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.