Intents, Upgrades and Assurance in Model-Based Development

Jonas Elmqvist, Simin Nadjm-Tehrani
Dept. of Computer and Information Science
Linköping University, Sweden
[jonel,simin]@ida.liu.se

Abstract

This position paper addresses topic 1 of the workshop: MoDES challenges in industrial practice. It highlights lessons learnt from three applications of model-driven development for software embedded in vehicular safety restraints, aerospace and secure radio communication systems. While our experiences in these three fields of application are compared and contrasted the emphasis will be placed on the specific requirements of safety-critical software in aerospace systems with the three characteristics: long life, high level of assurance, and necessity of efficient upgrades of individual components. A typical application in this class is also large, complex, and heterogeneous.

1. Introduction

Model-based development of embedded software is promoted as a means to achieve cost-efficient development of code, and platform independent design. In this paper we summarise our experiences from three case studies in the past year and point out the important aspects that need to be strengthened in today's tools before the vision of model-driven embedded system development can be a reality in high assurance systems. All three applications were in domains that some element of assurance is present: a future car airbag system being developed at the Swedish subsidiary of the company Autoliv [1], an encryption terminal (Tiger XS) for secure communication on top of any communication equipment at the company Sectra [2], and a sanitised version of an unmanned vehicle with multi-mode control (also human operated) at Saab Aerospace [3]. All three case studies were aiming to ascertain the benefit of the current modelling environments to the developers of these systems.

The three application areas also have characteristics that are distinctive for each. In the airbag system the main goal was to enable rapid product development amid changing technology. Thus, the company requires a faster and more reliable means of porting a subsystem that was e.g. developed for a 16 bit processor that has 128kb ROM to a 32 bit processor with 256kb ROM. For them, automatic code generation was studied as a means to increase efficiency in product development. Another main characteristic was the timeliness requirements – having 30ms between the crash detection and firing of a restraint implies that some algorithms have to be computed predictably (and within 500μs).

In the Tiger XS communication platform the main requirement is support for platform independence and security assurance. Tiger XS acts as a component in defence systems. It acts as a bridge that makes any secure application (e.g. encrypted phone calls or encrypted SMS) to run on top of any communication hardware (e.g. PDA or phone), and transforms "black" clear text data to "red" encrypted data. Hence, automatically generated code that refers to operating system primitives has to be easily adaptable to new underlying platforms. Moreover, the security-intrinsic applications demand that the generated code should follow a predefined coding style and be suitable for human inspection.

Both of above applications have small footprint requirements and thus essentially expect the size of the automatically generated code to be comparable to the hand-written code (the airbag system being at the extreme with its byte-optimal handwritten code). In the third category of systems, the aerospace related case study, footprint is a less dominant requirement. Instead the long life time and the safety-critical requirements of the system imply that any upgrades made to the system over its lifetime should be easily traceable to the original intent specifications, and efficiency in code generation has to be followed by efficiency in the verification process, assuring that component upgrades do not jeopardise system level safety requirements.

2. Essential properties

From the three case studies one could conclude that automatic code generation (to enhance shorter time to development) is an essential property of all tools that intend to bridge the gap between user level requirements and the implemented code. Safety-critical code, has however, the additional characteristic that the original sources of its

requirements, often linked to system level hazard analysis and mitigation of fault/error scenarios by architectural solutions, need to be clearly documented as intents, and traced to any future changes in the design or implemented code. Moreover, all changes to the design are followed by studying their impact on the documentation of the safety case. Supporting formal verification by a modelling tool makes the extra difference in this context. Our study of the tools that could be applied within each application domain rests on the above properties.

In what follows we give a short exposure to the sanitised example of the unmanned vehicle that was provided by Saab. This example, although much simpler than any realistic aerospace application has some elements that illustrate the need for (1) support for intent specifications and tracing the system level requirements over a long lifetime, including the need for tracing changing requirements all the way down to new design models of upgraded components, and (2) the necessity of support for formal verification to achieve efficient verification of safety-related properties; in particular, incremental verification of such properties upon component upgrades.

