Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-017-0450-9

REGULAR PAPER

Software engineering practices and Simulink: bridging the gap

Vera Pantelic[1] · Steven Postma[1] · Mark Lawford[1] · Monika Jaskolka[1] · Bennett Mackenzie[1] · Alexandre Korobkine[1] · Marc Bender[1] · Jeff Ong[1] · Gordon Marks[1] · Alan Wassyng[1]

Abstract Although widely used in embedded systems design, Matlab/Simulink is not considered a state-of-the-art design environment by the software engineering community. This paper is aimed at improving design with Simulink from the software engineering perspective by developing automated support for the application of some traditional software engineering principles when developing with Simulink. We present four tools: the *Signature Tool*, the *Reach/Coreach Tool*, the *Data Store Rescope Tool*, and the *Auto Layout Tool*. The Signature Tool extracts the interface of a Simulink subsystem, enabling developers to better understand the implicit data flow in Simulink models and use it more effectively, while also producing useful documentation. The Data Store Rescope Tool improves modularity of Simulink models by properly scoping *data stores*, the Simulink equivalent of variables in traditional languages. The Reach/Coreach Tool highlights data and control dependencies in Simulink models, making them easier to understand. Also, the tool supports debugging, reverse-engineering, refactoring, and static analysis of the models. Finally, the Auto Layout Tool automatically improves the layout of Simulink models, reducing the effort developers invest in graphical layout to comply with modeling guidelines and improve readability of their models.

Keywords Model-based development · Simulink · Tools · Software engineering · Refactoring · Model transformation · Data flow · Model slicing

✉ Vera Pantelic
pantelv@mcmaster.ca

Steven Postma
steven.m.postma@gmail.com

Mark Lawford
lawford@mcmaster.ca

Monika Jaskolka
bialym2@mcmaster.ca

Bennett Mackenzie
mackeb@mcmaster.ca

Alexandre Korobkine
korobkao@gmail.com

Marc Bender
marc.mb.bender@gmail.com

Jeff Ong
ongjf2@mcmaster.ca

Gordon Marks
marksgw@mcmaster.ca

Alan Wassyng
wassyng@mcmaster.ca

[1] McMaster Centre for Software Certification, Department of Computing and Software, McMaster University, Hamilton, ON, Canada

1 Introduction

Matlab/Simulink is a model-based design environment widely used in embedded systems development in numerous domains, including automotive. Its popularity can be attributed to its rich modeling and simulation capabilities, automatic code generation, and availability of a large number of MathWorks and third-party tools that aid development within the environment (e.g., MathWorks' Control System Toolbox). While a major part of the code base running in modern cars is automatically generated from Simulink, the environment still lacks proper support for the application of some traditional software engineering practices. For example, *data stores*, Simulink's analog of variables in traditional programming languages, cannot be declared *read-only*. Furthermore, Simulink lacks *self-documenting* capabilities of

imperative programming languages. For instance, an analog of a module interface in C, as defined in C header files, does not exist in Simulink.[1]

The main contribution of this work is the introduction of a set of tools which help developers apply some well-known software engineering principles in development with Simulink. Proper automatic support is crucial for the successful application of these principles [10]. We also note that it is not our intention to undertake a comprehensive analysis of how each software engineering principle is supported in Simulink, neither do we compare the model-based design of Simulink to more traditional development paradigms. Rather, we draw on our experience from working with production-scale Simulink models to suggest mechanisms and appropriate tool support to improve the application of some well-known software engineering practices and principles in development with Simulink. The toolset to be presented in this paper helps a developer adhere to some well-known software engineering practices in an automated manner, allowing for the proposed principles to be seamlessly integrated into an existing Simulink model-based software development process. We introduce four tools: the *Signature Tool*, the *Reach/Coreach Tool*, the *Data Store Rescope Tool*, and the *Auto Layout Tool*. The tools were developed as a result of collaboration with an industrial partner, a large automotive OEM (Original Equipment Manufacturer); however, the tools are not automotive-specific but general in that they can be applied in any model-based design with Simulink.

The Signature Tool extracts a Simulink subsystem's *signature*. A signature is a representation of a subsystem's interface [2,3]. Signatures identify implicit data flow mechanisms in Simulink, enabling their more effective use in design, analysis, and testing. Furthermore, signatures can be used as an indication of the quality of modularization of a Simulink design. Finally, the tool can be used to automatically generate parts of software documentation: our automotive industrial partner uses it to automatically document subsystem interfaces as part of a subsystem's *software design description*.

The Reach/Coreach Tool highlights, for the specified Simulink blocks, the parts of the model which are affected by the specified blocks (*Reach* functionality), or parts of the model that the specified blocks depend upon (*Coreach* functionality). The tool accounts for both data and control dependencies. To the best of our knowledge, there exist two other tools for slicing Simulink models that take into account both data and control dependencies: the tool presented in [18], and MathWorks' *Model Slicer* (available since Matlab R2015a as part of Simulink Design Verifier (SDV)). In Sect. 3.2.3, a detailed comparison between our tool and the two aforementioned tools is presented.

The Data Store Rescope Tool (formerly the Data Store Push-Down Tool) properly scopes Simulink's *data stores*. Data stores are analogous to variables in traditional programming languages, and algorithms exist for localizing their usage in languages such as C [20,21]. Our tool identifies the data stores that have scopes larger than necessary. Then, for each identified data store, its declaration is "pushed down" the model hierarchy to the lowest level possible, such that all the references to the data store are still within the data store's scope. The push-down operation improves the modularity of Simulink designs. The tool was introduced in [3] as the Data Store Push-Down Tool and was also used in [3] to illustrate the effectiveness of a software complexity metric introduced in that paper.

Finally, the fourth tool to be presented in this paper focuses on increasing readability of Simulink models by improving their layout. Surprisingly, there does not exist a comprehensive commercial automatic layout tool for Simulink models. While novel Simulink automatic layout algorithms have been proposed (e.g., [7,11,12]), no tools based on these algorithms are available for use in Simulink. Our approach with the Auto Layout Tool reuses a third-party graph drawing algorithm implemented in the open-source tool *Graphviz*, and further builds on it to address Simulink-specific layout requirements, as well as to automate some useful transformations valuable in every-day design with Simulink (flattening a subsystem, transforming a signal line into a Goto/From connection and vice versa, *etc*.).

This paper is an extended version of the conference paper [14]. The following contributions have been added to the work presented in [14]. Firstly, an industrial automotive model is used to illustrate the application and benefits of each of the four tools. The use of the real-world industrial example demonstrates the diversity of the tools' applications in a model-based development process, and more importantly, it provides evidence of the tools' applicability and practicality in an industrial setting. Additionally, a number of features have been added to the tools, and the current paper presents their applications and benefits in detail:

- While the main idea behind the Signature Tool is to identify the data items that are contained within the interface of a Simulink subsystem, the current version of the tool enhances this interface by including the data types of each of the data items contained in the signature.
- The Data Store Rescope Tool now includes the *Data Store Repair* transformation: when one or more references to a data store are not within its scope, the data store is properly rescoped. Further, the tool now generates information about the results of the its operations. More precisely, for each data store that was moved, its

[1] It might seem that the input and output signals of a block in Simulink are obvious from the block diagram; however, there is hidden data flow in Simulink not evident on the diagram as will be discussed in Sect. 2.

destination and source block are listed, as well as the total number of moved data stores in the model. Also, this paper demonstrates the application of the tool in the detection of data stores that are declared, but unused.
- When it comes to the Reach/Coreach Tool, the common assumption for If/Switch Case blocks that all inputs affect all outputs has now been removed. Consequently, the tool has been enhanced to identify the fine-grained dependencies between the blocks' inputs and outputs. While the improvement is conceptually rather simple, it has been shown to provide some subtle static analysis of real-world industrial models, as will be illustrated in this paper. Furthermore, the Reach/Coreach Tool is shown to provide an additional data view whose application is demonstrated when the tool is used as an *impact analysis* tool. Also, starting from Matlab R2015a, *Simulink Design Verifier*, a MathWorks' Matlab/Simulink toolbox, includes *Model Slicer* which has capabilities similar to those of the Reach/Coreach Tool. In this paper, we present a detailed comparison between the capabilities of the Reach/Coreach Tool and Model Slicer.
- The Auto Layout Tool has been significantly extended. Firstly, it now supports a number of minor, yet very practical transformations: transforming a From/Goto connection to a signal connection (and back), as well as flattening of a model's hierarchy. Secondly, the layout engine itself has been significantly enhanced to increase model readability, while following some of the best practices for layout of Simulink models as prescribed by the MAAB guidelines [28].

Also, it is worth noting that all four tools have been significantly improved in terms of usability. While they were initially Matlab command line functions, they are now invoked directly from the context menu. Furthermore, the Signature Tool and the Reach/Coreach Tool now feature simple graphical user interfaces. All the tools are available through Matlab Central, under open-source licenses [1,5,16,19].

All the tools have been successfully used on large industrial automotive models: the models contained between 3671 and 73044 blocks, with hierarchy depths between 8 and 16. However, for simplicity of exposition, smaller examples have been chosen in this paper to illustrate the tools' capabilities.

The outline of this paper is as follows. Section 2 analyses data flow in Simulink and presents an industrial model that will be used throughout the paper to demonstrate the application of the tools in an industrial setting. The Signature Tool, the Reach/Coreach Tool, the Data Store Rescope Tool, and the Auto Layout Tool are presented in Sects. 3.1, 3.2, 3.3, and 3.4, respectively. For each of these tools, we explain and illustrate their main capabilities, implementation, and possible applications in model-based design with Simulink. We also present the execution times of the tools when run on various real-world industrial automotive models. The tools were run on a Windows 7, Intel i3-2120 @ 3.30 GHz, 8.0 GB RAM machine. Section 4 summarizes the role of the tools in model-based development processes and their link with software engineering principles. Section 5 concludes the paper, with avenues for future work.

2 Data flow in Simulink and example

This section first presents an analysis of data flow in Simulink, which will serve as the necessary background knowledge for a reader to understand the tools' capabilities presented in Sect. 3. Afterward, a real-world industrial model is presented. This model will be used throughout the paper to illustrate the applicability of our tools in an industrial setting.

2.1 Analysis of data flow in Simulink

The notion of a *subsystem* is used in Simulink to represent systems inside systems in order to provide hierarchical modeling. A Simulink subsystem has inports and outports—explicit links to and from the subsystem, respectively. We view inports and outports as the *explicit interface* of the subsystem. However, there are *hidden data dependencies* in the Simulink's subsystem: we will refer to those as the subsystem's *implicit interface*. Hidden dependencies stem from two particular Simulink data mechanisms: *data stores* and Goto/From blocks.

Data stores Data stores are used in Simulink as memory and are analogous to variables in traditional programming languages. Data stores allow subsystems and referenced models[2] to share data without having to use inports and outports to pass the data from subsystem to subsystem, or level to level. A data store can be defined in Simulink using a Data Store Memory block. The data store is then referenced using Data Store Read blocks (for reading from the data store) or Data Store Write blocks (for writing into the data store). The scope of a data store is the subsystem where the Data Store Memory block is located, and all the subsystems below it in the model hierarchy, excluding referenced models. In Fig. 1, data store B is defined using Data Store Memory B block. The scope of this data store is the top level of the model and subsystem Sub1. The data store is accessed using Data Store Write B at the same level where it is defined, and it is read using Data Store Read B in subsystem Sub1 (the contents of subsystem Sub1 are shown in Fig. 2). Also, a data store can be implemented in the base workspace using a signal object: it is then called a *global data store* as it can be accessed from anywhere in the model, including referenced models.

[2] A referenced model is a model referenced from another model using a Model block.

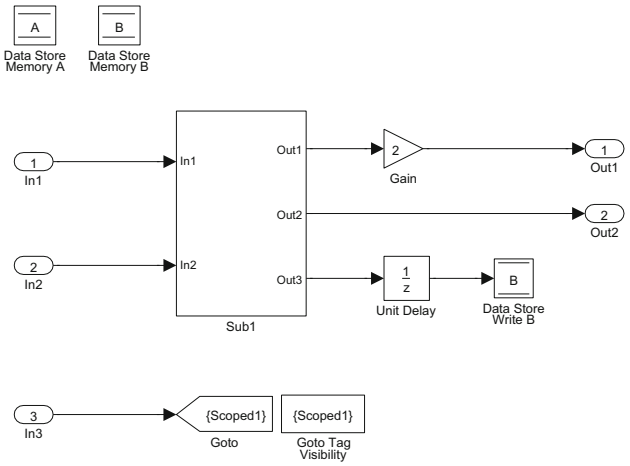


Fig. 1 A Simulink model

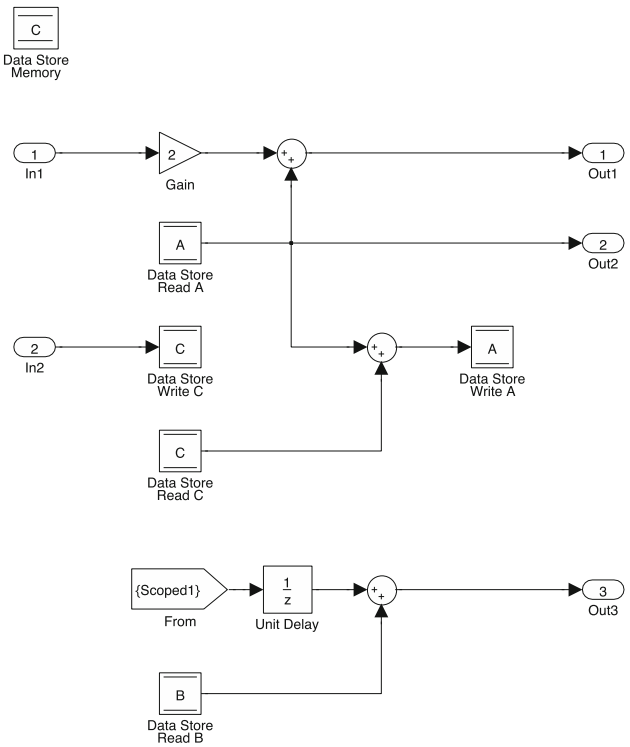


Fig. 2 Subsystem Sub1 from Fig. 1

There may also be multiple read and write blocks for a single Data Store Memory block within one simulation step. This introduces issues with the order of access to a data store (for an explanation of order of access errors and corresponding Matlab's diagnostics, an interested reader is referred to [25]).

