Stupid Tool Tricks for Smart Model Based Design

Mark Lawford(✉)

McMaster Centre for Software Certification, McMaster University,
Hamilton, ON L8S 4K1, Canada
lawford@mcmaster.ca

Abstract. Formal methods tools can be used to detect and prevent errors so researchers assume that industry will use them. We are often frustrated when we see industrial projects where tools could have been used to detect or prevent errors in the final product. Researchers often fail to realize that there is a significant gap between aa potentially useful tool and its use in a standards compliant, commercially viable, development process. In this talk I take a look at seemingly mundane industrial requirements - qualification (certification) of tools for use in standards compliant development process for general safety (IEC 61508), Automotive (ISO 26262) and Avionics (DO-178C), Model Based Design coding guidelines compliance, standards compliance documentation generation and integration with existing industry partner development processes. For each of these topics I show how "stupid tool tricks" can be used to not only increase adoption of academic methods and tools, but also lead to interesting research questions with industry relevant results.

Keywords: Simulink · Model-based design · Tool qualification · Software tools

1 Introduction

The title of this talk is based upon the former television host David Letterman's popular "Stupid Pet tricks" segment from the Late Show where people brought out their pets to perform various tricks. In introducing the segment during the November 15, 2013 show, Letterman described the segment as follows:

> Now please, the pets are not stupid. The people who taught them the tricks are not stupid. It's just that it's a colloquialism for ... "Oh! Isn't that cute!"

In the remainder of the paper I will briefly describe joint work with colleagues and students from industrial research projects that form the basis of the "Stupid Tool Tricks" I refer to in the title. In order to avoid confusion about my opinion of the excellent people I get to work with and the high quality work they produce, let me rephrase Letterman's description:

© Springer International Publishing AG 2016
S. Blazy and M. Chechik (Eds.): VSTTE 2016, LNCS 9971, pp. 1–7, 2016.
DOI: 10.1007/978-3-319-48869-1_1

> Now please, the tools are not stupid. The people who programmed the tool tricks are not stupid. It's just that it's a colloquialism for ... "Oh! Isn't that useful!"

Recently embedded software development has turned to Model Based Design (MBD) with code generation from models created with tools like Matlab/Simulink. In recent talks John Knight has declared that "Coding is over!", basically saying that it doesn't matter what language you teach anymore. Java, Python, C, C# are irrelevant. What matters is models. Engineers will create models and generate the code. Or to think of it another way, "Coding is dead! Long live encoding! (of models ... in MATLAB/Simulink)". As a result managers might think that we do not need software engineers any more. While domain engineers may create the models, companies will still need the Software Engineers to help manage the models and abstractions.

A recurring theme at VSTTE is moving the focus up the levels of abstraction to a more productive layer closer to the engineering problem. For example, automotive controls engineers will provide insight into how to model and design control systems using the controls oriented models of Matlab/Simulink, but with the pace of development, diverse product lines and absolute need for dependable software and systems will require appropriate software engineering methods and concepts to be applied. From precise requirements to design for change, software engineering principles will need to be applied to the models. A problem currently facing many industries is that the majority of engineers developing the models are not taught Software Engineering fundamentals such as those pioneered by Parnas [4]. With the move to Model Based Development, coding is mostly over. Software Engineering is *definitely* not over.

It is clear that many industries need help in dealing with Model Based Development of software. So why is the industry not using researchers tools, theories and methods that are promoted at conferences and workshops like VSTTE? In the remainder we provide some possible answers to this question.

2 What Is Tool Qualification? (and Why Should I Care About It?)

In a nutshell? Tool qualification comes down to insure that the tool is fit for use in the intended development context. Researchers should care about it because it is one of the biggest hurdles to getting their tools and theories used.

Figure 1 represents at the top level what is the main hazard that is of the utmost concern for most standards regarding tool use - the tool fails to detect an error or inserts an error. A rudimentary interpretation of the *DO-330 Software Tool Qualification Considerations* supplement to the DO-178B/C standard that is applied to civilian aviation software provides the second level possible causes that can lead to this hazard. The diagram then provides some detail of the third level of what could lead to the tool not being properly installed that then results in the tool failing to detect an error or inserting an error. For the purposes of

