tool is used to automatically add inports and outports to the extracted subsystem to mimic the data flow via data stores. Furthermore, the inports corresponding to data store reads inherit the type of corresponding data stores so that the test generation is properly focused only on valid data. Fig. 12 shows five additional ports of subsystem Compute Torque added for the purpose of unit testing, namely inports 3, 4, and 5, and outports 2 and 3. Although not used in this example, scoped goto/froms are handled by the tool in a similar manner: data flow through a from is mimicked with an inport fed into a goto block with the same name, and data flow through a goto is observed by feeding it into an outport.
4. The testing tool is used to automatically generate test cases on the augmented model in order to maximize testing coverage in a number of test metrics.

Therefore, the signature can be used for 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

Table 1
Comparing coverage and testing effort for harness generated by testing tool vs. signature-extended harness.

	Harness (H)	Extended Harness (EH)	Improvement ($\frac{\lvert EH-H\rvert}{H} \times 100\%$)
Test steps	1165	392	66%
Branch	88% (58/66)	100% (66/66)	14%
Decision	100% (34/34)	100% (34/34)	0%
MC/DC	95% (19/20)	95% (19/20)	0%
Boundary	100% (15/15)	100% (21/21)	0%
Total	95% (126/135)	99% (140/141)	4%

of the subsystem, in that it appears entirely on the left bottom part of the model (see Fig. 12). The main benefit of this separation 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.

Table 1 shows the resulting test coverage and testing effort (the number of test steps) for subsystem Compute Torque for two cases: 1) the harness is generated by the testing tool only, and 2) the harness is generated by the testing tool and augmented using the Signature Tool. The table shows that the testing effort decreases substantially with improved coverage in the case of the extended harness.

Finally, adding the strong typing information to the signature (as presented next in this section) further enhances testability in that the typing information can be used to focus test generation only to data of interest — the set of possible values.

Typing Inclusion of the signature in a subsystem has another major benefit: we can use the patterns presented by Rau in [3] 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 this approach. First, the application of his design patterns presents overhead for developers since they essentially need to be programmed by hand. Second, this 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 also quite clean, as it is completely contained within the signature and thus does not visually affect the rest of the subsystem.

Behavior preservation An obvious question which arises when including signatures in a subsystem is whether we have changed the behavior of that subsystem. To show that the behavior 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 [13]: if name-binding, control flow, and data flow are preserved, then we have a strong argument that behavior has been preserved. By simply analyzing the patterns in the signature, we can make a simple argument that we have not changed the behavior 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 behavior. 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 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 are concerned, we are simply 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 an automatic test generation tool[5] to perform model-based coverage testing of a set of subsystems before and after signature inclusion. The results were identical in all cases.[6]

We also generated code before and after signature extraction for the examples; the code was identical (with some cosmetic changes).

Nevertheless, we intend to further explore the comparison of generated code before and after signature extraction, for larger models.

[5] Educational licensing terms of the tool do not allow us to publish the name of the tool.

[6] If, however, the included signature is used to augment test harnesses to properly account for implicit data flow through the subsystems, test coverage increases as shown previously in this section.

5. Signature metric

Understandability, maintainability, and testability of software are subjective qualities of software systems, and as such, have often been evaluated based on one's engineering judgment. In order to make the determination of these qualities more objective, a number of metrics have been proposed as a means of quantifying these subjective notions.

Metrics for Simulink/Stateflow models are typically general code metrics that have been suitably adapted for this development environment. For example, branch coverage, which is a test coverage metric defined for code, corresponds to a decision coverage metric for Simulink models, where decision points like switch blocks and MinMax blocks are considered. Likewise, the most cited and frequently used software complexity metrics, cyclomatic complexity [14] and Halstead metrics [15], have been adapted in a similar fashion. However, neither cyclomatic complexity nor Halstead metrics are sensitive to the degree of cohesion and coupling that is present in hierarchical designs. Low coupling and high cohesion are two major goals in the design process, and there are some metrics in the literature (see [16]) which strive to quantify these qualities. For Simulink blocks, [16] defines a coupling metric as the number of blocks providing input signals to a block, with another metric defined for outputs in a similar manner.