When a data store is located higher in the hierarchy than the current subsystem, it will be referred to as an *inherited* data store for the subsystem. It should be noted that while signatures have previously been defined only for virtual subsystems [2], the current implementation of the Signature Tool is extended to support the extraction of signatures of non-virtual subsystems as well. Non-virtual subsystems are subsystems executed as a single unit (atomic execution). A virtual subsystem, on the other hand, is flattened in order to derive the block update order, with Simulink ignoring the subsystem's boundaries when determining this order. That is to say, the subsystem's boundaries do not impact the model's behavior.

Goto/From mechanism Another mechanism for implicit data flow in Simulink is the Goto/From mechanism. The data fed into a Goto block is passed to its corresponding From blocks (the From blocks with the same *tag*), without a signal line between them. Goto/From blocks are used to implicitly connect blocks, simplifying the visual presentation of models. A single Goto block may have multiple From blocks, but a From block may only receive data from a single Goto block. The scope of the Goto block is determined by the Goto block's Tag Visibility parameter, which can take on the following values:

- Local: Goto and From blocks with the same tag are in the same subsystem.
- Scoped: The scope of the Goto block is determined by the position of the corresponding Goto Tag Visibility block. The Goto block and its From blocks have to be in the same subsystem as the Goto Tag Visibility block or lower in the model hierarchy, while not crossing a non-virtual subsystem boundary, i.e., the boundary of an atomic, conditionally executed or function-call subsystem, or a model reference. In Fig. 1, scoped tag Scoped1 is defined using a Goto Tag Visibility block. This block affects the scope of the Goto block Scoped1, found to its immediate left at the same level, while the corresponding From is found in Sub1, as shown in Fig. 2.
- Global: Goto and From blocks with the same tag can be anywhere in the model, except in locations that span non-virtual subsystem boundaries.

2.2 Industrial example

This section introduces the obfuscated version of the industrial model that will be used as a running example in this paper. The original model was obfuscated for confidentiality purposes. The model is a subsystem of a larger model that itself implements a portion of the gear selection functionality in a hybrid electric vehicle. The model is shown in Fig. 3a–h.

3 Tools

This section presents the tools. For each tool, its basic capabilities are presented in detail and illustrated on a toy example. Afterward, the application of the tools is demonstrated on the industrial model. Using both the simple toy

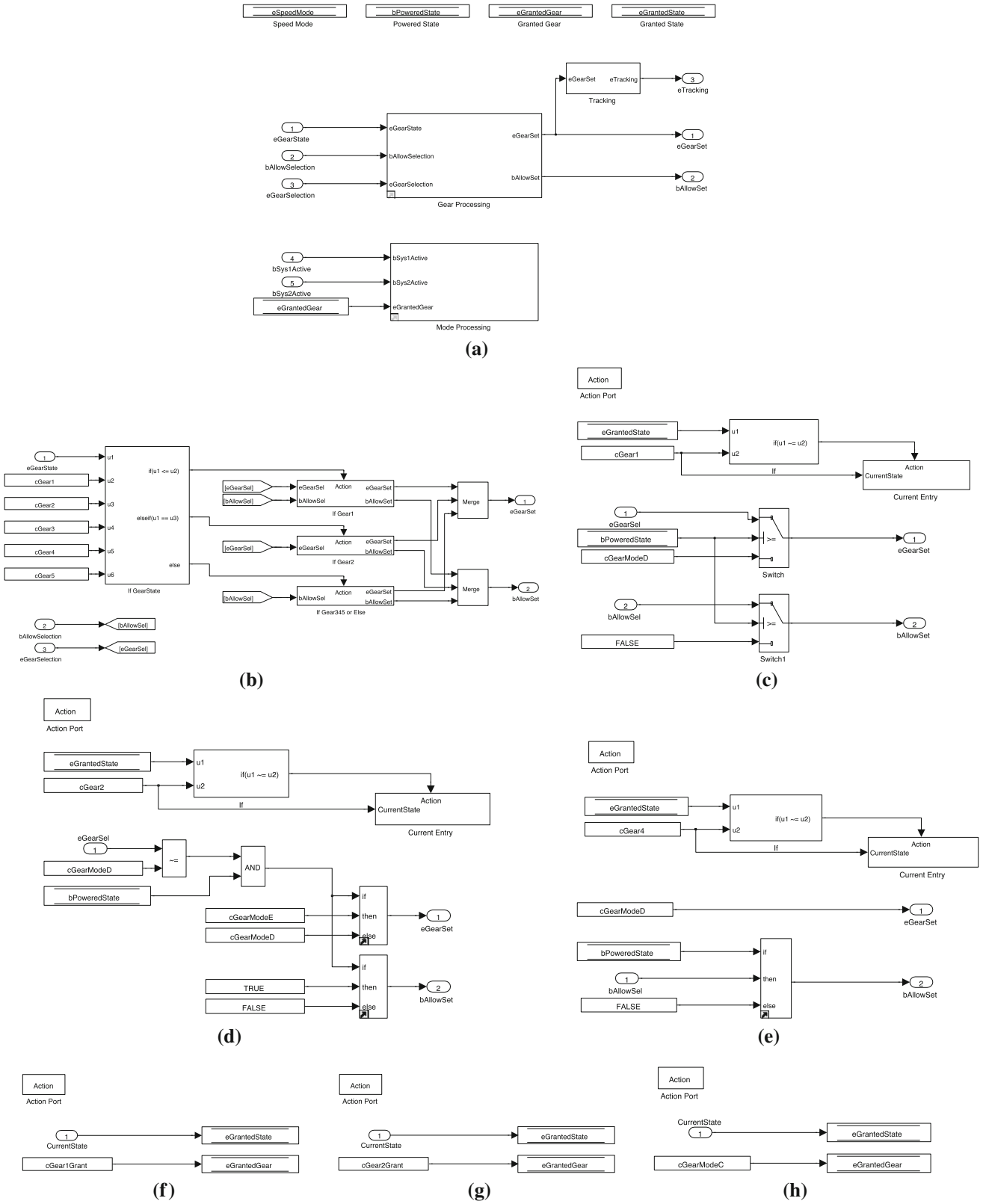


Fig. 3 Industrial example Simulink model. **a** Subsystem Gear and Mode Processing performing gear selection. **b** Subsystem Gear Processing from **a**. **c** Subsystem If Gear1 from **b**. **d** Subsystem If Gear2 from **b**. **e** Subsystem If Gear345 or Else from **b**. **f** Subsystem Current Entry from **c**. **g** Subsystem Current Entry from **d**. **h** Subsystem Current Entry from **e**

model and the industrial model enables us to first show the basic capabilities of the tools in a simple manner, and then showcase their applicability and practicality on a real-world industrial automotive model.

3.1 The Signature Tool

The Signature Tool extracts the *signature* of a Simulink subsystem. The concept of signatures was first introduced in [2]. A signature represents the *interface* of a Simulink subsystem by making all of the data flow into and out of the subsystem explicit.

Further building on this work, we now view global From/Gotos and global data stores as special cases of scoped From/Gotos and data stores. This is due to the fact that global tags can be replaced by scoped tags, with a corresponding visibility tag placed in the top-level subsystem. Likewise, global data stores can be replaced by normal data stores, with the corresponding declaration moved to the top-level subsystem.

3.1.1 Signature

The signature identifies the elements of the interface for a given Simulink subsystem:

- *Explicit interface*: Inports and outports,
- *Implicit interface*: Inherited data stores, and scoped tags defined higher up in the model hierarchy,
- *Imposed interface*: Data stores and scoped tags defined in the subsystem

The Signature Tool identifies two useful signatures for a subsystem: the *strong signature* and the *weak signature*. In general, the strong signature identifies the data mechanisms that *are accessed* by the subsystem or any of its children. The weak signature identifies the data mechanisms that a subsystem *can access* (those which are declared locally or higher up in the hierarchy), but is not necessarily using. 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. The signature approach for Simulink provides data flow analysis in a setting where semantics are not available.

More precisely, the strong signature of a subsystem contains:

- *Inputs:*
  - Inports
  - Implicit Inputs:
    - All inherited data stores that are *only read* from in the subsystem or any of its children.
    - Scoped tags with Goto Tag Visibility defined higher in the model hierarchy that have the corresponding From block located in the subsystem or in any of its children, unless the Goto block is also in the subsystem or any of its children.
- *Outputs:*
  - Outports
  - Implicit Outputs:
    - All inherited data stores that are *only written* to in the subsystem or any of its children.
    - Scoped tags with Goto Tag Visibility located higher in the model hierarchy, that have the corresponding Goto block located in the subsystem or any of its children.
- *Updates*: All inherited data stores that are both read from and written to in the subsystem or any of its children.
- *Declarations*: All data stores declared in the subsystem, and scoped tags located in the subsystem.

In the description of a strong signature's inputs, it is important to note that the rationale for excluding scoped tags with both From and Goto blocks from the inputs is due to the fact that when the Goto block is located in a subsystem or below it in the hierarchy, it is considered local, since no other subsystem can write into it.

A Simulink model and the contents of its subsystem Sub1 are shown in Figs. 1 and 2, respectively. The strong signature for subsystem Sub1 as generated by the Signature Tool is given graphically on the left side of Fig. 4. The strong signature of Sub1 specifies the data mechanisms that the subsystem or its children *access*. The subsystem Sub1 both reads from and writes to the inherited data store A; hence, the data store is included in the *Updates* set of the signature. Sub1, however, only reads from the data store B, thus the data store is in the *Inputs* set of the signature. The subsystem only reads from Scoped1 tag, and for this reason the tag belongs to the signature's *Inputs*.

The weak signature is about a subsystem's context—identifying the data mechanisms the subsystem *has access to*, but is not necessarily using:

- *Inputs:*
  - Inports
  - Implicit Inputs:
    - For virtual subsystems, scoped tags declared higher in the hierarchy with the corresponding Goto declared higher in the hierarchy

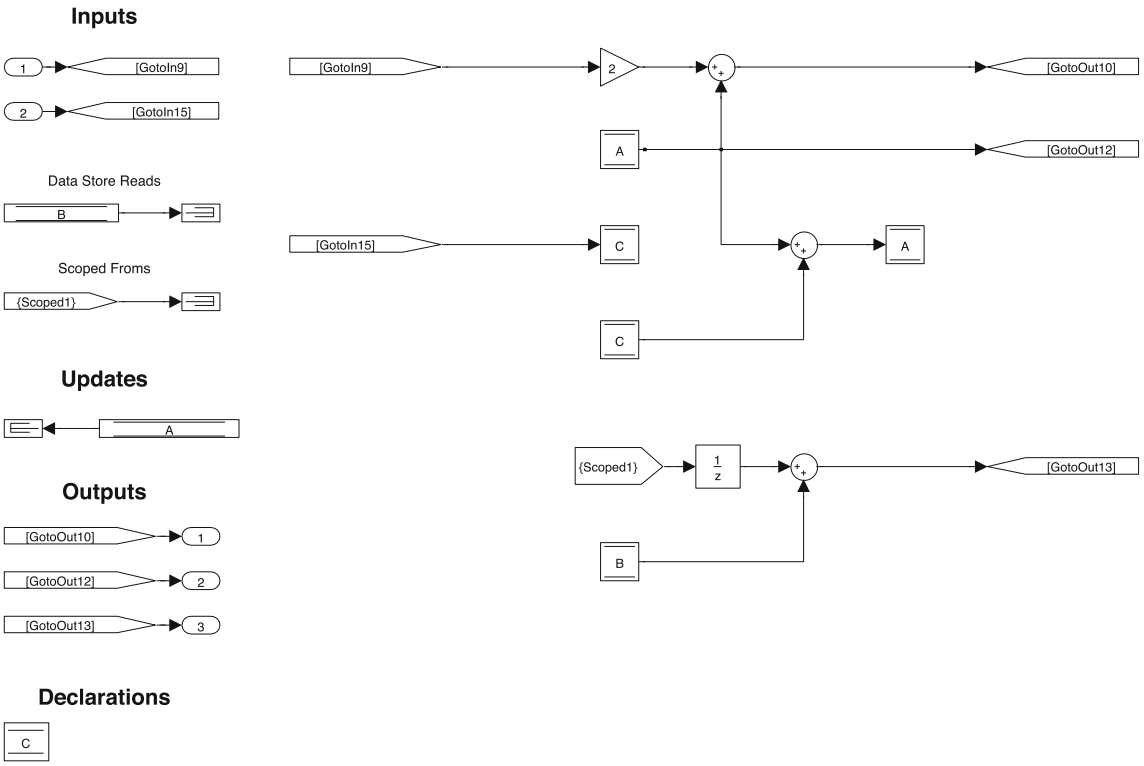


Fig. 4 Subsystem Sub1 including strong signature generated by the Signature Tool

  - For non-virtual subsystems, this set is empty since From and Goto blocks cannot cross boundaries of non-virtual subsystems

– *Outputs:*

  – Outports.

– *Updates:*

  – Inherited data stores
  – Scoped tags located higher in the model hierarchy that do not have corresponding Goto declared higher in the hierarchy.