3. Support for upgrades

The development process followed by most companies today, at least in the safety-critical arena, follows what can be considered as a variant of V method. It essentially assumes a strict control of the integrator company over the developed components (in-house or subcontracted).

Model-driven tools and specially the UML-based support have grown from the world of software development, with the advent of object-oriented design in the last two decades. The safety requirements of aerospace systems can, however, hardly be traced to a software component alone. Software is typically not harmful to the environment and can only *contribute* to violation of safety. Achieving safety is typically ensured by a mix of architectural decisions [4] and rigorous process for system development based on functional decomposition. Examples of architectural decisions are incorporation of fault tolerance via redundancy, hardware interlocks as a backup for software failure, watchdogs, monitors, and so on. An interesting question is: how to support the engineers who primarily perform system development in the old worlds of structured design, to encompass the "new" world of software design, and link the two in the systems and safety engineering process? An orthogonal question is how to support the process of upgrading an existing component when new functional or safety requirements arise?

In current system development processes all the safety analyses, including fault-tree analysis (FTA), failure modes and effects analysis (FMEA), and component-level and system level verification has to be redone for every upgrade in the life cycle of the system. Model-based development needs to address how this process can be "shortened" by making an efficient analysis that assures preservation of safety properties.

Elmqvist [3] presents the Sigma development metaphor as a model for system level upgrades based on component updates (see Figure 1).

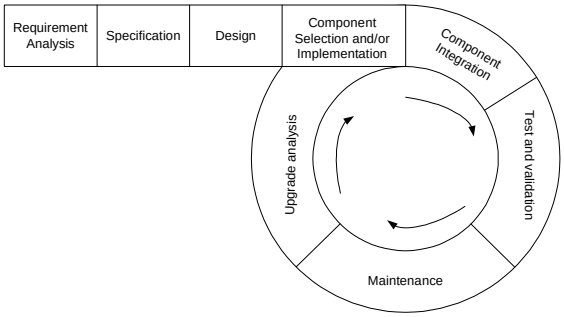

Figure 1. The Sigma development process model

The model captures the iterative upgrade process of existing components as well as the analysis processes that are essential during a system's life-cycle.

3.1 Example: unmanned vehicle

The unmanned vehicle is controlled by the *Remote Vehicle Control Unit* (RVCU). The vehicle operates inside a closed area (see Figure 2) consisting of a work area, a parking area and (stationary) obstacles. The vehicle can be controlled by the operator either hands-on with a joystick or by planning missions. Once leaving the parking area, the vehicle is not allowed to stop.

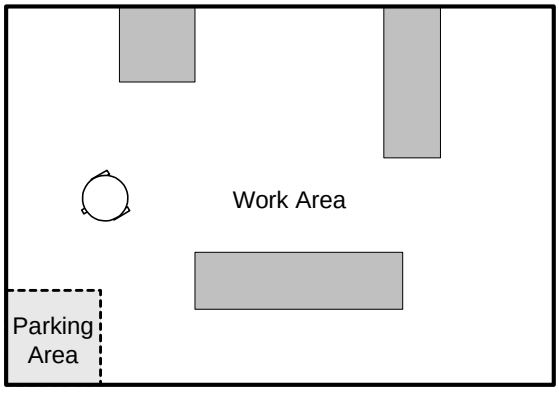

Figure 2. A possible environment with obstacles for the unmanned vehicle.