Although these metrics are beneficial quality measures in their own right, they fail to consider the implicit data flow signals discussed in this paper. Dependencies between two blocks are discovered by identifying incoming and outgoing signals connecting the two; however, the data flow mechanisms discussed in Section 2 are not taken into account. This is another indication of how seriously implicit data flow is neglected in general during the analysis of Simulink/Stateflow models. We again highlight that signatures overcome this problem by presenting the implicit interface explicitly so that it is not the case that any data flow is overlooked during analysis.

When it comes to software metrics in general, we feel that no metric on its own, or in combination with others, can exactly quantify good modularization or provide an accurate indication of testability and maintainability. However, we do believe that some metrics can be used as helpful indicators of good modularization. Therefore, in an attempt to better quantify coupling and cohesion in Simulink/Stateflow designs, we propose a metric based upon the difference between weak and strong signatures of a subsystem as a gauge of the quality of design in terms of modularity. Intuitively, a large difference in size between a subsystem's weak and strong signatures indicates that the subsystem has access to many data values that it is not actually using. As a consequence, this increases the potential for a developer to create a new dependency between subsystems that was not present in the original system design, access a data value incorrectly, or inadvertently interfere with another subsystem in the hierarchy. We now formally define the *signature metric*.

Definition 5.1 *(Signature metric).* Let $Sig^w(S)$ and $Sig^s(S)$ be the weak and strong signature, respectively, of a valid subsystem S. The *signature metric* for the subsystem S, $d(S)$, is defined as:

$$d(S) = Size(Sig^w(S)) - Size(Sig^s(S)), \text{ where}$$
$$Size(\Sigma) = |D^I| + |D^O| + |T^I| + |T^O| \text{ for an arbitrary signature } \Sigma = ((P^I, D^I, T^I), (P^O, D^O, T^O), (D^M, T^M)).$$

The metric measures the difference in the number of implicit inputs and outputs in a subsystem's weak and strong signatures. Since, for a valid subsystem, the set of inports/outports/declarations in its weak signature is identical to the set of inports/outports/declarations in its strong signature, these are not explicitly used in the definition of the signature metric. Therefore, the metric counts how many resources a subsystem is able to access, but is not currently accessing. A large distance in the metric indicates that a large number of data stores and tags are available but are not used in the subsystem. This number can serve as an indication of poor design where, for example, a large number of data stores are declared at a much higher hierarchy level than needed, such as at the top level. This, of course, reflects a bad software engineering practice that is analogous to programming with global variables. Data stores, like variables in traditional programming languages, should be restricted in scope in order to avoid inadvertent/unwanted access, hide low-level details, and explicitly specify the scope of usage for data stores.

To address the issue of improperly scoped data stores, we developed a tool that automatically decreases the scope of data stores by "pushing" their declarations down the system hierarchy to the lowest level that still results in a valid system. Let us first illustrate the operation of the tool. Fig. 13 is a pictorial representation of the initial system, before the data store push-down (rescoping) operation is employed. The S_i nodes are systems, and the rectangular elements are blocks, where SUB_i nodes are subsystem blocks. DSM_i, DSR_i, and DSW_i nodes correspond to data store memory, data store read, and data store write blocks, respectively. R_i nodes are reference blocks (blocks that refer to a block in another file, here denoted by F_i nodes). Arrows emanating from system S_i to a block denote membership of the block in the system. An arrow emanating from a subsystem block to a system denotes that the system is associated with the subsystem; a similar relationship exists for references and files. Fig. 14 displays the system after the data store push-down operation has been applied to it.

We have used signatures to evaluate quality of design in a large industrial model. Figs. 15 and 16 show the top level of the model before and after the push-down operation, respectively. Simple visual comparison of the figures reveals that the push-down operation significantly decreases the number of data stores declared at the top level (data stores are represented by the small rectangular blocks at the bottom of the figures). Note that many details on the figures are not legible for confidentiality purposes — the figures are used merely to illustrate the change in complexity (the number of data store memory blocks) at the model's top level caused by the push-down operation.

