A Software Analysis Framework for Automotive Embedded Software

Jochen Quante

Robert Bosch GmbH, Corporate Research
Stuttgart, Germany

Jochen.Quante@de.bosch.com

Abstract

One major drawback of model based development is that support by software analysis tools is usually not available. This is because each modeling language would require specially crafted tools. We present a framework that circumvents this problem by allowing integrated analysis of different models from the automotive domain. It also exploits certain specialities of the domain to realize analyses that would otherwise not have been possible.

1 Introduction

The trend towards model-based software development is quite intense in the automotive domain. This paradigm shift has a lot of advantages. For example, domain specific languages allow engineers to express specifications in a language that is closer to their domain. However, it also brings a number of disadvantages with it. In particular, the models are often proprietary, and there is only limited tool support to deal with them. There are hardly any tools for analysis of such models. However, since these models are also software and underly the same effects as traditional software artifacts, they become large and complex as more and more changes and extensions are performed [6]. Maintenance and understanding of these models becomes more and more expensive and error-prone. Tools that support these tasks are needed - but it is uneconomic to come up with a new tool for every new modeling language. In the following, we present a framework that is capable of analyzing different kinds of models from the automotive domain.

2 Automotive Models

The modeling language ASCET[1] is designed for developing embedded automotive software. It supports block diagrams, state charts, ESDL code, and C code. Block diagrams support a data flow oriented view on the software and are quite close to what electrical engineers are used to work with. State charts can be used to specify program logics. ESDL is a language that is similar to Java and allows textual specification of rather control flow centric functions. C code can also be embedded. All these artifacts are translated to C code and integrated on the C code level. However, for analyses, it is desirable to do that on the model level, because this is the level that the developer works with.

[1] http://www.etas.com/en/products/ascet_software_products.php

3 Architecture

The basic idea is to have an intermediate representation that is capable of expressing certain aspects of the different input models in a uniform way – similar to what has already been done for integrating different textual languages [5]. Since all these languages end up in C code, an intermediate representation on the basis of C code constructs can be used for procedural aspects and interfaces. Additionally, one must then ensure traceability – i. e., support navigation from the intermediate representation back to the original model. This representation is then adequate for performing uniform analyses on all the different input models – and also when they cross modeling language borders. Additional languages can then be easily added by implementing another frontend that transforms the input model to the uniform intermediate representation.

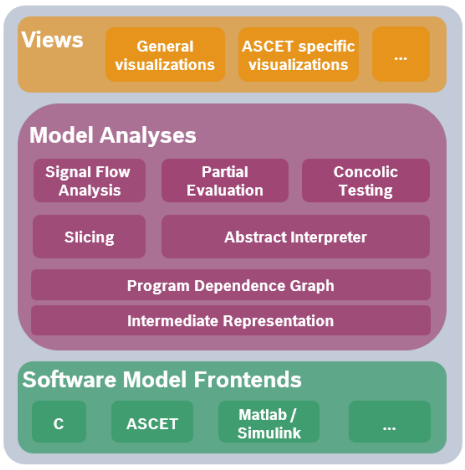

Figure 1: Framework architecture.

The central layers of the architecture as shown in Figure 1 build on the common intermediate represen-

tation and provide advanced analyses. The advantage here is that all the analyses are readily available for every modeling language that will be added in the future. On top of all that, there are visualizations. Those are partly input model specific, like results that are shown as an overlay in a block diagram, while others are independent visualizations of analysis results.

4 Basic Analyses and Interpreter

On this common basis, we perform a number of standard analyses, like control and data flow analysis and alias analysis. The result is a structure that is similar to the well-known program dependence graph (PDG) [2]. The PDG can directly be used for slicing. Several PDGs that also may originate from different models can be linked together (using inlining) to form a single large PDG that can then be used for further analyses. Inlining is possible here since recursion is not allowed in our software.

Our framework implements a relaxed version of **abstract interpretation**, compared to Cousot's original idea [1]. For us, it is basically the interpretation of a computer program using *some* abstraction for values and calculations. It need not necessarily be monotonic functions over ordered sets. Of course, this implies some differences. For example, loop handling is different (there is not necessarily a fix point). Since we are mostly concerned with analyzing software that contains very few and limited loops [7], this seems to be okay. On the other hand, we gain much more freedom and flexibility in the choice of the abstraction, and we will see the benefit of that in the use cases. The abstract interpreter performs abstract operations on abstract values and uses an abstract storage model. The freedom of choice of the abstraction makes it very flexible. For example, the same interpreter is used for data flow analysis, partial evaluation, interval arithmetics, and concolic testing [3] – by just using different abstractions.

5 Use Cases

The analysis framework is meanwhile used for a number of standard use cases as well as for individual (scripted) analyses. The very first and obvious use case is **signal flow analysis**, which corresponds to slicing of models. Slicing simply means to calculate all reachable nodes in the PDG from a given point. Given a starting point, e. g., in an ASCET block diagram, we just need to identify the corresponding node in the PDG, calculate all reachable PDG nodes, and map the result back to the original model. This highlighting of signal flows turned out to be very helpful especially for calibration engineers, who have to find out which parameters (that they have to adjust) influence which output values.

Another use case of the PDG is **C code visualization**, which means transforming the code to a block diagram representation. Since block diagrams emphasize data flows, and data flows are explicitly represented in the PDG, the transformation is largely straight-forward. These two use cases are meanwhile available as a feature of ETAS' eHandbook[2].

A third use case is **partial evaluation**. Partial evaluation is the execution of program code with partially existing but incomplete assignments of variables occurring in a program with concrete values [4]. By assigning values to only few of the variables, it is often possible to largely reduce the model. This is done by graying out parts of the model that are not executed in the given setting. This enables developers to understand a model much more quickly since they can concentrate on certain scenarios without having to manually interpret the program. It is also helpful to be able to assign values to some variables and see the effect on a local result without having to define the entire environment. The latter is often quite complex and tedious in automotive software.

6 Conclusion

We have presented the basic ideas of our software analysis framework. The main advantage of having a specialized framework within Bosch are that it can exploit the specialities of automotive software, and that it can deal with upcoming models. Meanwhile, it is available for everyone within Bosch ("Bosch internal open source"). It is now used for many different purposes like test case generation, multicore analyses, security analyses, and others.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of 6th POPL*, pages 238–252, 1977.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[3] A. Hoffmann, J. Quante, and M. Woehrle. Experience report: White box test case generation for automotive embedded software. In *Workshop Proc. of ICST, TAIC PART*, 2016.

[4] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, June 1993. Available from `http://www.itu.dk/people/sestoft/pebook/`.

[5] R. Koschke, J.-F. Girard, and M. Würthner. An intermediate representation for reverse engineering analyses. In *Proc. of 5th WCRE*, pages 241–250, 1998.

[6] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change.* Academic Press Professional, Inc., 1985.

[7] V. Schulte-Coerne, A. Thums, and J. Quante. Challenges in reengineering automotive software. In *Proc. of 13th CSMR*, pages 315–316, 2009.

[2] `http://www.etas.com/en/products/ehandbook-details.php`