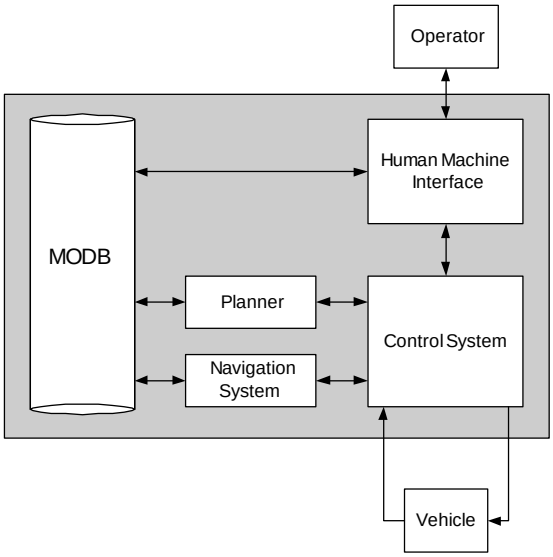

Figure 3. The RVCU architecture.

The role of the RVCU is to make sure that the vehicle is controlled safely inside the area i.e. avoiding collisions with obstacles and avoiding the vehicle to navigate outside the closed area.

The Dynamic Window Approach by Fox *et al* [5] was used for obstacle avoidance. The algorithm calculates an optimal trajectory by reducing the search space of possible velocities based on the dynamics of the vehicle and the position of the obstacles. The algorithm was slightly modified to fit in the context of the unmanned vehicle example.

The five components of the RVCU are presented in the architectural model in Figure 3:

- *The Human Machine Interface* is the interface between the operator and the system
- *The Map and Obstacle Database* (MODB) provides a representation of the map and the obstacles
- *The Planner* takes care of high-level mission planning during unmanned missions
- *The Navigation System* handles the final control of the vehicle by using the Modified Dynamic Window Algorithm
- *The Control System* is the coordinating module that besides interacting with both Planner and Navigation system also communicates with the robot

The RVCU can be considered a safety-critical real-time system as the collisions with the obstacles or moving outside the designated area can be considered to result in harming people or environment.

To illustrate life-time changes we have added a new system requirement, and considered the effects of an upgrade to satisfy that requirement on the existing design. The stationary obstacles in the first design are considered to be moving objects in the new upgraded version.

4. Intent Specifications

Intent Specifications [6] is a new approach for specifying and designing systems that is based on research both in system engineering and psychology. The primary difference from other approaches is the structure (see Figure 4). An intent specification is structured in six levels, each level answering the question *"why?"* i.e. providing intentions about the level below, as opposed to traditional specification methods where levels are divided into answering *what* to do from *how* to do it. Each level is mapped to levels below providing traceability of system goals and high-level requirements down to implementation and vice versa. Each level has its own view of the system and is a different model of the system [7].

- *Level 0* is the project management's view of the system.
- *Level 1* is the customers view, includes system goals, high-level requirements, hazards, design constraints, assumptions and system limitations.
- *Level 2* is the system engineer's view of the system and it describes the system design principles.
- *Level 3* describes the black box behaviour of the system and its modules. Formal analysis methods can be used on this level.
- *Level 4 – 6* provides information of physical and logical representation of the system down to implementation and maintenance information. These levels were not the focus of this study.

The tool used for implementing an intent specification was SpecTRM (Specification Toolkit and Requirements Methodology), a commercial tool from Safeware Engineering [8]. It is a document-oriented tool that works more or less as an advanced word processor and uses the intent specification methodology as a foundation with the seven levels as different chapters in the specification. The black box models in Level 3 are written in a specification and modelling language called SpecTRM-RL based on the state-based specification language RSML that essentially summarise state transitions using AND/OR trees [9].

The tool provides simulation and some static analysis of the SpecTRM-RL models.