– *Declarations:* All data stores declared in the subsystem and scoped tags located in the subsystem

The weak signature of Sub1 from Fig. 2 (the subsystem of the root system in Fig. 1), as extracted by the Signature Tool, is given on the left side of Fig. 5. Since the weak signature pertains to the resources that are at Sub1's disposal, all inherited data stores (A and B) are included as *Updates* since they can be both read from and written to. Sub1 cannot write into Scoped1 since the root level model is already writing into it and, as noted earlier, there cannot be two Goto blocks for the same tag. For this reason, the tag is placed in the signature's *Inputs*.

The Signature Tool can extract a subsystem's signature (strong or weak), and then either include it in the subsystem itself, as was done in Figs. 4 and 5, or export it to external documentation in the form of a table. The tool supports virtual and non-virtual subsystems. The data items in the signature generated by the Signature Tool are presented in the following way (see Figs. 4 and 5) for the case when the signatures are included in the subsystem: Data Store Read blocks and Data Store Write blocks and scoped Froms are fed into terminators; all input ports are fed into local Gotos, and output ports are fed from local Froms (our industrial automotive partner uses this technique whenever a port needs to be used multiple times in a model). It is argued in [2] that the behavior of the subsystem does not change for either simulation or code generation purposes when the signature is included in the subsystem.

3.1.2 Implementation and applications

The Signature Tool is implemented using Matlab functions executed from either the command line or the context menu. The weak signature is implemented as a recursive top-down algorithm on the system tree of a model, while the strong signature is implemented with a bottom-up recursive algorithm [3]. The tool has been used on large industrial automotive Simulink models, each implementing a vehicle function. For models containing between 3671 and 7792

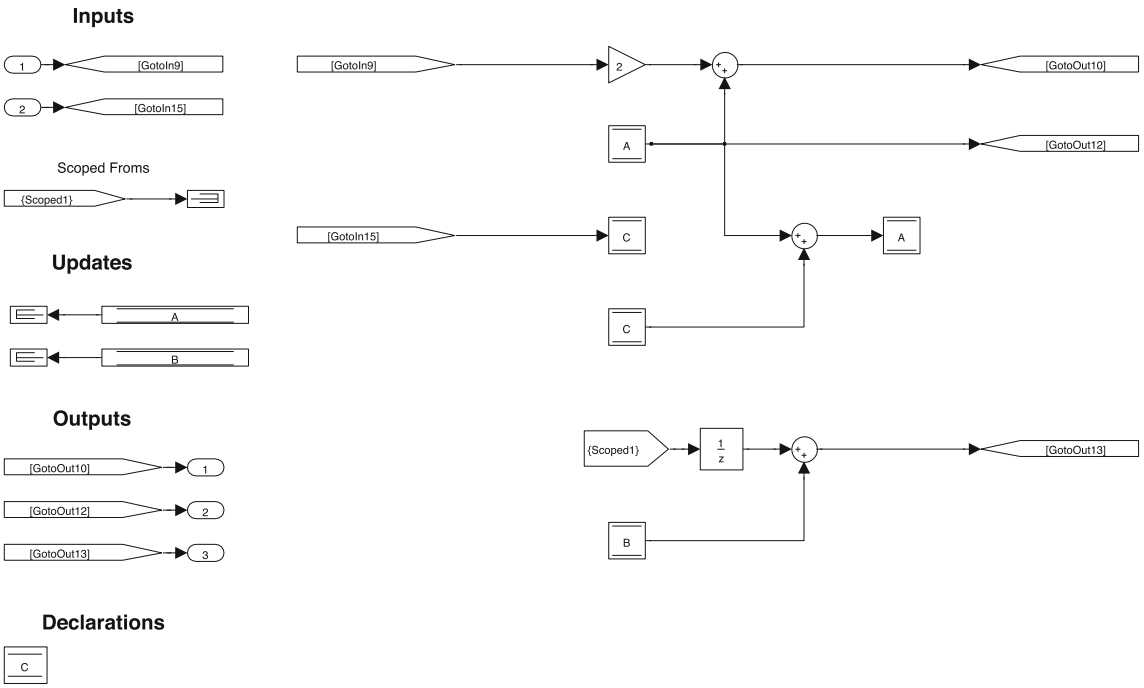


Fig. 5 Subsystem Sub1 including weak signature generated by the Signature Tool

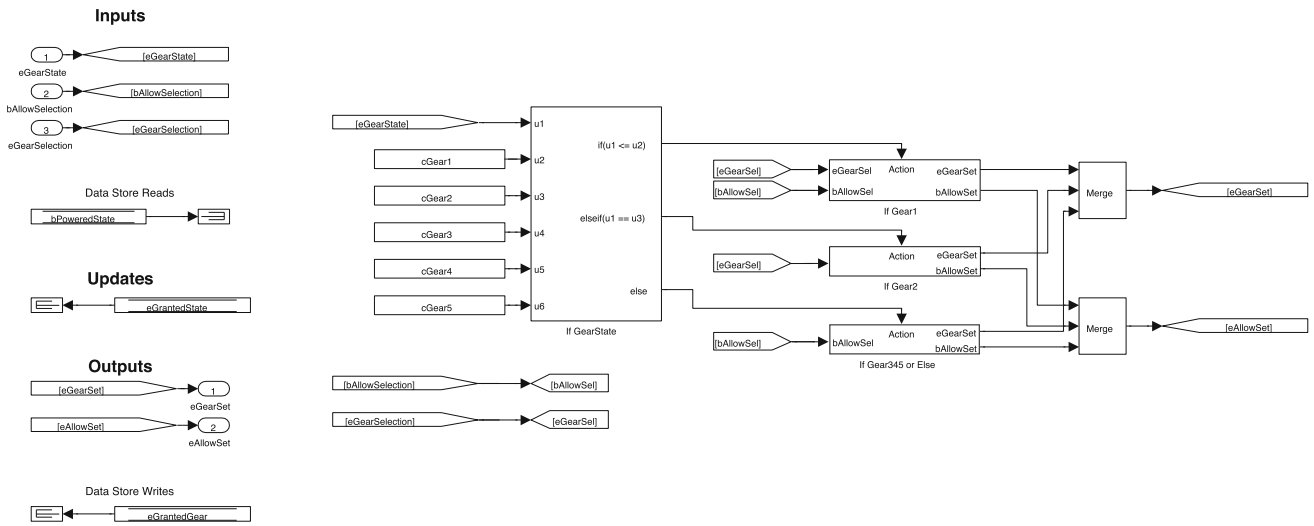


Fig. 6 Strong signature for Gear Processing subsystem from Fig. 3b

blocks, with hierarchy depths between 7 and 10, the total signature extraction function runtime that includes signature extraction for all subsystems in the model's hierarchy was below 53 s; for the largest analyzed model (the model contained 73044 blocks with hierarchy depth of 16), the signature extraction function runtime was 14 min.

The benefits and applications of signatures are numerous. Signatures represent a simplified view of data flow in and out of the subsystem, explicitly identifying hidden dependencies in Simulink models and allowing for easier comprehension of models. For the industrial example subsystem in Fig. 3b, the signature as extracted and then included in the model is shown in Fig. 6.

The signature eliminates the need to search vertically and horizontally through the model's hierarchy in order to understand the model's data inflow and outflow. In this particular example, the user would have had to inspect seven subsystems at three hierarchy levels to gain such an understanding of the data flow.

Also, the Signature Tool has the capability of extracting strong signatures into external documentation. The generated documentation effectively represents the *interface specifica-*

```
INPUTS
Inports:
    eGearState: uint16
    bAllowSelection: Inherit: auto
    eGearSelection: Inherit: auto
Data Store Reads:
    bPoweredState: boolean

OUTPUTS
Outports:
    eGearSet: uint16
    bAllowSet: Inherit: Inherit via internal rule
Data Store Writes:
    eGrantedGear: uint16

UPDATES
    eGrantedState: uint16

TAG DECLARATIONS
DATA STORE DECLARATIONS
```

(a)

INPUTS

Inports

Name	MATLAB Type
eGearState	uint16
bAllowSelection	Inherit: auto
eGearSelection	Inherit: auto

Data Store Reads

Name	MATLAB Type
bPoweredState	boolean

OUTPUTS

Outports

Name	MATLAB Type
eGearSet	uint16
bAllowSet	Inherit: Inherit via internal rule

Data Store Writes

Name	MATLAB Type
eGrantedGear	uint16

UPDATES

Name	MATLAB Type
eGrantedState	uint16

DECLARATIONS

N/A

(b)

Fig. 7 Software documentation generated automatically by the Signature Tool. **a** Strong signature of Gear Processing subsystem exported into text file output. **b** Strong signature of Gear Processing subsystem extracted into a LaTeX table

tion of a subsystem as a part of the subsystem's *software design description* documentation. Therefore, this capability automates part of the process of software documentation production, and this is precisely how our industrial partner uses the tool. In Fig. 7, the signature of the subsystem Gear Processing is extracted into external documentation in two different formats: a text file and a LaTeX table. Note that the external documentation includes the data types of the signature's data items. The type information is automatically extracted from the model.

Software requirements and design descriptions of Simulink models are often generated using *Simulink Report Generator*, a MathWorks tool built into Matlab, that partially automates documentation production from Simulink models and simulations. It supports a number of different formats for generated documentation including Microsoft Word, Rich Text Format (RTF), PDF, DocBook (XML) and LaTeX. The tool offers a number of Simulink-specific components that correspond to typical Simulink content, enabling the extraction of relevant information from Simulink models directly into documents in a highly automated manner. For example, a snapshot of a Simulink subsystem can be included in the design documentation through the use of the System Snapshot component, and will be updated as the model itself changes. In a similar manner, the documentation can also include tables with information extracted from the model using the Array-Based Table component. A table can be populated with the data items and their data types from a subsystem's signature programmatically by using our Signature Tool, in a straightforward manner. More precisely, the table is generated through Simulink Report Generator's Array-Based Table component, which then calls the appropriate functions of the Signature Tool in order to populate the table with the names of signature's data items and types. The generated table for subsystem Gear Processing can be identical to that shown in Fig. 7b, and included in e.g., a Word document.

The use of the Signature Tool, appropriately extended where needed, can go far beyond a comprehension aid and documentation generator:

– Signatures can be used to instill software engineering discipline in design with Simulink. For example, the actual interface of a subsystem can be refined from its

weak signature generated by the tool, e.g., a data store should be removed from the signature's *Updates* and included in *Inputs* if it is to be read-only by the subsystem. The signatures therefore encourage information hiding and encapsulation within a Simulink model. Further, if a (interface) specification of a subsystem is given as its signature, and if its weak signature (as extracted by the Signature Tool) has mechanisms not contained in the specification, this is an indication of the subsystem's potential to access data flow mechanisms that may cause unintended interference with other subsystems in the model hierarchy. In fact, a metric based on signatures has been defined in [3] that measures the difference in the number of mechanisms in the subsystem's weak and strong signature. A larger value for the metric indicates a potentially problematic design, as the subsystem has access to far more resources than it is actually using. The tool supports the calculation of this metric.

– A lack of proper consideration of implicit data flow in tools for testing Simulink models was first noted in [2]. Existing testing tools typically neglect to account for data flow via data stores when generating a test harness for a subsystem. For example, a Data Store Read/Data Store Write block referencing a data store defined outside of a subsystem is a part of the subsystem's implicit input/output, which some testing tools fail to properly include in the generated test harness. To tackle this issue, the subsystem's strong signature extracted by the Signature Tool can be used to generate a test harness that properly accounts for all the incoming/outgoing signals, with the additional benefit of the harness being easily detachable. In fact, we have demonstrated a large improvement in a subsystem's testability (increased coverage and decreased testing effort) when the Signature Tool is used to augment test harnessing before automatic test generation with a commercial testing tool [3]. The test harness augmentation is completely automated using the Signature Tool [3].
– Signatures can be used to classify dynamic inputs (inputs that often change through a simulation run) versus static (inputs that rarely change through a simulation run). Since signatures make scoped tags as explicit (visible) as inports, applying the discipline of e.g., using scoped Goto/Froms for static inputs and using inports for dynamic inputs, would significantly declutter the explicit interface of the subsystem.
– When the subsystem signatures are included in the model, they can be used to incorporate strong typing into subsystems' interfaces. For more details, an interested reader is referred to [2].

3.2 The Reach/Coreach Tool

In this section, we present the Reach/Coreach Tool. First, basic capabilities of the tool are introduced. Then, its implementation and applications are explained. Finally we provide a comparison with related tools.

3.2.1 Tool's capabilities

The Reach/Coreach Tool identifies dependencies in a Simulink model in two different ways:

– *Reach*: For a specified set of blocks (e.g., a set of inports), the tool identifies the blocks and lines of the model which are affected by the specified blocks. The reachable model (the submodel of the original model that consists of the identified blocks, and signal lines that connect them) is then highlighted.
– *Coreach*: For a specified set of blocks (e.g., a set of outports), the tool identifies the blocks and lines of the model which affect the specified blocks. The coreachable model (the submodel of the original model that consists of the identified blocks, and signal lines that connect them) is then highlighted.

The dependencies identified by the Reach/Coreach include both data and control dependencies as will be detailed later in this section.

Once the Reach/Coreach submodel is identified and marked in the Simulink model, the unmarked parts of the model can be trimmed away. The remaining submodel represents a *model slice*.

The application of the tool on a simple example is illustrated in Figs. 8 and 9. The coreachability analysis for outport Out1 results in the submodel marked in gray as shown in Fig. 8a (with subsystem WhileSub from Fig. 8a shown in Fig. 8b). The reachability analysis done on inports In1, In2, and In3 would result in the same submodel being highlighted. The reachability analysis for Data Store Write A in Fig. 9a is highlighted in gray, with the subsystem Sub1 shown in Fig. 9b. Both reachability and coreachability analyses preserve the hierarchical structure of the model.