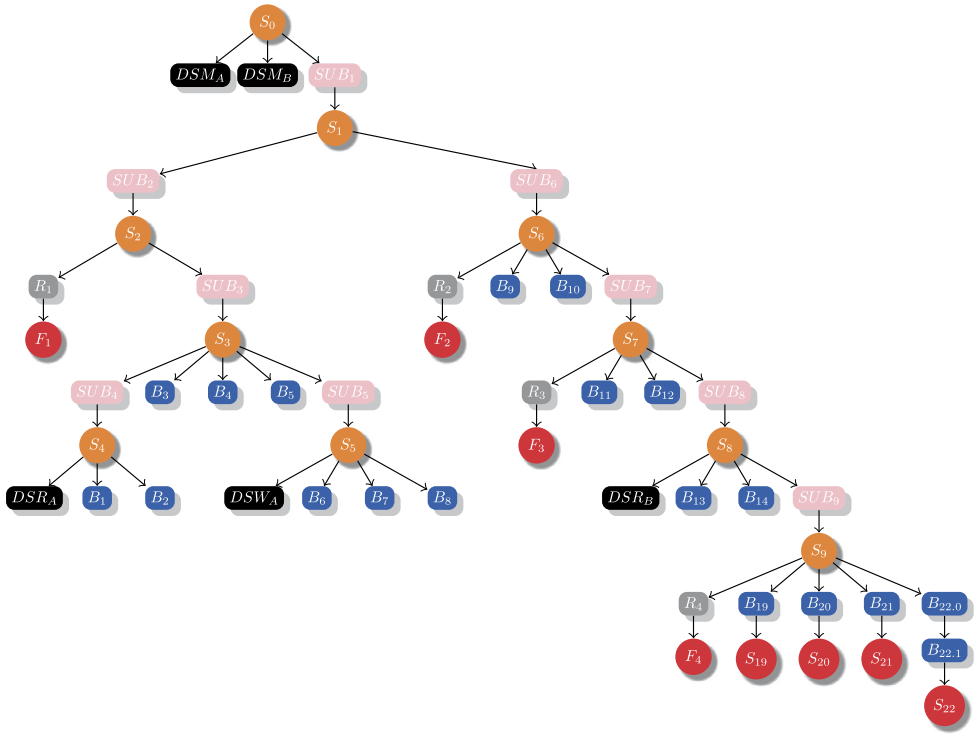


Fig. 13. Data store push-down: before.

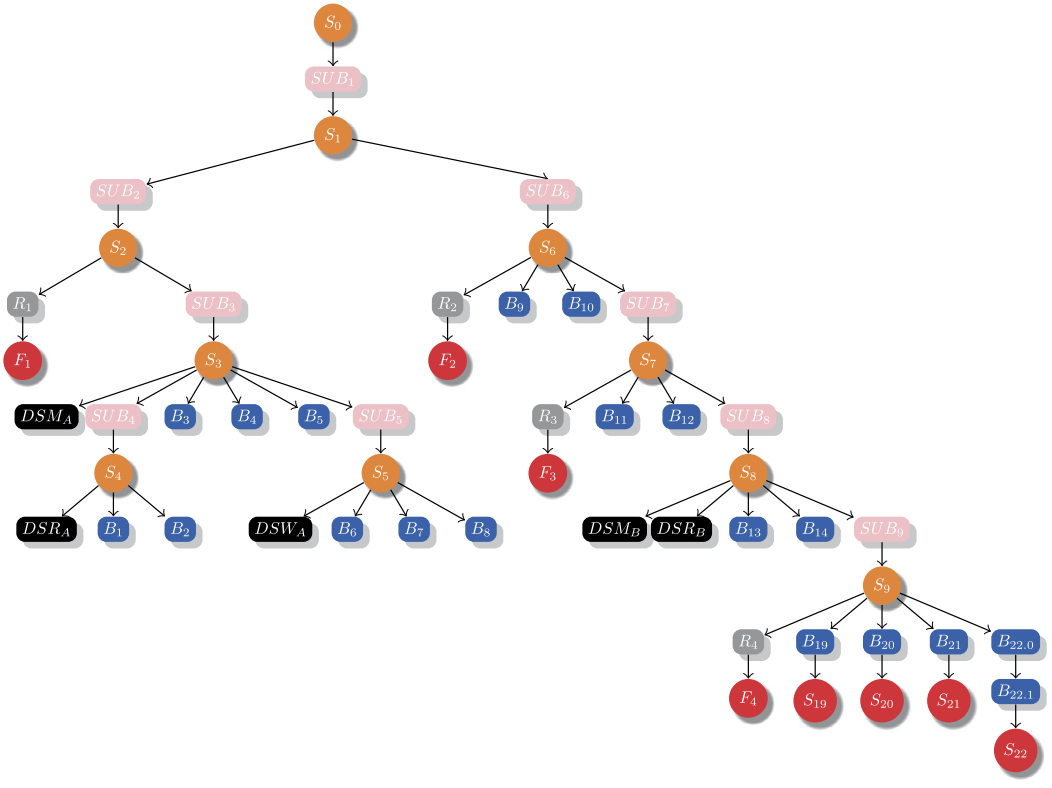


Fig. 14. Data store push-down: after.