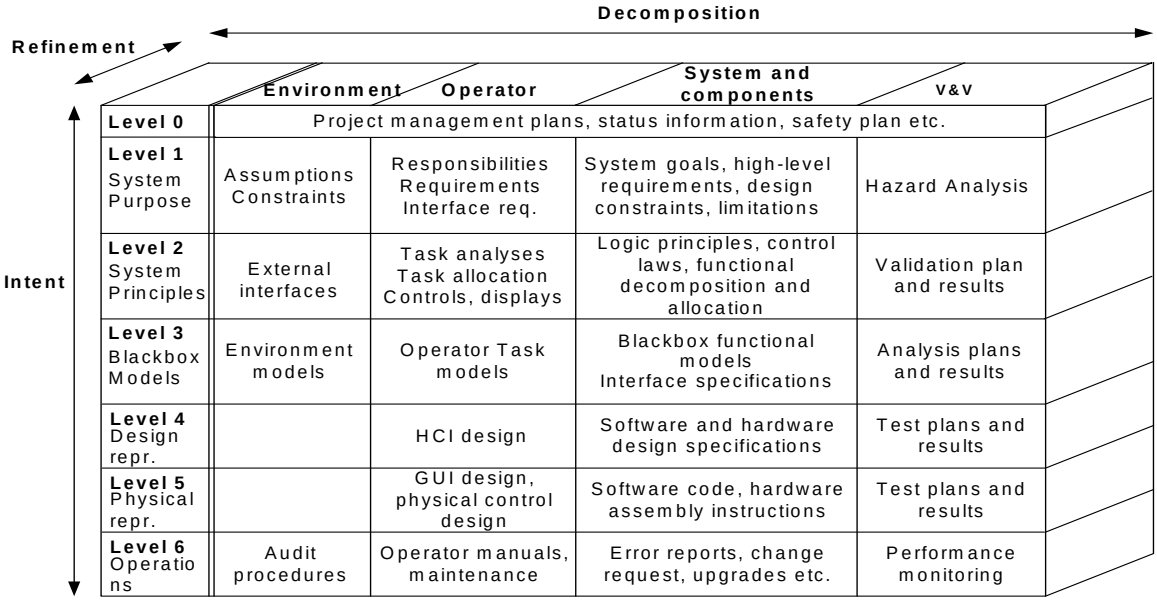

Figure 4. The structure of an Intent Specification [10].

5. From specification to Design

The third level of intent specifications is quite close to a design model but does not (yet) have the ambition of supporting model-driven development. The level 3 in SpecTRM provides an input-output interface for each component, and a description of the internal states and externally visible modes of the system. In addition, it gives a human readable logic for state transition conditions in terms of AND/OR tables. However, to go from intents to implementations, and in particular via designs whose dynamic properties are formally analysable, we need a bridge to a tool that supports both code generation and formal analysis.

In the RVCU case study we chose the Esterel Studio programming environment for further development of the model [11]. The choice was primarily motivated by the support for formal verification in Esterel, using the Prover plugin model checker that can deal with systems with large state spaces using Stålmarck's method [12]. It also exemplifies a tool that is suitable for this class of applications due to its ability to deal with heterogeneity. Esterel designs have Mealy machines as formal semantics and are as such suitable for hardware/software codesign. A high-level description of an application can after formal analysis be translated to code that is the basis of a software implementation (C code) or hardware implementation (VHDL code).

Another benefit of using this environment in our safety-related case study was that the same environment (the same design model) can in fact be used as a test bed for study of fault tolerance and failure mode analyses by systematically plugging in failure modes for various inputs or outputs of a component and studying the effects of single or multiple faults in terms of violations of safety at system level [13]. This combines model-based development with formal analysis of safety (in the spirit of FTA/FMEA) using the same design model, and without building fault trees.

6. Results and lessons learnt

This section outlines the results of application of model-driven development to all the three application domains, and in each case summarises the remaining challenges facing the application developer. We begin by the unmanned vehicle example as it was described in more detail here, and then briefly describe the comparison with the other two studies mentioned in section 1.

6.1 Upgraded unmanned vehicle

An upgrade of the requirement was done after the initial design and verification of the unmanned vehicle control system. Instead of having static obstacles inside the closed area, the vehicle should be able to avoid moving obstacles.