The Reach/Coreach Tool tracks and highlights *both data flow (data dependencies) and control flow (control dependencies)* in the model:

Data flow Data flow in Simulink is explained in more detail in Sect. 2. The Reach/Coreach Tool tracks the explicit data dependencies — the data flow through signal lines. Assuming that changes to an input propagate to changes in any output for any block (except for Subsystem, If, Switch Case, Bus Creator, and Bus Selector blocks), that is, any

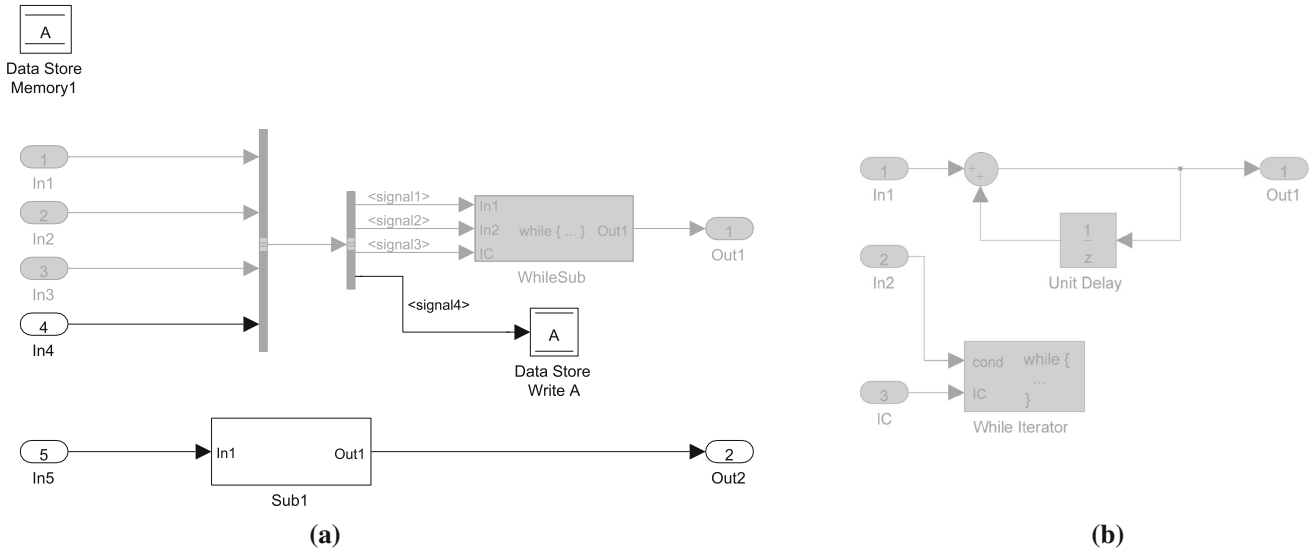


Fig. 8 Coreach analysis for Out1. **a** Coreach for Out1. **b** Subsystem WhileSub from **a**

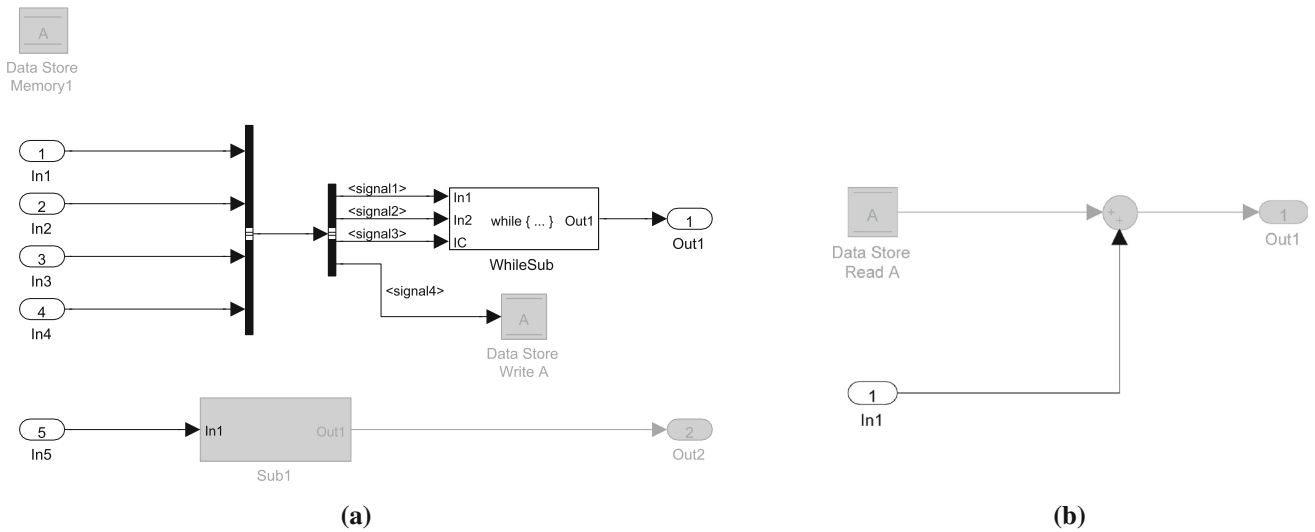


Fig. 9 Reach analysis for Data Store Write A. **a** Reach for Data Store Write A. **b** Subsystem Sub1 from **a**

input can influence any output of a block, and by deducing the input–output influence relation of a subsystem from the structure of its components, it is possible to construct the signal line dependency of the model. The Reach/Coreach Tool provides fine-grained tracking of the data flow through Simulink buses (see Fig. 8a). Tracing dependencies through If and Switch Case blocks will be explained in the next paragraph. Further, as elaborated in Sect. 2.1, there are hidden data dependencies in Simulink in the form of implicit data flow through data stores and Goto/From blocks. The Reach/Coreach Tool tracks these data dependencies as well. For example, in Fig. 9a, the flow from the Data Store Write A block is tracked to Out2, due to the fact that the data store is being read in Sub1. Therefore, the Reach/Coreach Tool accounts for implicit data flow in that it not only tracks the data flow through inports/outports, but also the data flow via data stores, and Goto/From blocks. This is a major difference between our tool and that of [18], as the definition of data dependency in Simulink in [18] does not account for the implicit data flow.

Control flow Control flow logic in Simulink can be implemented using a variety of mechanisms. For if-then-else logic, an If block is used to implement the if-then-else conditions. When evaluated true, a condition triggers the corresponding If Action Subsystem (see Fig. 10). Likewise, for switch logic, a Switch Case block is used to implement case conditions, which when evaluated trigger attached Switch Action subsystems corresponding to each case. Loops are straightforwardly implemented through the use of While Iterator and For Iterator subsystems.

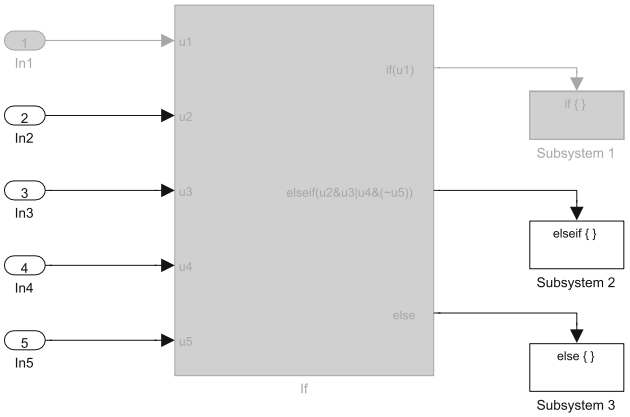


Fig. 10 Coreach on Subsystem 1: first output of If block is affected by first input only

For an example of control flow dependencies tracked by the tool, consider the outputs of an If block that are used to trigger If Action Subsystems, depending on the evaluation of a specified condition on inputs of the If block (for an illustration, see Fig. 10, where Subsystem 1, Subsystem 2, and Subsystem 3 are If Action Subsystems). In early releases of the Reach/Coreach Tool, we assumed that for If (Switch Case) blocks, all inputs affect all outputs—this, however, was a very rough approximation of actual dependencies. For example, in Fig. 10, only In1 affects the first output of the If block. For a condition at an output port of an If block, our tool currently finds its dependencies on the If block's inputs based on the existence of the block's input names within the condition and within the If block's conditions at the ports above (since the conditions are evaluated top down): if the name of an input is found in the condition for an output or in any of the If block's conditions above, then there exists a dependency between the input and the output; otherwise, there is no dependency. Note, however, that this is still an overapproximation of actual dependencies. For instance, if the condition at the first output of the If block in Fig. 10 is $u1 \wedge \neg u1$, the output signal is obviously not affected by the value of $u1$ although our tool would still highlight the dependency of the output on $u1$.

An If Action Subsystem block executes when the input at its Action Port block evaluates to true. If a signal is tracked to an output of an If block, then it further propagates to the If Action Subsystem connected to the output, and all of the If Action Subsystem's outputs. Similarly, if the coreachability analysis is performed from a block whose input is the output of an If Action Subsystem, it will trace back to its Action Port, as well as the data input(s) of the If Action Subsystem as determined by the coreachability analysis inside the subsystem.

When it comes to an Iterator subsystem, each block in the subsystem is (control) dependent on the Iterator block in the subsystem, and, therefore, on its inputs. For example, in Fig. 8a, Out1 is dependent on all the inputs of WhileSub subsystem, since the While Iterator (Fig. 8b) executes the contents of the subsystem based on IC (initial condition) and cond inputs.[3]

The conditional subsystems Enabled Subsystem, Triggered Subsystem, Enabled and Triggered Subsystem, and Function-Call subsystems also have a control input in addition to data inputs. When the control input satisfies a condition, the subsystem is executed. For each conditional subsystem, outputs are dependent on the control input. This means that when the Coreach analysis is performed on a subsystem's output, it traverses back through the subsystem's control input (as well as through any relevant data inputs, as determined by the Coreach analysis inside the subsystem). Similarly, if the reachability analysis traverses to the control input of a conditional subsystem, it is further propagated to all of the subsystem's outputs.

Finally, we note that the dynamics of one part of a model can influence the simulation behavior (simulation results) of another part of the model even if no control or data dependencies exist between the model's parts. More precisely, if a variable-step solver is used for simulation, different time constants of different parts of a model will have impact on the times at which simulation results are produced, and on the accuracy of the solution. The Reach/Coreach Tool does not track this kind of dependency. However, given that the target models of the tool are discrete-time control systems from which code is generated via a fixed-step solver, the dependency via a solver is not relevant.

3.2.2 Implementation and applications

The Reach/Coreach Tool is implemented using Matlab's object-oriented programming facilities. The Reach and Coreach algorithms are fixed-point algorithms, that identify the immediate reached/coreached blocks of a current set of blocks, starting from the initial specified set of blocks on which the Reach/Coreach analysis is to be performed. The average execution time of the Reach algorithm run from a top-level inport of several large industrial models containing between 3671 and 21206 blocks, and on average 6236 blocks, was 62.3 s; for the largest model (the model containing 73044 blocks with hierarchy depth of 16), the Reach runtime was almost 18 min.

Next, we elaborate on how the tool can be used in a model-based software development process.

[3] At the beginning of a time step, if the IC input does not hold, the subsystem is not executed in that time step. If the IC input does hold, the subsystem gets executed, and then, if the cond input is true, the iterator executes the subsystem again. The iterations continue while cond input is true and the number of iterations is less than or equal to the Maximum number of iterations.

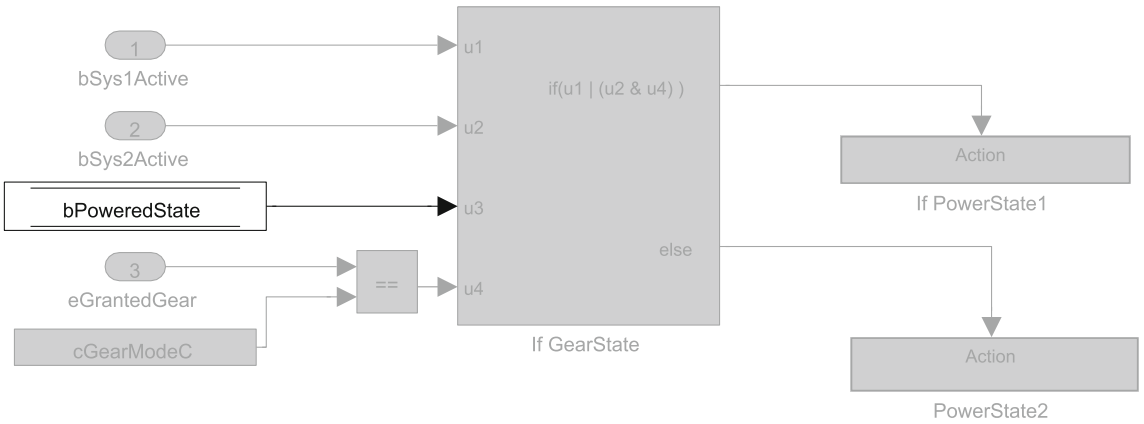


Fig. 11 Subsystem Mode Processing after Coreach was run on all outputs of Gear and Mode Processing

Comprehension While signatures address the comprehension issue at the subsystem level, the Reach/Coreach Tool offers an abstracted view of data and control flow from given blocks backwards to inputs, or forwards to outputs. Both the views generated by the Signature Tool, and the views generated by the Reach/Coreach Tool, are *data flow views*, and, as claimed in [15]: "such views can be very helpful for tracking data flows through a system — especially when there are additional hidden dependencies." Furthermore, by trimming the extraneous blocks from a model, it is possible for a user to gain a greater understanding of the structure of the model. Reach/Coreach analyses can assist developers, testers and reviewers to fully grasp data and control dependencies in a model, as understanding the flow in complex models is very hard without proper abstractions even when the models are reasonably well-documented.