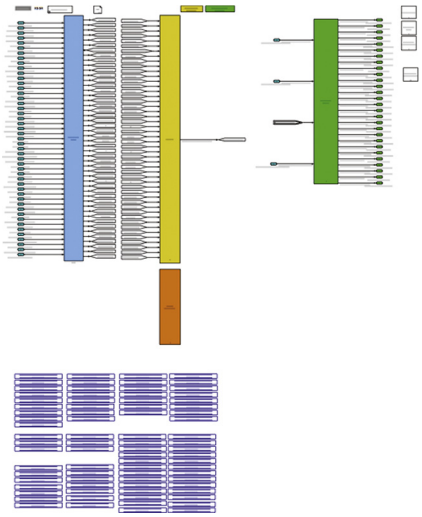


Fig. 15. The top level of the industrial model before the push-down operation.

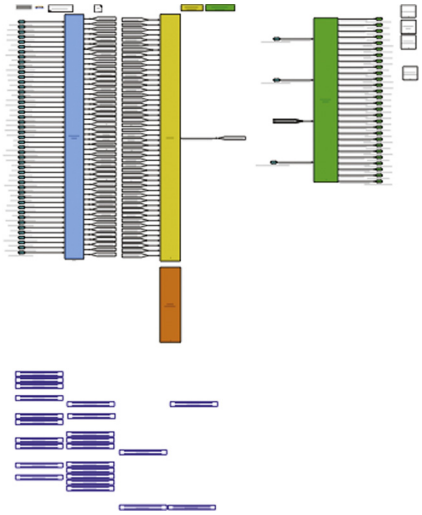


Fig. 16. The top level of the industrial model after the push-down operation.

Further, the visual inspection of the signatures of a large number of the subsystems in the industrial model before and after the push-down operation provides a clear view of cluttered implicit interfaces before the push-down operation, and how the interfaces are reduced after the push-down. However, the signature metric offers a much more effective diagnosis of a problematic design, and represents an efficient tool for the comparison of a design with its alternative representations. For instance, the signature metric is calculated for a number of subsystems in the industrial model, and the results are shown in Table 2, for both before and after the push-down operation. As noted already, this model has many data stores declared at the top-level subsystem in its hierarchy. For example, looking at the data for Sub_1[7] in Table 2, one can see that the size of the weak signature is large (182), especially when compared to the size of its strong signature (12). This indicates that there might be many data stores with scopes much higher than necessary, and therefore, these data stores can possibly be pushed down in the model hierarchy. In the case of Sub_1, the value of the metric decreases by 108 (by 63.6%) after the push-down operation. This shows the large improvement in modularity of the design. The results in Table 2 are given for subsystems found relatively high in the model's hierarchy — this is where the push-down operation has the highest impact on the signature metric. Lastly, not all the subsystems are affected by the push-down operation. More precisely, if no data stores could be pushed down into a subsystem, the sizes of weak and strong signatures, and the signature metric for the subsystem remain unchanged, as is the case for Sub_7 in Table 2. This is an indication that the subsystem is properly designed in the sense that it uses exactly the resources it has access to.

[7] Sub_1 is a subsystem located at the model's top level.

Table 2
Sizes of signatures and the metric before and after pushdown operation.