The study proved SpecTRM (the version of summer 2003) to be a rather immature tool and more suitable for the design of control-handling modules such as the *Control System* than data-intensive modules such as the *Navigation System*. Further,

SpecTRM does not yet provide any automatic traces or any overview of the traces i.e. the traceability must be created explicitly by the developer. A more sophisticated support for traces would be appropriate in order to make the tool suitable for industrial use.

By manually converting the SpecTRM-RL models to Esterel modules, Esterel Studio could be used to verify the system and its components. Esterel Studio and its built-in model checker were able to prove a majority of the control properties of the RVCU. However, Esterel Studio does not provide any framework for modular or compositional verification. After the upgrade, the original verification process had to be redone. Furthermore, dealing with numerical properties in the current version of the Prover plugin was insufficient for our purposes.

A tool environment aiding the developer beginning with a specification language such as SpecTRM-RL down to code generation is needed. The unmanned vehicle example presents such a framework that combines SpecTRM and Esterel Studio, but with lack of automatic translation between the tools, the environment is still incomplete. A positive aspect of the SpecTRM implementation is its use of the Java-based Eclipse [14] environment that allows plugin translators to be added conveniently. A trial plugin, SpecTerel, that automatically translates very simple SpecTRM-RL models to very simple Esterel models was implemented within a few days as a proof of concept.

6.2 Secure communication platform

The Tiger XS module is a software platform that provides a middleware function in a larger software system development. Here it was obvious that tools that support the object-oriented design process are the main candidates. Among the UML based tools for code generation two representatives were studied in terms of the requirements of the case study: Rhapsody from iLogix [15] and Visual State from IAR systems [16]. However, in both cases it was found that both tools provide both too much and too little support.

Rhapsody provides too much support in the sense that it has a powerful extended UML language having a comparatively steeper learning curve. However, the primary reason for not being considered as a candidate for tool generation at Sectra was that it targets complete systems and some difficulty was experienced in merging existing (legacy) C-code for other parts of the application and the automatically generated Target XS code.

Visual state, on the other hand, was a light-weight tool with little extra functionality. In particular, it was possible to adjust the coding style to that required by Sectra by implementing the translation of the action language (the part that defined effects of state transitions in terms of new value assignments to variables) so that it suits the in-house requirements. The main weakness of the tool in this specific case was the support for integration of the generated C-code with other legacy code, and in particular, that it was cumbersome to use user-defined types. Also, the automatically generated code was organised in terms of a number of arrays that were not so human-readable and satisfactory for security assurance-related inspections.

Although both tools were considered to generate small enough footprint compared to the Sectra hand-written code, they were not adopted due to the above reasons. To suit the needs of this case study, in the end, an interpreter-based translation scheme was deemed the most useful. It resulted in an in-house code generator based on a subset of Statecharts [2].

6.3 Airbag software

Rhapsody was also tested as a candidate for code generation for the air bag software. Here, the architecture of the system was clearly divided in two types of modules: those that were control intensive and those that were data intensive. For the control intensive parts the abstract modelling in Rhapsody in C, both in terms of class diagrams and Statecharts were found useful, and the automatically generated code was tested on a target micro controller (TX 19A). For the data intensive parts, the code that implements signal processing algorithms to detect when the vehicle is in crash, another tool that is closer to the data flow abstractions used by the control engineer was deemed useful. The tool Scade [11] was studied for code generation in this part of the application. Another useful feature of Scade was the formal verification support with the Prover plugin that was tested to a limited extent on the crash algorithm model.

The use of both modelling languages was found to reduce the time for development of code (after excluding the learning time). For a particular airbag function, this gain was quantified as a 60% decrease compared to the estimated time taken for hand written code. The main drawback for the Rhapsody-generated code was the code size, still a significant factor in choosing such technology in the air bags systems. The low cost constraints of the ROM violates this option, as the size of the generated code was twice as large as the optimised hand-written code. In both cases support for timing analysis of the air bag software was missing, and needs to be performed separately.