Dead/unreachable code The tool can be used to find unreachable parts of a model. When reachability analysis is performed on all of the model's inputs, unmarked blocks/signals approximate unreachable parts of the model. When the coreachability analysis is performed on all of the model's outputs, the extraneous blocks are unnecessary in the sense that they have no (data or control) effect on the outputs of the model. For example, running coreachability analysis on the outputs of subsystem Gear and Mode Processing from Fig. 3a revealed the Data Store Read block of data store bPoweredState inside the Mode Processing subsystem does not have any impact on the If GearState block's outputs (see Fig. 11). Moreover, the combination of both Reach and Coreach analysis revealed a data store in the Gear and Mode Processing subsystem that is declared, but never used (data store eSpeedState, see Fig. 12). The same issue can also be discovered using the Data Store Rescope Tool as will be shown in Sect. 3.3.

Impact analysis Since the tool identifies the parts of a model affected by a change of a given block, it supports impact analysis. Impact analysis can be of great value in indicating what effect a change in requirements or design can have on system's design. During both initial design and refactoring, the tool can be used for preliminary evaluation of different designs with respect to the extent that the future anticipated requirements/design changes will have on the system's design. Therefore, the tool would help the developer *design for change*. Further, after a change to the design has been applied, the impact analysis can be extremely beneficial, focusing and possibly reducing verification efforts, which are typically very large, especially in the case of safety-critical systems. Verification efforts can be reduced by focusing on the parts of the system affected by the change—thus we view the Reach/Coreach Tool as a useful means of providing impact analysis to help avoid costly reanalysis and testing of the unaffected parts of the system.

For example, a developer refactoring the logic regarding driver requests for Gear 2, Gear 3, Gear 4, and Gear 5 inside the Gear Processing subsystem from Fig. 3b might be inclined to group or change the order of corresponding constants at the inputs of the If GearState block (last four inputs). The developer may believe that the If Action Subsystem If Gear1 will not be affected by the change, as there is no obvious dependency between the subsystem and any of the last four inputs of the If GearState block, since the execution of the If Gear1 subsystem is dictated by a condition that depends only on the first two inputs of the If Gear1 block. However, application of the Reach analysis on the four constants suggests that there is some implicit dependency between the last four inputs of the If GearState block and If Gear1 subsystem. If the developer wishes to learn the origin of this dependency, he/she can use the results of the Reach analysis on this subsystem (including its descendant subsystems). This view (Fig. 13a), however, is overloaded with other information as it also contains dependencies that the developer expected (dependencies of If Gear2 and If Gear345 or Else on the last four inputs of the If GearState block). Therefore, we used the Reach/Coreach Tool to generate a simplified

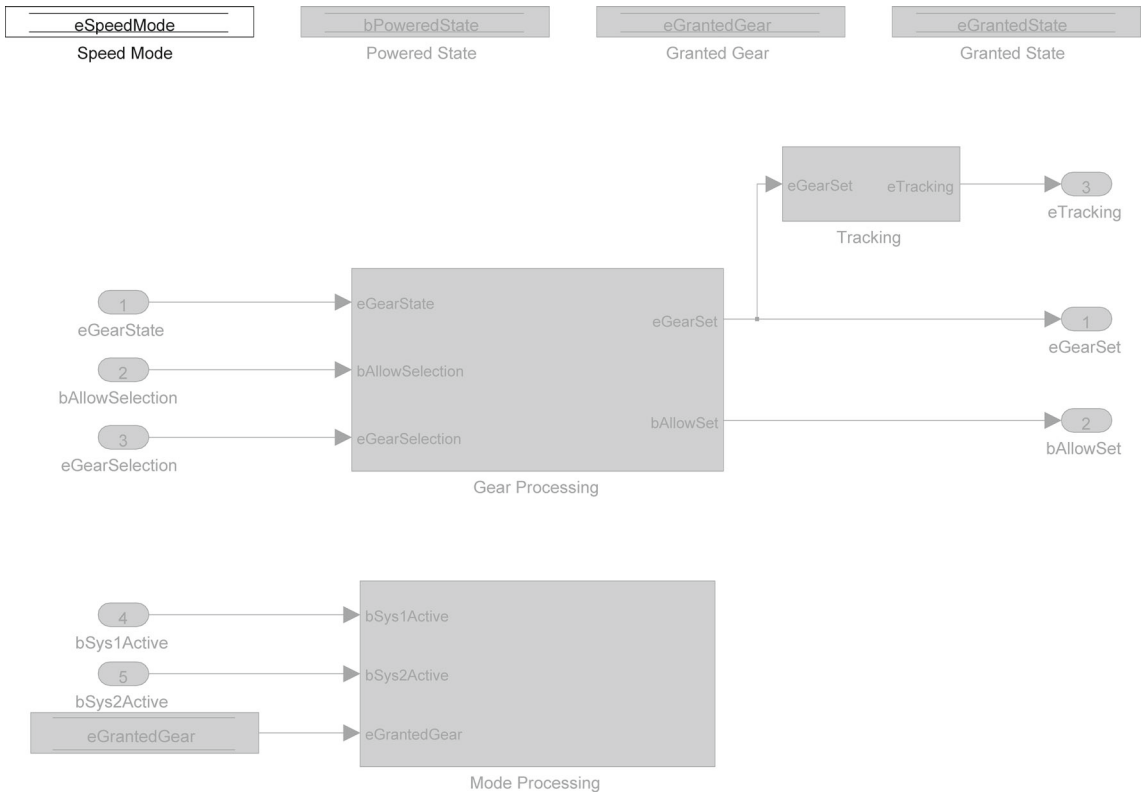


Fig. 12 Subsystem Gear and Mode Processing after Reach and Coreach were run on all its inputs and outputs, respectively

view of the system that identifies the effects of the four constants on the If Gear1 subsystem only, without showing the constants' effects on If Gear2 and If Gear 345 or Else except dependencies that are relevant to If Gear1. Using this view, as shown in Fig. 13b–d, one can see that If Gear2 subsystem (Fig. 13c) writes into data store eGrantedState (Fig. 13d), and If Gear1 (Fig. 13b) reads from this data store. Therefore, the developer is offered a model view in which only the relevant dependencies (in this case, the ones that were not initially foreseen) are highlighted. In general, after a user specifies the subsystems that should not be affected by a change, the tool can be used to generate a view of the undesired dependencies, if they exist.

Refactoring The tool can be used to find independent/weakly dependent data flows. These may be candidates for separation into different subsystems. Likewise, some seemingly independent data flows can be shown to have hidden dependencies, as shown in the previous paragraph.

3.2.3 Comparison with similar tools

A Simulink model slicing tool is presented in [18], although, to the best of authors' knowledge, the tool itself is not available for use. In [18], data dependencies are calculated by traversing signal lines, and control dependencies are derived from the calculation of *Conditional Execution Contexts* — Simulink schedules for the execution of blocks modeling conditional dependence. Compared to the model slicing approach of [18], our tool accounts for not only the data flow through signal lines, but also the data flow through Simulink's data stores and From/Goto blocks. Furthermore, the Reach/Coreach Tool provides fine-grained data flow tracking through Simulink buses, which is not supported in the tool of [18].

With the release of Matlab 2015a came the introduction of Simulink Design Verifier's (SDV) *Model Slicer* tool, which identifies dependencies between blocks with many of the same motivations as the Reach/Coreach Tool. It is no surprise then, that the core functionality is similar between these tools. With respect to a block, they both support slicing to discover the blocks which are dependant on it (termed Reach and Downstream for our tool and MathWorks' tool, respectively), those blocks which it depends on (Coreach/Upstream for our tool and Mathwork's tool respectively), as well as slicing in both these directions (bidirectional). In the three cases, the tools both support highlighting in various colors, as well as the actual extraction of a slice from the remainder of non-dependant blocks in the model.

Nevertheless, our use and evaluation of Model Slicer has revealed several distinguishing factors between the two tools. Firstly, it is commonly the case that large companies have long and complex tool chains which evolve slowly. Specifically, our industrial partner relies on an older version of Matlab for its control software development, across multiple

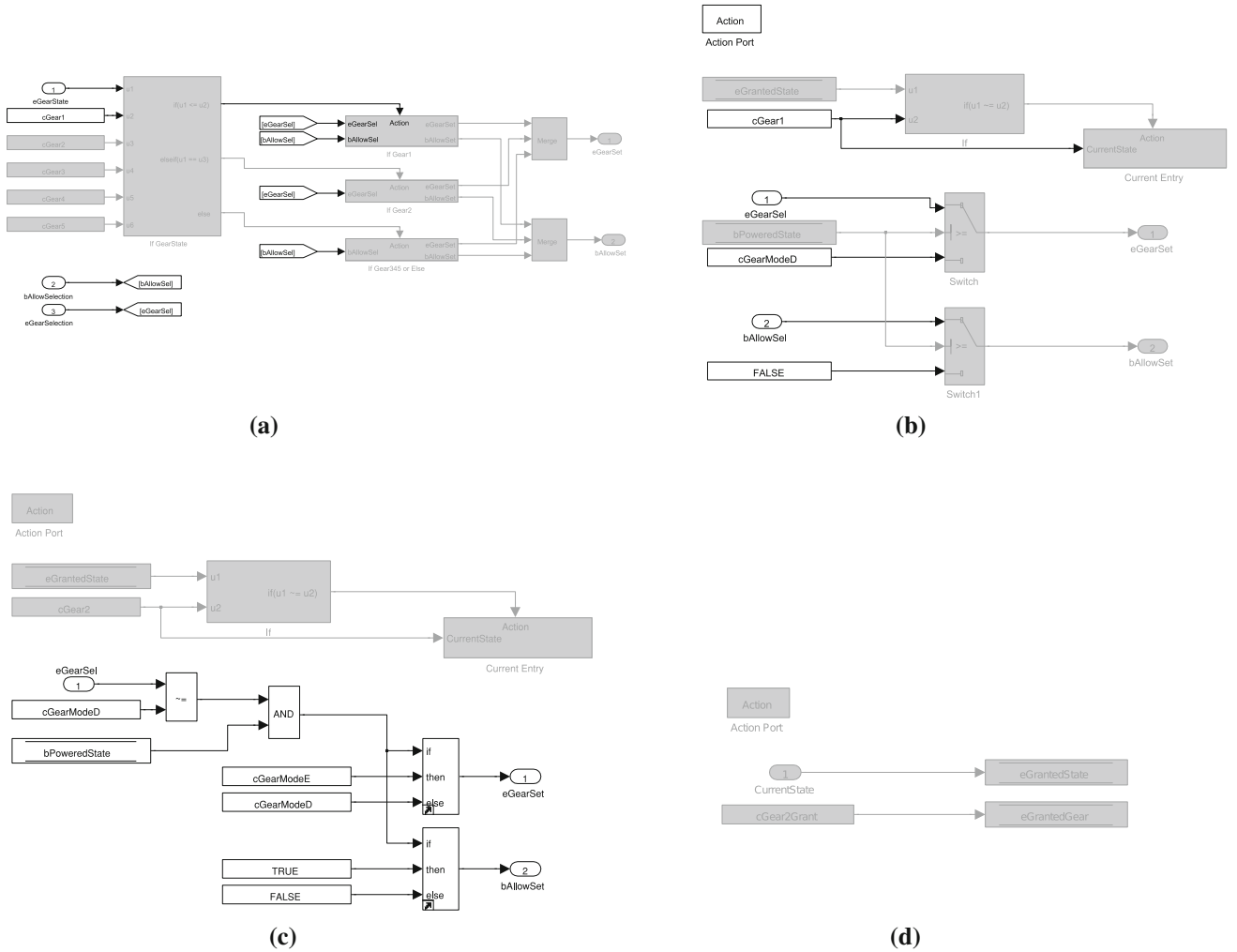


Fig. 13 Impact analysis. **a** Reach from last four inputs of the If GearState block. **b** Effect of last four inputs of If GearState on If Gear1 only (If Gear1 subsystem). **c** Effect of last four inputs of If GearState on If Gear1 only (If Gear2 subsystem). **d** Effect of last four inputs of If GearState on If Gear1 only (If Gear2's Current Entry subsystem)

teams and projects. For this reason, backward compatibility with older versions of Matlab is necessary in practice. With Model Slicer being newly added to the Mathworks product line in Matlab 2015a, it is not possible to run this tool on an earlier version. In contrast, the Reach/Coreach Tool is supported for previous versions of Matlab up to and including Matlab 2011b.

Additionally, the difference in costs for acquiring these tools is considerable. While the Reach/Coreach Tool is free for use on Matlab Central, the price for single and concurrent licenses of SDV are approximately $8,500 CAD and $34,000 CAD, respectively. The Model Slicer tool is but a small component of SDV; however, it cannot be acquired without purchasing SDV in its entirety. For smaller companies, the price of purchasing SDV as a whole just to access its model slicing capabilities may not be justifiable nor feasible from a financial standpoint.