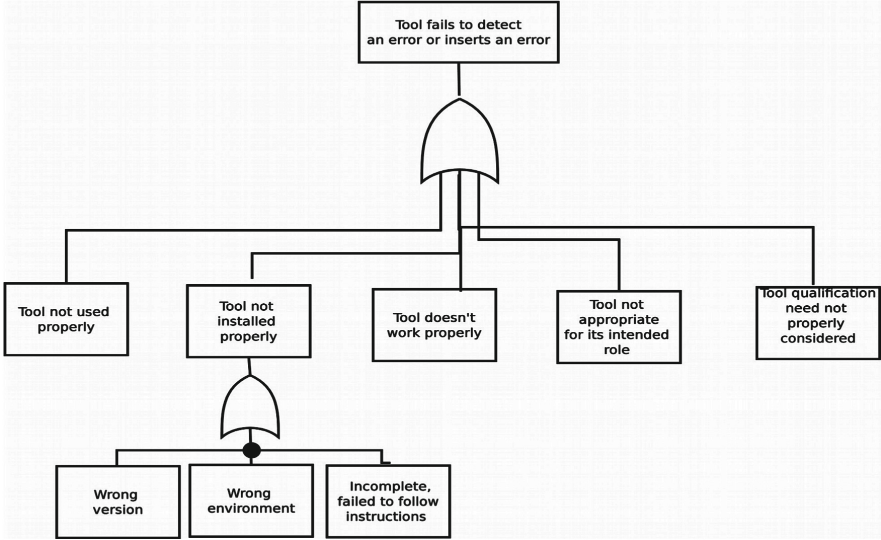

Fig. 1. Representation of tool hazard analysis implicit in DO-330

brevity we have not provided the complete third level expansion. Other standards such as IEC 61508 and ISO 26262 have similar reasoning behind their need for tool qualification before the tools are used in the development of a critical system.

The successful use of the PVS theorem prover to perform software verification of the Darlington Nuclear Reactor Shutdown Systems software has been documented in [6] and a description of how consideration of the entire development process was important to that success can be found in [7]. One of the key insights is that while the use of a formal methods tool like PVS provided increased confidence and considerable benefits, the final development process accepted by the regulator required all of the proofs done in PVS to be performed manually too in order to mitigate any potential failures of PVS and the supporting tool chain used in the design verification of Darlington. Tools are great, but they do not buy you as much as you think if they can be a single point of failure. At the time of the Darlington Redesign Project the regulator wanted to mitigate a failure of PVS with a known method, manual proof. It was a reasonable requirement at the time, but it limited the benefits of the formal methods tools.

In the intervening years since the Darlington Redesign Project was completed, standards have evolved to provide better guidance to engineers wishing to use software tools. For example, the Latest version of IEC-61508-3 now provides better guidance here:

7.4.4.5 An assessment shall be carried out for offline support tools in classes T2 and T3 to determine the level of reliance placed on the tools, and the potential failure mechanisms of the tools that may affect the executable software. Where such failure mechanisms are identified, appropriate mitigation measures shall be taken.

NOTE 1 Software HAZOP is one technique to analyse the consequences of potential software tool failures.

NOTE 2 Examples of mitigation measures include: avoiding known bugs, restricted use of the tool functionality, checking the tool output, use of diverse tools for the same purpose.

In particular, Note 2 suggests checking of the tool output or use of diverse tools for the same purpose. DO-330 (S. 4.4(e)) in the avionics domain and ISO 26262 (clause 11.4.1.1) in the automotive domain provide similar guidance in avoidance of single points of failure in development tool chains.

2.1 Solving the Tool Qualification Problem

The bad news is that in order to get your tools and methods used, you will, in all likelihood, need to use two different (diverse) tools in order to avoid having to do work manually because "demonstrating soundness of the tools" to a regulator in a cost effective way will likely be difficult or impossible. The good news is that it is not as hard as you might think to knock the tool qualification requirements down a level by doing the same thing with 2 or more tools. Intermediate Domain Specific Languages (DSLs) can be used to generate code for multiple theorem provers, SMT solvers, or model checkers, often providing more than one way to get the same result. This technique has the additional benefit that it can help avoid vendor lock-in for verification tools.

In developing tools and methods, researchers need to consider this tool qualification problem if they want industry to use their work. In developing the Tabular Expression Toolbox for Matlab Simulink [2] this was the main impetus behind having the completeness and disjointness conditions checkable by both PVS and an SMT solver. This could then be used as part of an argument to the regulator the checks for domain coverage and determinism of the specification would not need to be manual checked. This brings us to

Stupid Tool Trick #1

Do everything twice in two different ways.

3 Integrating with the Development Process and Documentation

Recent industrial research projects have given us access to a large number of industrial Matlab/Simulink models that are used for code generation. In an effort to understand those models, we began examining the explicit dataflow due to model input/output ports and implicit data flow due to (scoped) data stores and goto/from blocks in Simulink [1]. This led to the development of tools for Matlab/Simulink that help with model comprehension and refactoring for improved software qualities such as model comprehension, testability, modularity [3].