6.4 Final remarks

Our studies support the claim by providers of tools for model-based development in that these tools do indeed reduce the time taken for development of executable target code from high-level models that are easier to inspect, to communicate, and to use as a documentation of a complex system. The needs of various application areas in terms of requirements on the generated code were illustrated by three examples ranging from very tough code size (memory) restrictions in the air bag system to less demanding requirements on code size in the secure communication support and the unmanned vehicle case.

Support for documentation of upgrades in a long life time, and in particular when the traceability of the rationale of early design decisions and intents is a prerequisite to maintaining safety arguments is an obvious shortcoming of the pure "code-generators" in model based development today. Where intents can be documented and traced (e.g. SpecTRM), the support for code generation, formal verification, and compositional analysis of upgrade effects are still missing. Where design level models were the starting point (Rhapsody, VisualState, Esterel, Scade), no such support for component upgrade verification or longer term documentation of intents and safety-related arguments were part of the picture. In two of the applications we observed the dichotomy between data-flow abstraction and the state-based control abstraction. These issues still need to be addressed in tools that aim to support development of complex and heterogeneous systems. In particular, combining the worlds of structural design (hardware and mechanics) and object-oriented design (software) is a key to model-based development of high assurance systems.

Acknowledgements

This work was supported by the Swedish strategic research foundation project SAVE on component-based safety-critical vehicular systems, and the national aerospace research program NFFP3. The lessons learnt were partly based on the work of two Masters students Andreas Eriksson and Anders Grahn who did their final year projects under the supervision of the second author. The experimental SpecTerel plugin translator was developed by Erik Sundvall.

References

[1] A. Eriksson. *Model-based Development of an Airbag Software*. Thesis number LiTH-IDA-EX--04/025--SE. *Dept. of Computer and Information Science*, Linköping University, March 2004.

[2] A. Grahn. *Code Generation from High-level Models of Reactive and Security-intrinsic Systems*. Thesis number LiTH-IDA-EX--04/030--SE. *Dept. of Computer and Information Science*, Linköping University, April 2004.

[3] J. Elmqvist. *Analysis of Intent Specification and System Upgrade Traceability*. Thesis number LiTH-IDA-EX--03/074--SE. *Dept. of Computer and Information Science*, Linköping University, Dec 2003.

[4] J. H. Lala and R. E. Harper. "Architectural Principles for Safety-Critical Real-Time Applications", *Proceedings of the IEEE*, 82(1):25—40, Jan. 1994.

[5] D. Fox, W. Burgard, and S. Thrun. The Dynamic Window Approach to Collision Avoidance, *IEEE Robotics & Automation Magazine*, 4(1): 23—33, March 1997.

[6] N. Leveson. Intent Specifications, An approach to Building Human-Centered Specifications. *IEEE Transactions on Software Engineering*, 26(1):14—35, Jan. 2000.

[7] L. Grady, J Howard, and P. Andersson. Safety-Critical Requirements Specification and Analysis Using SpecTRM. *In Proceedings of the 2nd Meeting of the US Software System Safety Working Group*, Feb 2002.

[8] SpecTRM, http://www.safeware-eng.com, accessed 14th April 2004.

[9] M. P. E. Heimdahl and N. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Transactions on Software Engineering*, 22(6):363—377, 1996.

[10] Safeware Engineering. SpecTRM User Manual, 2003.

[11] Esterel Technologies. http://www.esterel-technologies.com, accessed 14th April 2004

[12] M. Sheeran and G. Stålmarck. A Tutorial on Stålmarck's Proof Procedure for Propositional Logic. *In Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Springer Verlag, 1998.

[13] J. Hammarberg and S. Nadjm-Tehrani. Development of Safety-Critical Reconfigurable Hardware with Esterel. *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. Elsevier, June 2003.

[14] Eclipse. http://www.eclipse.org, accessed April 2004.

[15] ILogix. http://www.ilogix.com, accessed March 2004.

[16] IAR. http://www.iar.com, accessed March 2004.