Several shortcomings in SDV Model Slicer's ability to trace certain dependency paths have been revealed during our use of the tool on industrial models. Most notably, the tracing of dependencies beginning at so-called *virtual* blocks is not supported. Virtual blocks are considered by Simulink to be those blocks which do not affect the execution of the model, but rather facilitate the visual structuring of the model [27]. Blocks considered to be always virtual include Goto/From, Mux, Demux, and Terminator. Save for some very specific circumstances, inport, outport, and subsystem blocks are also virtual. For instance, inport blocks are only ever non-virtual if they are located inside a conditionally executed subsystem or an atomic subsystem, and are directly connected to an outport block. Outports are virtual unless they are located at the root level of the model, i.e. outports located within subsystem blocks are virtual. Thus, a major hindrance in using this tool is that for the majority of cases, slices beginning from inports and outports are not permit-

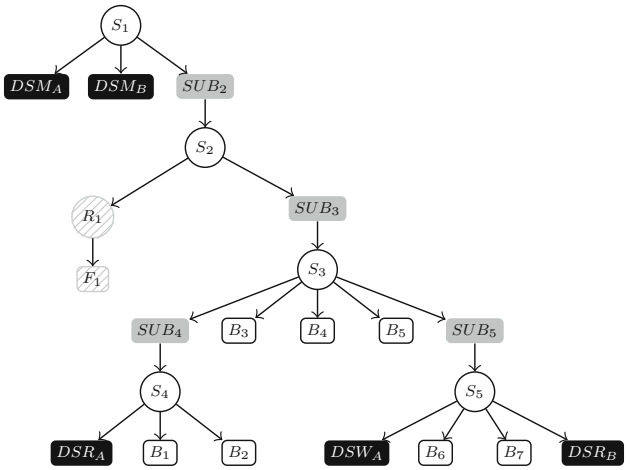


Fig. 14 Data store push-down: before

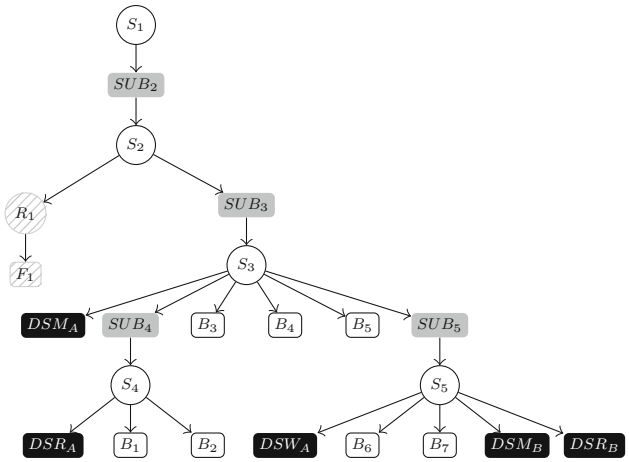


Fig. 15 Data store push-down: after

ted. This severely restricts one's ability to view how inputs affect the computation of outputs, and vice versa. Replicating the results of Fig. 8a is not possible with Model Slicer, as the subsystem is not located at the top level of the model, causing the outports to be virtual, and therefore a slice is not permitted to begin at these outports.

Moreover, in the case of subsystem blocks, Model Slicer handles slices beginning at these blocks inconsistently. Simulink considers subsystems to be virtual in any situation where they are used in non-conditionally executed and non-atomic subsystems. Nonetheless, generating slices with subsystems as starting points in such cases is supported, contradicting the assertion that all virtual blocks cannot be starting points.

Furthermore, Model Slicer neglects to account for some finer details of control flow in Simulink: it does not provide the same level of precision in tracking control flow dependencies as the Reach/Coreach Tool. More precisely, for the mechanisms dealing with control flow, such as If and Switch Case blocks, Model Slicer assumes that all inputs affect all the outputs, which, in general, is a much rougher approximation of actual dependencies than that generated by the Reach/Coreach Tool as discussed in Sect. 3.2.1.

3.3 The Data Store Rescope Tool

In working with the industrial models of our automotive partner, we observed that many models defined the majority of their data stores at the top level of a model's hierarchy. This practice is analogous to programming using a large number of global variables, and it is widely considered a bad software engineering practice. Data stores, like variables in traditional programming languages, should be properly scoped in order to disable inadvertent/unwanted access to the data stores. Also, proper scoping declutters the interface of a subsystem by hiding low-level details of the subsystem, therefore providing proper *encapsulation*. Proper data store scoping also reduces the number of (implicit) inputs for testing, resulting in possibly fewer generated tests involving fewer test steps. Therefore, proper scoping of data stores enhances comprehensibility, maintainability, testability, and reusability of Simulink subsystems.

In this section, the concepts of data store rescope operations are discussed, and their implementations and applications are detailed.

3.3.1 Illustration of data store push-down operation

The *scope* associated with data stores and Goto/From blocks was previously discussed in Sect. 2.1. A block falls within the scope of another block b if it is contained in a system that belongs to the scope of block b. Our goal is to limit the scope of data stores as much as possible. For example, if a Data Store Memory block occurs high up in the model hierarchy, and there are two data store references (Data Store Read or Data Store Write blocks) lower in the hierarchy, the goal is to *push-down* the data store (Data Store Memory block) in the model hierarchy to the smallest subsystem such that both references are still within the scope of the data store.

In Fig. 14, a system is shown before the data store push-down operation has been applied. The circle nodes represent systems, and rectangular elements are blocks. Gray blocks are subsystem blocks, and circles with a gray line pattern are reference blocks (blocks that refer to a block in another file, here denoted by a patterned rectangle). Black nodes labeled DSM_i, DSR_i, and DSW_i correspond to Data Store Memory, Data Store Read, and Data Store Write blocks, respectively. Arrows emanating from systems to blocks denote membership of the block in the system, that is, if A and B are systems, $A \rightarrow B$ denotes that subsystem A contains subsystem B. A similar relationship exists for references and files. Figure 15 shows the system after the data store push-down operation has been applied to it.

3.3.2 Implementation and application

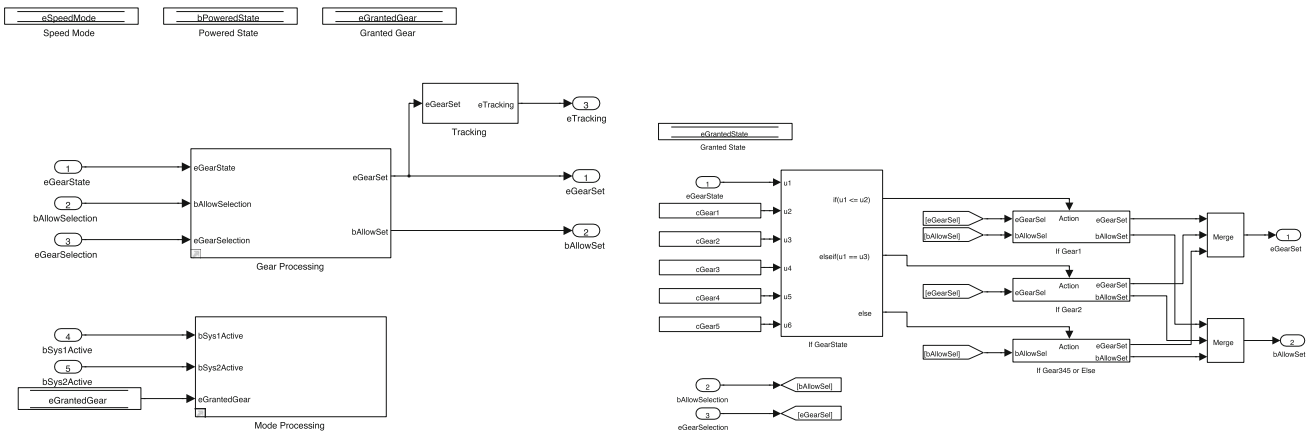
The push-down algorithm is implemented as an iterative Matlab function. Firstly, the algorithm searches the model for all Data Store Memory blocks. Then, it searches for all the corresponding Data Store Read and Data Store Write blocks. The addresses of these corresponding Data Store Read and Data Store Write blocks are then parsed to find their lowest common ancestor. Lastly, unless the lowest common ancestors reside in a library linked subsystem, the Data Store Memory blocks are then pushed-down to their respective lowest common ancestors. The longest execution times of the algorithm on the large industrial models we analyzed was 8.7 s; the model contained 7792 blocks and 94 Data Store Memory blocks out of which 62 were rescoped.

The application of the tool on the industrial example from Fig. 3a results in one data store being pushed-down: the data store eGrantedState's declaration is moved from the subsystem from Fig. 3a (see Fig. 16a), and placed in the subsystem from Fig. 3b, as shown in Fig. 16b. Further, the application of the tool on the same system revealed another potential issue with the model from Fig. 3a: data store eGrantedGear has no references. The issue of declared, but unused data stores is ignored by Matlab, and, to the best of our knowledge, there exists no tool to detect the problem.

Given its importance, we propose the proper scoping of variables to be included in the *modeling style guidelines* for Simulink. With Simulink/Stateflow emerging as a leading environment for model-based design of embedded systems, a number of guidelines have been created to assist designers in modeling. Guidelines typically provide a wide range of rules/recommendations (e.g., naming conventions, usage of Simulink patterns for different constructs such as case constructs, grouping of blocks into subsystems, *etc*.). In the automotive industry, the most notable modeling standard is published by The MathWorks Automotive Advisory Board (MAAB) [28]. In addition, companies use in-house guidelines to improve the quality of their software. In-house guidelines typically contain a number of checks from standard guidelines. Adherence to these rules improves software testability, understandability, and maintainability. Also, compliance enhances simulation and code generation capabilities. For example, each of the MAAB rules/recommendations for Simulink/Stateflow is justified by one or more of the following:

- Easily understood algorithms (readable models, uniform appearance of models, code, and documentation, clean interfaces, professional documentation)
- Effective development process and workflow (ease of maintenance, rapid model changes, reusable components)
- Efficient simulation and analysis
- Generation of code that is efficient and effective for embedded systems
- Ability to verify and validate a model and generated code (requirements traceability, testing, problem-free system integration, clean interfaces)

In order to include proper scoping of data stores in modeling guidelines, a rule can be formulated to require that each data store (with certain exceptions) be defined at the lowest hierarchy level such that all the references to the data store are still within its scope. The exceptions account for situations where a developer might desire to leave a data store defined at a higher hierarchy level than currently needed, foreseeing that the data store will be used in the future by other subsystem(s) at the same or higher hierarchy levels. Therefore, the user of the tool should be able to choose the data stores that should



(a) **(b)**

Fig. 16 Illustration of push-down operation. **a** Subsystem from Fig. 3a after push-down operation. **b** Subsystem from Fig. 3b after push-down operation

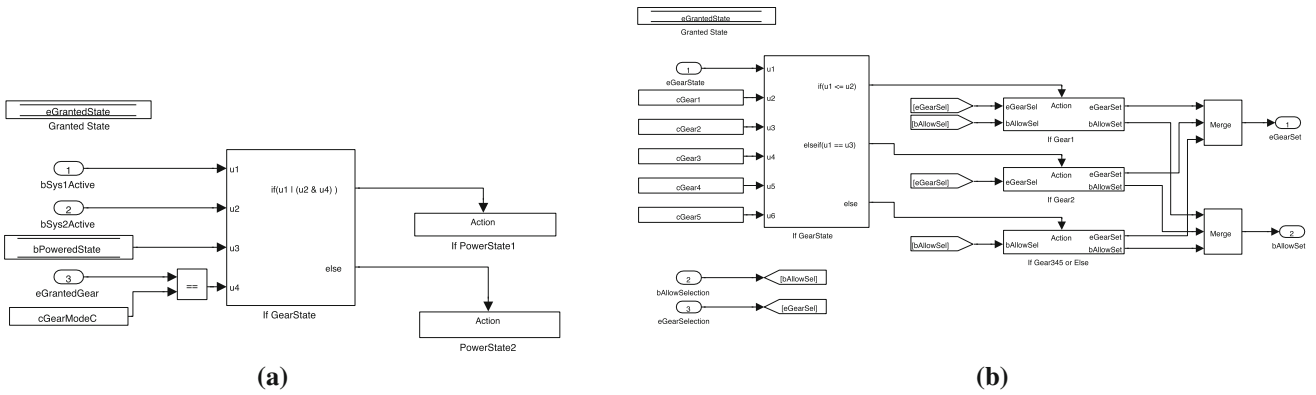


Fig. 17 Illustration of data store repair operation. **a** Before. **b** After: data store eGrantedState has been moved to subsystem Gear Processing (from Fig. 3b)

Table 1 Push-down metric

Model	Number of data stores	Number of data stores pushed	Push-down metric
$Model_1$	82	55	87
$Model_2$	195	111	334
$Model_3$	14	11	33
$Model_4$	13	13	37
$Model_5$	12	3	6
$Model_6$	13	4	8
$Model_7$	62	36	96
$Model_8$	15	15	24

not be moved down the hierarchy. Our tool supports this feature. A straightforward modification of our tool can then be used to check for the compliance of a Simulink model with the rule, and then correct the model (perform the push-down operation) if needed so that the model adheres to the rule. The guidelines published by the Japan MathWorks Automotive Advisory Board (JMAAB) include a rule which strongly recommends positioning Data Store Memory blocks as low as possible in the model hierarchy, and discourages top level use [26]. This is an example where our tool can serve as an autocorrection tool and assist in achieving compliance with published industry guidelines.

The Data Store Rescope Tool performs another useful operation. Assume that the declaration for data store eGrantedState has been erroneously moved to subsystem Mode Processing (Fig. 17a). The references to the data store are now outside of its scope and Simulink throws an error upon diagram update. Our tool can then be used to repair the model: the declaration is first moved to the model's top level, and then it is rescoped downwards to minimize the data store's scope, as described in Sect. 3.3.1 (see Fig. 17b).

When it comes to the push-down operation, both the number of data stores and the numbers of levels data stores have been pushed down the hierarchy represent a useful indication of the improvement in modularity of a model after the push-down operation has been applied to it. That is why we define the *Push-Down metric* as the total number of levels that all the data stores in a model were pushed-down. Our tool calculates the metric. The results presented in Table 1 represent the metric values after a number of industrial models have been rescoped.