		Before pushdown (BP)	After pushdown (AP)	$BP - AP$	$\frac{BP-AP}{BP} \times 100\%$
Sub_1	$Size(Sig^W(S))$	182	74	108	59.3
	$Size(Sig^S(S))$	12	12	0	0.0
	$d(S)$	170	62	108	63.6
Sub_2	$Size(Sig^W(S))$	231	123	108	46.8
	$Size(Sig^S(S))$	155	87	68	43.9
	$d(S)$	76	36	40	52.6
Sub_3	$Size(Sig^W(S))$	156	150	6	3.9
	$Size(Sig^S(S))$	8	6	2	25.0
	$d(S)$	148	144	4	2.7
Sub_4	$Size(Sig^W(S))$	277	169	108	39.0
	$Size(Sig^S(S))$	95	95	0	0.0
	$d(S)$	182	74	108	59.3
Sub_5	$Size(Sig^W(S))$	214	196	108	50.5
	$Size(Sig^S(S))$	57	57	0	0.0
	$d(S)$	157	49	108	68.8
Sub_6	$Size(Sig^W(S))$	220	214	6	2.7
	$Size(Sig^S(S))$	70	70	0	0.0
	$d(S)$	150	144	6	4.0
Sub_7	$Size(Sig^W(S))$	104	104	0	0.0
	$Size(Sig^S(S))$	91	91	0	0.0
	$d(S)$	13	13	0	0.0

6. Conclusions and future work

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 the system's modular structure has been broken, as we have discussed in Section 3. Furthermore, the use of the proposed metric in determining modularity serves as a way of assessing and improving system design quality. We hope to have convinced the reader of the usefulness and practicality of signatures. Our industry partner has incorporated signatures into their software development process, and we will continue to collaborate closely with them in the deployment of the proposed metric.

In order to show the benefits of the proposed signature for subsystems in Simulink, we plan to perform a comprehensibility study with software engineers working for 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 [7,8]. 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, and including signatures in a comprehensive software engineering methodology currently being developed at the McMaster Centre for Software Certification (McSCert).

References

[1] B. Schätz, A. Pretschner, F. Huber, J. Philipps, Model-based development of embedded systems, in: Advances in Object-Oriented Information Systems, in: Lecture Notes in Computer Science, vol. 2426, Springer, 2002, pp. 298–311.
[2] A. Rau, Potential and challenges for model-based development in the automotive industry, in: Business Briefing: Global Automotive Manufacturing and Technology, 2000, pp. 124–138.
[3] A. Rau, On model-based development: a pattern for strong interfaces in SIMULINK, Gesellschaft für Informatik, FG 2.1. 1: Softw.tech.-Trends 22 (1) (2002) 12–19.
[4] A. Rau, Decomposition and interfaces revisited, Gesellschaft für Informatik, FG 2.1. 1: Softw.tech.-Trends 21 (2) (2001) 19–23.
[5] B. Meyer, Applying "Design by contract", Computer 25 (10) (1992) 40–51.
[6] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[7] A. Cox, S. Gauvin, T. Smedley, Towards comprehensible control flow in visual data flow languages, in: Proceedings of International Conference on Distributed Multimedia System, 2004, pp. 247–252.

[8] T. Green, M. Petre, When visual programs are harder to read than textual programs, in: Human-Computer Interaction: Tasks and Organisation, Proceedings of 6th European Conference on Cognitive Ergonomics, 1992, pp. 167–180.
[9] M. Bender, K. Laurin, M. Lawford, J. Ong, S. Postma, V. Pantelic, Signature required: making Simulink data flow and interfaces explicit, in: Proceedings of 2nd International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2014, SCITEPRESS, 2014, pp. 119–131.
[10] MathWorks, Subsystem, atomic subsystem, nonvirtual subsystem, CodeReuse subsystem, http://www.mathworks.com/help/simulink/slref/subsystem.html, 2013 [Online accessed 25-August-2013].
[11] MathWorks, Best practices for data stores, http://www.mathworks.com/support/solutions/attachment.html?resid=1-6F3I63&solution=1-5NM3AN, 2008 [Online, accessed 20-August-2013].
[12] J. Quante, Views for efficient program understanding of automotive software, Softw.tech.-Trends 33 (2) (2013) 55–56.
[13] M. Schäfer, M. Verbaere, T. Ekman, O. de Moor, Stepping stones over the refactoring Rubicon, in: S. Drossopoulou (Ed.), ECOOP 2009–Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 369–393.
[14] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 4 (1976) 308–320.
[15] M.H. Halstead, Elements of Software Science, Operating, and Programming Systems Series, vol. 7, Elsevier, New York, NY, 1977.
[16] M. Olszewska, Simulink-specific design quality metrics, Tech. Rep. 1002, Turku Centre for Computer Science, 2011.