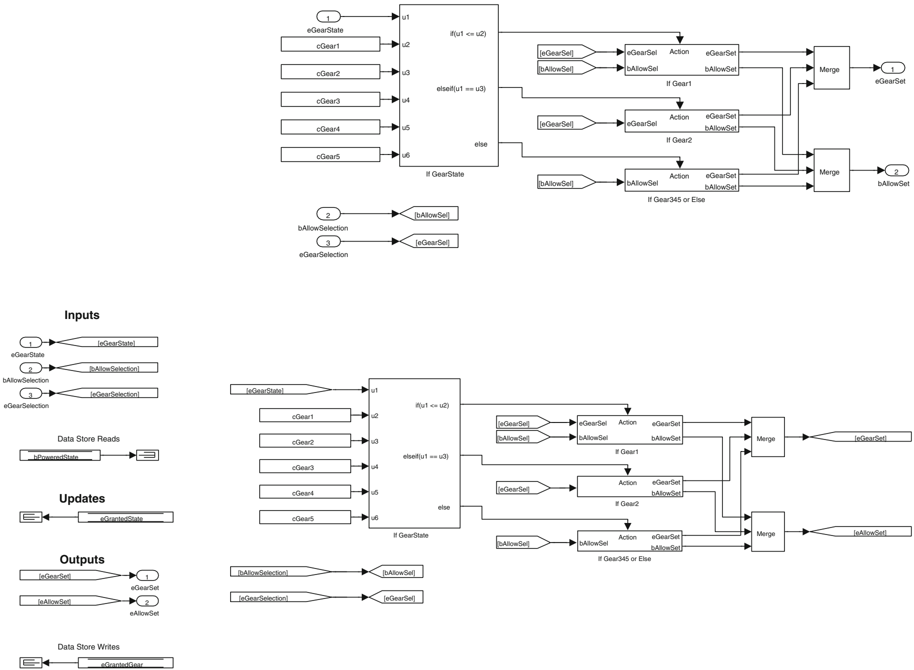

Fig. 2. Original Simulink model (top) and with "signature" (bottom)

One tool in particular, the signature tool, made all of the dataflow explicit by modifying the models to create explicit ports for all of the dataflow, explicit and implicit, on the left side of the model (see Fig. 2). One could think of this as the equivalent of a function prototype in a C header file. The industry partner did not want to modify the layout of their models because of potential code generation impacts so the tool was initially rejected, but later found use as a test harness generator when we demonstrated significant improvements in test coverage, with reduced testing effort, when the signature was used to help make dataflow explicit to commercial test case generation tools.

Stupid Tool Trick #2

> Consider alternative uses of a tool. These might be more useful than your original purpose.

4 Coding Guideline Compliance and Research

During our examination of implicit dataflow in Simulink models we noted that many of the models developed by domain experts tended to have the majority of their data store declarations at the top level of the model hierarchy. This is equivalent to programming with global variables. Data stores, like variables

in traditional programming languages, should be restricted in scope in order to avoid inadvertent or unwanted access and help to make the design more modular.

We developed a tool that examined the dataflow and determined where the data stores were actually accessed and then rescoped the data stores to be as low as possible in the model hierarchy. This tool was initially called the Data Store Push-Down Tool and has since been renamed the Data Store Rescope Tool since it can also move a data store declaration higher up in the model hierarchy if access is added a part of the model that is not below (i.e., in the scope of) the data store declaration. Since the development of this tool modeling 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 data store declarations [5].

Noticing that some models had well scoped data store declarations and were hence easier to understand, we then developed a metric that computed the difference between the number of data items that subsystems had access to and the number that it actually used. A lower the total difference might then be an indicator of better model quality. This in turn has led us to reconsider how Simulink models can be developed to embody the software engineering principles such as those in the works Parnas *et al.* [4].

Stupid Tool Trick #3

If a tool is useful, ask yourself why is it useful. This might lead to interesting research ideas.

5 Conclusion

By working with our industrial partners we were motivated to discover simple "stupid" tool tricks that improved the applicability of research tools, helping to improve software engineering methods for Model Based Design and led to interesting research problems that had industrial relevance. The reader is encouraged to examine their own research tooling efforts in the context of industrial development to see if similar stupid tool tricks can lead to improved industry uptake of research results.

Acknowledgments. The author would like to acknowledge the work of all of the researchers and students in the McMaster Centre for Software Certification (McSCert). This work would not have been possible without the support of our industry partners.

References

1. 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. In: Science of Computer Programming, Part 1, vol. 113, pp. 29–50 (2015). Model Driven Development (Selected & extended papers from MODELSWARD 2014)

2. Eles, C., Lawford, M.: A tabular expression toolbox for Matlab/Simulink. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 494–499. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20398-5_38
3. 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: 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 50–61. IEEE, February 2015
4. Parnas, D.L.: Software design. In: Hoffman, D.M., Weiss, D.M. (eds.) Software Fundamentals: Collected Papers by David L. Parnas, pp. 137–142. Addison-Wesley (2011)
5. The MathWorks. Japan MathWorks Automotive Advisory Board (JMAAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 4.01, March 2015. www.mathworks.com/solutions/automotive/standards/maab.html. Accessed Feb 2016
6. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45236-2_9
7. Wassyng, A., Lawford, M.: Software tools for safety-critical software development. Int. J. Softw. Tools Technol. Transf. (STTT) **8**(4–5), 337–354 (2006)