An interesting synergy between signatures and the push-down operation was demonstrated in [3]. The push-down operation was applied on an industrial automotive model shown in Fig. 18a. The number of data stores at the root level (data stores are represented by rectangles at the bottom of the figure) has been significantly decreased by the push-down operation, as illustrated by Fig. 18b (the details of both these models are not legible in the figures for confidentiality purposes). From a software engineering perspective, the push-down operation clears the interfaces of the model's subsystems at different hierarchy levels. Given the large number of data stores defined at the top level of the model, the impact of the push-down operation is largest for the subsystems located at the model's top level. This change in subsystems' interfaces is obvious with simple visual inspection of their signatures as generated by the Signature Tool. Further, the Signature Tool was used on the model's subsystems to calculate the difference in the number of data items in a subsystem's weak and strong signature. This metric as formalized in [3] and, as explained in Sect. 3.1, indicates the quality of modularization of designs in Simulink. The values of the metric for the model's subsystems, before and after the push-down operation, indicate the significant improvement in modularity realized by the push-down operation.

Finally, we note that the scoped and global Goto/From mechanisms can be properly scoped in much the same manner as data stores. The rescope operation of the scoped and global Goto/From tags is planned for a future release of the tool.

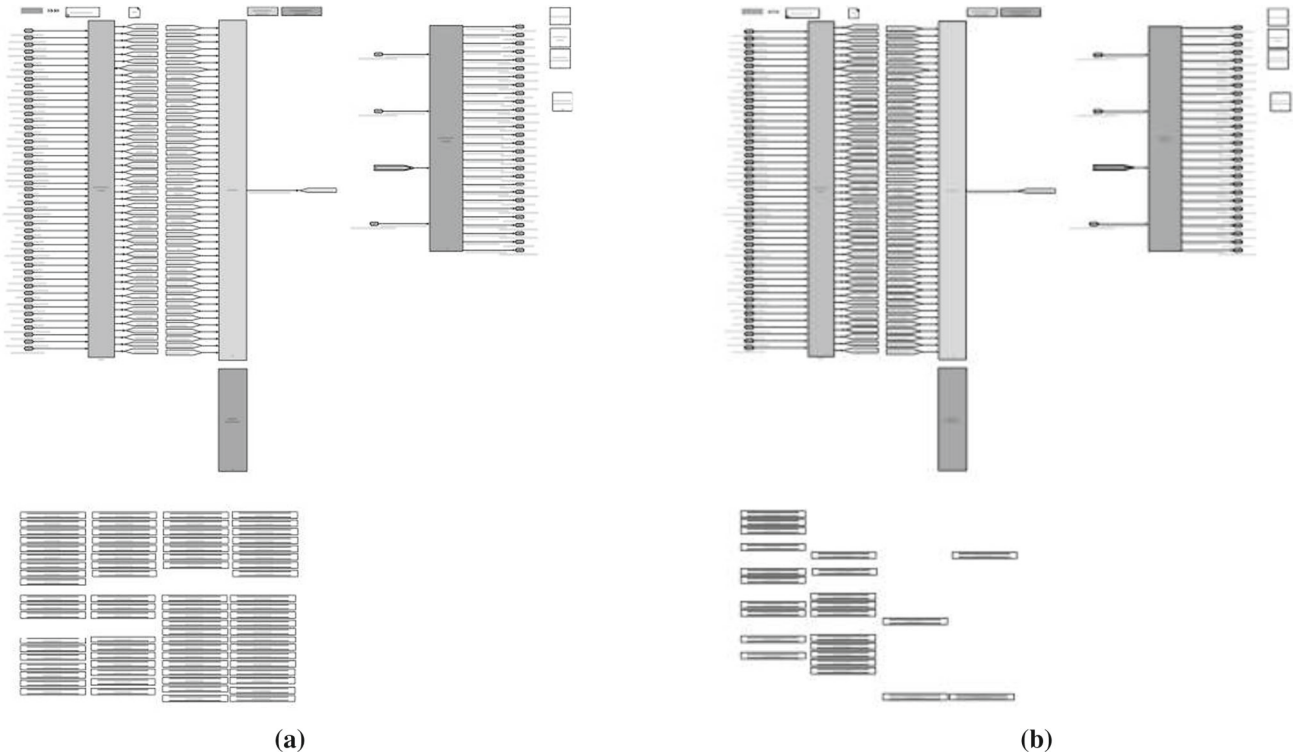


Fig. 18 Push-down on a large industrial model. **a** Top level of the industrial model before push-down operation. **b** Top level of the industrial model after push-down operation

3.4 The Auto Layout Tool

In a modeling environment such as Simulink, the readability of models is largely determined by the graphical layout of blocks and lines in the model. This is similar to traditional textual programming languages and the visual formatting their IDEs (Integrated Development Environments) prescribe. However, in the case of Simulink's graphical notation, achieving a good visual layout requires more effort. Since most refactoring operations perturb model layout, and readjustment of model layout is a tedious and error-prone process if performed manually, automatic layout is viewed as an essential component of model refactoring. However, while there are tools which increase the readability of Simulink models, there does not exist a commercial tool that comprehensively tackles the automatic layout of Simulink models. For example, there exist commercial tools that check Simulink models for compliance with respect to modeling style guidelines (*MXAM* by Model Engineering Solutions [13], Simulink *Model Advisor* by the MathWorks [22] with the *Verification and Validation Toolbox* [23], etc.). For some of the rules from the guidelines, these tools also automatically repair models so that they adhere to the rules. However, the repair capabilities of these tools are modest at best. While a number of layout algorithms have been proposed (e.g., [11,12]), to the best of authors' knowledge, no tools based on these papers are available for download for Simulink. Further, a layout tool is available via Matlab Central [24]. The tool cleans up Simulink models without significant changes to their layout structure (positioning of blocks), but rather focuses on line transformations. While the tool is very useful in certain cases, it struggles with very messy layouts that need comprehensive restructuring (block displacements), and is much slower than our tool.

The Auto Layout Tool consists of the Layout Engine and some minor transformations. The Layout Engine uses an existing graph drawing algorithm [8] implemented in *Graphviz*. Graphviz is a set of open-source tools for drawing graphs represented by the DOT graph description language. More precisely, our tool harnesses Graphviz's layout engine *dot* for the auto-positioning of model blocks; signal lines are then auto-generated using Simulink's built-in automatic line positioning support. Graphviz's *dot* tool uses the algorithm of [8] to rearrange blocks and lines in a consistent and organized manner to maximize readability (the same algorithm is also partially used by [11]). The tool resizes blocks based on the number of inputs and outputs, and organizes the lines such that the number of crossings is minimized. Figure 19a presents the Gear and Mode Processing subsystem with a perturbed layout, and Fig. 19b depicts the model after the Auto Layout Tool was run on it. Note that model comprehension and readability have markedly improved. Also,

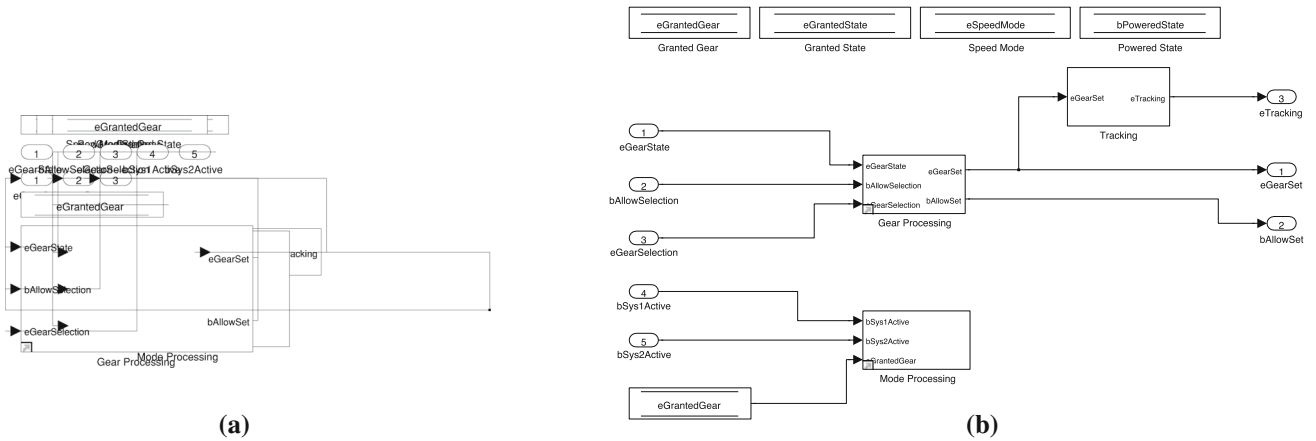


Fig. 19 Illustration of Auto Layout Tool. **a** Before. **b** After

the Layout Engine accommodates some Simulink-specific layout requirements. For example, one of the important recommendations for Simulink diagrams' layout as suggested in the MAAB guidelines [28] states that inports and outports should be placed on the left and right side of the model, respectively, unless they are moved to prevent crossings.

The execution time of the tool on the model from Fig. 19a was 2.7 s. On a larger subsystem (also an industrial model's subsystem) containing 351 blocks and 243 lines, the tool's execution time was 53 s.

The current version of the Auto Layout Tool also features a number of refactoring transformations that were found to be very practical:

Signal Line to/from Goto/From Two simple related transformations have been proven very useful in every-day development with Simulink: *Signal Line to Goto/From* and *Goto/From to Signal Line* transformations. As stated in Sect. 2, the Goto/From mechanism in Simulink is used to replace signal lines to avoid line crossings that impede the readability of models. However, although some guidelines on the usage of the mechanism exist [28], the decision on whether to use a Goto/From or a signal line in a specific instance is often subjective and left to developer's discretion. Figure 20 illustrates an application of *Signal Line to Goto/From* that eliminates two line crossings.

The transformations can be useful in many different situations. For example, we have seen industrial models that heavily use local Goto/Froms in order to avoid multiple line crossings. However, the heavy use of Goto/From also impairs readability given that a Goto/From connection is implicit and has to be traced by matching the names of the Goto block and its corresponding From blocks. The developer can try to maximize the readability of diagrams by transforming some of the Goto/From blocks to signal lines, and then using the Layout Engine to minimize the number of crossings, and improve readability in general. Similarly, a model that suffers from an overuse of signal lines leading to a large number of line crossings, could use an introduction of Goto/From blocks to increase readability. Also, standard or company-specific guidelines might impose or restrict the usage of Goto/From in certain instances. Then, our tool can be used to automate a required transformation to ensure that the model conforms to the standard/company rules/recommendations.

A transformation identical to *Goto/From to Signal Line* was implemented in the tool presented in [29]. To the best of our knowledge, this tool is not publicly available.

Subsystem flattening Use of the *Flatten Subsystem* transformation is illustrated in Fig. 21. The transformation replaces a virtual subsystem with its content. Figure 21 illustrates the flattening of subsystem Mode Processing. Figure 21b shows an intermediate stage of the transformation, when the content of subsystem Mode Processing is copied over to its parent diagram. As seen in the figure, the layout of the diagram has become significantly perturbed and hard to read, with a number of blocks overlapping and lines crossing. Consequently, the Layout Engine is run to improve the layout, with the resulting model as shown in Fig. 21c. The intermediate stage (Fig. 21b) is not evident to the user: we show it here just to demonstrate the inner workings of the transformation.

Starting from Matlab 2015a, MathWorks offers the *Expand Subsystem* transformation which is identical to our *Flatten Subsystem*. However, our tool works on older versions of Matlab, up to 2011b. Given the slowly evolving tool chains of larger companies as discussed in Sect. 3.2, the backwards compatibility of our transformation enables those companies to immediately benefit from the transformation. Another flattening tool was developed in [7]. The tool is implemented outside of the Matlab environment, necessitating one to export the model before performing the transformation, and then re-import it. It also does not adjust the visual layout of the model afterward. Further, to the best of authors' knowledge, this tool is not publicly available.

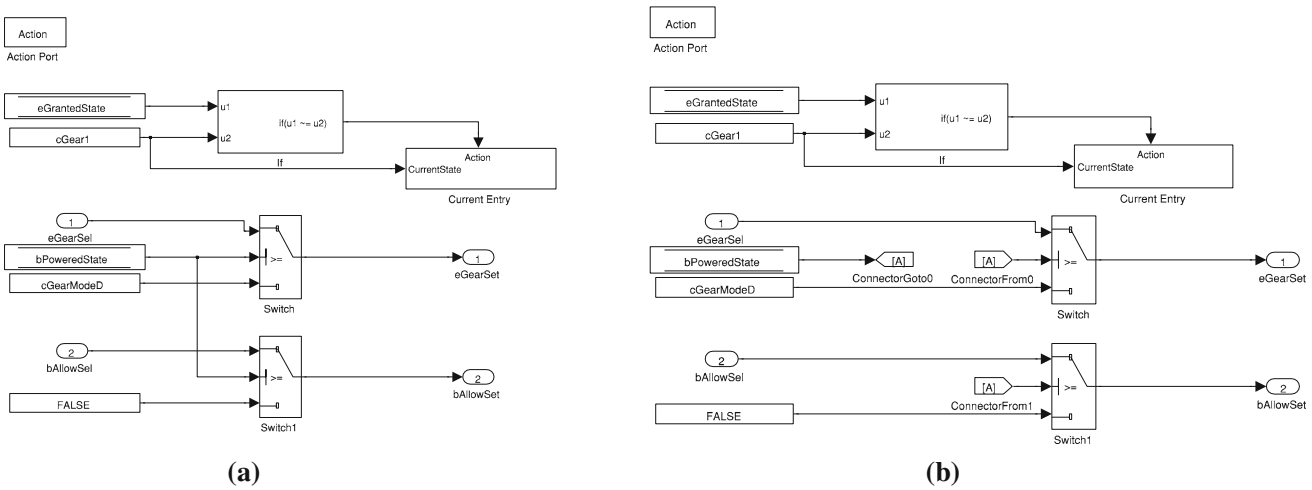
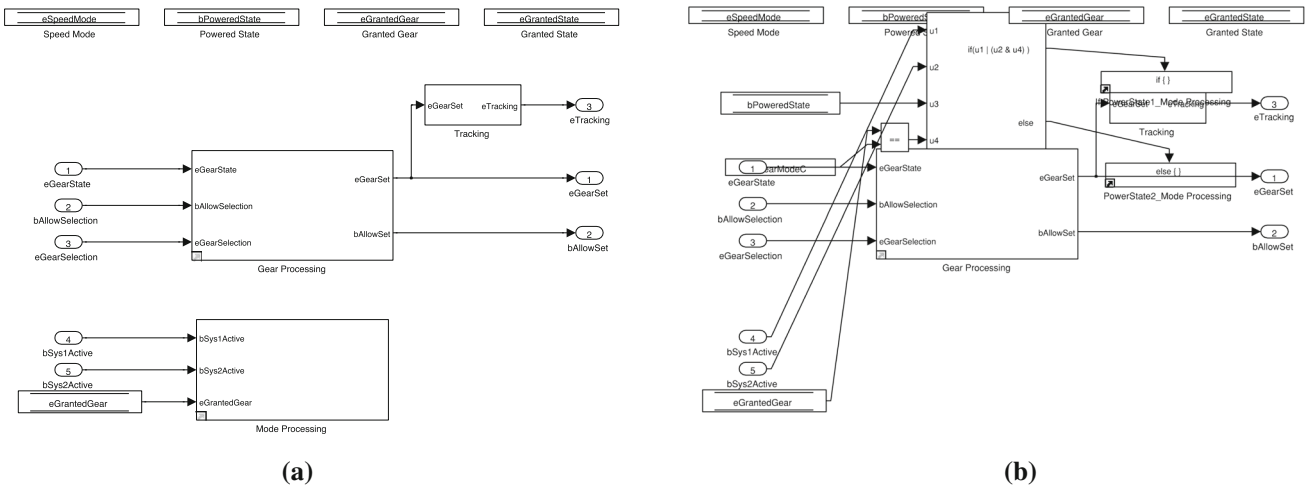


Fig. 20 Illustration of Signal Line to Goto/From transformation. **a** Before. **b** After



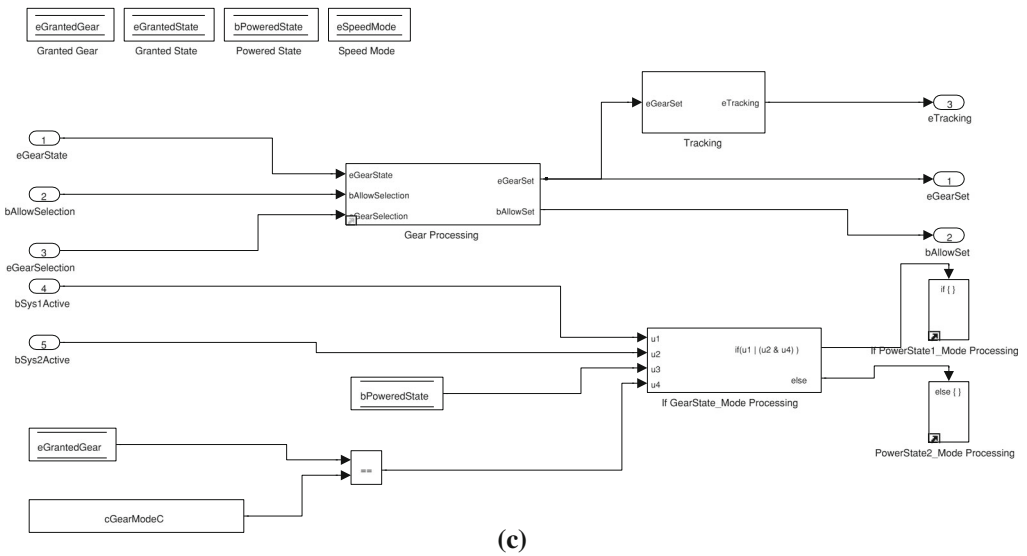


Fig. 21 Illustration of Flatten Subsystem transformation. **a** Subsystem Gear and Mode Processing before Subsystem Flatten. **b** Intermediate stage in flattening subsystem Mode Processing. **c** Subsystem Gear and Mode Processing after flattening subsystem Mode Processing

4 Software engineering rationale for the tools

In order to apply software engineering principles, software engineers must be equipped with appropriate tools and techniques [9]. While the previous section elaborated on the role of each of the tools in model-based development and how the tools are related to some traditional software engineering practices and principles, this section offers a brief summary of how the tools are used in model-based development, while making the link between the tools and integral software engineering principles more concrete. Further, we clarify how the concepts of *module* and *component* from traditional software engineering map to Simulink's constructs.

The set of tools described in this paper act upon the primary design artifact of the model-based development approach: the model. A Simulink model can be thought of as a component—a self-contained and independent grouping of related functions. The various functionalities of a component are typically implemented as modules, or in the case of Simulink, (virtual) subsystems. Subsystem blocks are often implicitly identified as the Simulink equivalent of a module (e.g., [4,6,17]). The interfaces of a model are generally stable as development takes place, and remain intact through the code generation process (e.g., a C file is generated for each model), whereas subsystem boundaries are abstracted away during code generation. For this reason, we have found that in industry, modularity at the subsystem level is not a major concern. Nevertheless, in our experience with a major automotive OEM, we have observed that changes in requirements and design are mainly reflected within a model, across subsystems. Therefore, software engineering principles, such as those concerning modularity, are also necessary at the subsystem level. As a result, the tools focus on supporting software engineering practices for models and their subsystem levels.

The Signature Tool can be used by software engineers during several phases of the development process: design, implementation, testing and documentation. The signature concept introduces self-documenting capabilities of imperative programming languages into Simulink. In explicitly identifying data dependencies, the tool provides engineers with a clear understanding of what data are available to them at any level in the model hierarchy, facilitating the design and implementation of features. Signatures support the principle of modularity by providing well-defined interfaces of subsystems. Moreover, in analyzing the difference between a system's weak and strong signatures, the tool can guide refinement of an interface in order to minimize access, thus supporting encapsulation and the principle of information hiding. This also can be viewed as a way of instilling the principle of least privilege, as constraining an interface to its strong signature will eliminate access to data which is not essential to its functionality, thus removing the potential for unintended data modification. In the testing phase of development, the Signature Tool's ability to create test harnesses which account for hidden data flow is beneficial in supporting unit testing, ultimately increasing coverage with less effort for the test engine. Lastly, the tool can be used for automatic generation of subsystem interfaces in design documentation.

The Reach/Coreach Tool is applied in the design, implementation, verification, and maintenance phases of model development. Its different dependency analyses are useful for identifying how changes to the design will impact other areas of the model. The tool supports the principle of the separation of concerns by allowing engineers to concretely visualize and ensure separation. In particular, it can identify areas of models that are weakly dependant on each other, providing candidates for separation into independent subsystems. Further, the tool promotes the design for change principle as it allows developers to estimate the impact of potential requirements or design changes on a design and then make design decisions, as well as focus subsequent verification efforts accordingly. Finally, the tool helps with standards and modeling guidelines compliance by identifying dead/unreachable code and unused data stores.

The Data Store Rescope Tool is beneficial when it comes to the implementation of models. While the Signature Tool can be used to identify instances where hidden data flow is available (i.e., present in the interface) but not actually needed, the Data Store Rescope Tool may be able to remedy this situation by rescoping the data store to a lower level in the model hierarchy. In general, this tool expedites the proper scoping of data stores and helps enforce the principles of modularity and least privilege throughout a model.

The Auto Layout Tool focuses on the implementation and maintenance phases of the model development process, automating the substantial yet tedious task of model formatting. Changes to a model will often perturb the graphical layout of its elements, and this tool facilitates the model readjustment which follows suit. Just as with textual programming languages, consistency is an integral principle for graphical programming languages. Models are the primary design artifact, and so increasing their readability and understandability via consistency is necessary. The Auto Layout Tool is the graphical equivalent of Integrated Development Environment auto formatting of textual programming languages that we now take for granted.

5 Conclusions

Automation is an essential part of any software development process. This paper presents a set of tools that help automate some traditional software engineering practices when designing with Simulink. The practicality of the tools has been proven on large industrial models from the automo-

tive industry as demonstrated in the paper. The concepts presented in this paper, and tools based on the concepts, represent only a first step in our investigation of the issues of integrating some traditional software engineering practices in design with Simulink.

References

1. Auto Layout Tool: http://www.mathworks.com/matlabcentral/fileexchange/51228-auto-layout-tool (2015) (**online**). Accessed Feb 2016
2. Bender, M., Laurin, K., Lawford, M., Ong, J., Postma, S., Pantelic, V.: Signature required: making Simulink data flow and interfaces explicit. In: Proceedings of 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), pp. 119–131. SCITEPRESS (2014)
3. Bender, M., Laurin, K., Lawford, M., Pantelic, V., Korobkine, A., Ong, J., Mackenzie, B., Bialy, M., Postma, S.: Signature required: making Simulink data flow and interfaces explicit. Sci. Comput. Program. **113**(Part 1), 29–50 (2015)
4. Chen, C.q., Ji, Y.: Modular aircraft simulation platform based on Simulink. In: IEEE International Conference on Mechatronics and Automation (ICMA 2010), pp. 1454–1459 (2010). doi: 10.1109/ICMA.2010.5589162
5. Data Store Rescope Tool: http://www.mathworks.com/matlabcentral/fileexchange/51160-data-store-rescope-tool (2015) (**online**). Accessed Sept 2016
6. Dajsuren, Y., van den Brand, M.G.J., Serebrenik, A., Roubtsov, S.: Simulink models are also software: modularity assessment. In: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA), pp. 99–106. ACM (2013)
7. Fehér, P., Mészáros, T., Mosterman, P.J., Lengyel, L.: Flattening virtual Simulink subsystems with graph transformation. CoSMoS **2013**, 39 (2013)
8. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P.: A technique for drawing directed graphs. IEEE Trans. Softw. Eng. **19**(3), 214–230 (1993)
9. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice-Hall, Englewood Cliffs (2002)
10. Hunt, A., Thomas, D.: Ubiquitous automation. IEEE Softw. **19**(1), 11 (2002)
11. Klauske, L.K., Dziobek, C.: Improving modeling usability: automated layout generation for Simulink. In: Proceedings of the MathWorks Automotive Conference. MAC (2010)
12. Klauske, L.K., Schulze, C.D., Spönemann, M., von Hanxleden, R.: Improved layout for data flow diagrams with port constraints. In: Cox, P., Plimmer, B., Rodgers, P. (eds.) Diagrammatic Representation and Inference. Lecture Notes in Computer Science, vol. 7352, pp. 65–79. Springer, Berlin (2012)
13. Model Engineering Solutions: MES Model Examiner (MXAM DRIVE). http://www.model-engineers.com/en/model-examiner.html (2014) (**online**). Accessed Sept 2014
14. Pantelic, V., Postma, S., Lawford, M., Korobkine, A., Mackenzie, B., Ong, J., Bender, M.: A toolset for Simulink: improving software engineering practices in development with Simulink. In: Proceedings of 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), pp. 50–61. SCITEPRESS (2015)
15. Quante, J.: Views for efficient program understanding of automotive software. Softwaretechnik-Trends **33**(2) (2013)
16. Reach/Coreach Tool: http://www.mathworks.com/matlabcentral/fileexchange/51180-reach-coreach-tool (2015) (**online**). Accessed Feb 2016
17. Rau, A.: On model-based development: a pattern for strong interfaces in Simulink. Gesellschaft für Informatik, FG **2**(1), 12 (2002)
18. Reicherdt, R., Glesner, S.: Slicing MATLAB Simulink models. In: Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012), pp. 551–561. IEEE Press, Piscataway, NJ, USA (2012)
19. Signature Tool: http://www.mathworks.com/matlabcentral/fileexchange/49897-signature-tool (2015) (**online**). Accessed Feb 2016
20. Sankaranarayanan, H., Kulkarni, P.A.: Source-to-source refactoring and elimination of global variables in C programs. J. Softw. Eng. Appl. **6**(4), 264–273 (2013)
21. Smith, A.R., Kulkarni, P.A.: Localizing globals and statics to make C programs thread-safe. In: Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2011), pp. 205–214. ACM, New York, NY, USA (2011). doi:10.1145/2038698.2038730
22. The MathWorks: Model Advisor: http://www.mathworks.com/help/simulink/ug/consulting-the-model-advisor.html (2014) (**online**). Accessed Sept 2014
23. The MathWorks: Verification and Validation Toolbox. http://www.mathworks.com/products/simverification/ (2014) (**online**). Accessed Sept 2014
24. The MathWorks: BOT. http://www.mathworks.com/matlabcentral/fileexchange/45670-bot (2015) (**online**). Accessed Oct 2015
25. The MathWorks: Data Store Diagnostics. http://www.mathworks.com/help/simulink/ug/using-data-store-diagnostics.html (2015) (**online**). Accessed Oct 2015
26. The MathWorks: Japan MathWorks Automotive Advisory Board (JMAAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 4.01. www.mathworks.com/solutions/automotive/standards/maab.html (2015) (**online**). Accessed Feb 2016
27. The MathWorks: Simulink User's Guide. http://www.mathworks.com/help/releases/R2015b/pdf_doc/simulink/sl_using (2015) (**online**). Accessed Sept 2015
28. The MathWorks: The MathWorks Automotive Advisory Board, Version 3.0. http://www.mathworks.com/automotive/standards/maab.html (2015) (**online**). Accessed Feb 2016
29. Tran, Q.M., Wilmes, B., Dziobek, C.: Refactoring of Simulink diagrams via composition of transformation steps. In: The 8th International Conference on Software Engineering Advances (ICSEA 2013), pp. 140–145 (2013)