Intents and Upgrades in Component-Based High-Assurance Systems

Jonas Elmqvist[1] and Simin Nadjm-Tehrani[2]

[1] Department of Computer and Information Science, Linköping University
jonel@ida.liu.se
[2] Department of Computer and Information Science, Linköping University
snt@ida.liu.se

Summary. This chapter addresses challenges for model-driven development of embedded systems in industrial practice. These are rooted in the necessity of flexible development of new functionality at low development cost. Where a dependability requirement is added, e.g. support for assurance of safety requirements, then extending functionality by pluging in a new component, or modifying an existing component, without extensive safety-related and fault tolerance tests, is far from today's industrial practice.

The chapter highlights lessons learnt from three applications of model-driven development for high-assurance software components. The components were embedded in vehicular safety restraints, aerospace, and secure radio communication systems respectively. 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 following three characteristics: long life, high level of assurance, and forthcoming demands on efficient upgrades of assemblies of components. We discuss the need for relating intent specifications to formally verified design models from which safety-critical code is generated.

Key words: model-driven development, embedded systems, high-assurance, intent specifications, code generation, verification, component upgrades

1 Introduction

Model-based development of embedded software is promoted as a means to achieve cost-efficient development of code, and platform independent design. If successful, the model-based approach is a means of realising the "correct by construction" philosophy whereby flaws in a product design are discovered early at the design stage. Once adequate analysis on the design models assures adherence to system requirements, generating executable code is a systematic process that translates models to platform-independent and platform-dependent parts respectively. The idea is very attractive amid the increasing drive for higher efficiency in producing embedded software. Software is

undoubtedly the essential ingredient in introducing flexibility in product upgrades and achieving shorter time-to-market for novel products. The questions that are central to the studies in this chapter are: does model-based development of software aid in developing embedded software that has specific extra-functional requirements such as dependability and small footprint, and does it support future upgrades including integration with legacy code?

The chapter summarises experiences from three case studies performed during year 2003 and points 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 [Eri04], an encryption terminal (Tiger XS) for secure communication on top of any communication equipment at the company Sectra [Gra04], and a sanitised version of an unmanned vehicle with multi-mode control (also human operated) at Saab Aerospace [Elm03]. All three case studies aimed to ascertain the benefit of the current modelling environments to the developers of these systems, including the above criteria: efficient generation of usable code, ease of assuring dependability requirements, and support for system upgrades and integration.

The three application areas also have individual characteristics that are distinctive for the different classes. In the airbag system the main goal is 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 500 μs).

In the Tiger XS communication platform the main requirements are integration with legacy code, 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. Since an aerospace product has to go through well-established assurance prcedures and the certification of the new generation of a product is as strict as the certification of the earlier generation, the upgrade verification has a special status. 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. This requirement is the one that is least covered by the literature on model-based development and therefore deserves special attention in the current chapter.

2 Intents and upgrades

From the three case studies one could initially deduce that automatic code generation (to enhance shorter time to development) is an essential property of 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. Also, a modelling tool that supports formal verification makes the extra difference in this context.

Upgrades necessitate replacing one component with another, or modifying one component in some respect. The upgrade process is specially costly for safety-critical systems as much of the analysis and review of the safety arguments has to repeated for every significant change to the system. We believe that tool support in this industrial sector needs not only to encompass fast time-to-market and support for formal analysis, but also to support cost-effective upgrades. In this section we briefly review the potential approaches for developing high-assurance systems that are built from components. These may range over components that are acquired off-the-shelf (no source code) to components that are developed in accordance with model-based development techniques with design specifications and automatically generated code. We briefly cover two broad areas that are gaining attention in recent years: the component-based approach to development of software-intensive systems, and the approach that can be labelled as "constructing the correct".

2.1 Dependability and Components

The software engineering community [Szy02, CSSW04] is promoting methods for development of systems from components and is, to begin with, trying

to define the notion of components by providing a number of examples (e.g. [vOvdLKM00]). However, the emphasis in the works up to now has been on the efficiency in the development process as opposed to assurance procedures. Specially, the compositionality of the extra-functional properties has only recently gained attention. The problem with extra-functional properties is that they are typically defined at the system or service level. It is therefore not trivial to pin down the properties that a component should have in order to satisfy the system level properties when placed in the context of other components. In some sense there is an inherent conflict between separation of concerns (thereby restricting some design analyses to component level) and overall guarantees for system/service level properties.

Let us consider a few examples of non-functional properties. A prime concern in most high-assurance embedded systems is adherence to end-to-end timing constraints, and the two specific attributes of dependability [ALR01], namely reliability and safety. Real-time properties are cross-cutting concerns and it is not possible to assure a real-time service by a system assembled from components unless the component model captures the parameters needed for real-time analysis in a well-defined manner [TNHN04, AT05]. This is an area of work that has recently attracted attention with some progress in sight. But what can be said about component assemblies and assessing the safety of the system from characteristics of the components? The first obstacle that one meets is that assessing system safety and reliability has been predominant in hardware-intensive systems, thereby many approaches to assessment are hardware-oriented. Extending these methods in a systematic way to software components is only at early stages of research [Sch03].

A widespread myth about software is that the smaller the software unit the higher its reliability is. A study by Jones shows for example that the number of delayed and cancelled projects dramatically increases as the number of function points (a measure of size) increases beyond 5000 [Jon95]. At this level of complexity (roughly corresponding to 500k lines of code in languages like Fortran) 79% of the projects are cancelled or delayed by over 6 months. Thus, the ability to deliver a software intensive product that satisfies the specification efficiently is obviously a major problem.

A valid question is therefore: can we increase the reliability of a system by breaking it down to small manageable components? A follow-up question being: if we have demonstrated/estimated the reliability for a component how can we derive the reliability for the whole system based on a composition of the reliability measures for the parts. There are some initial attempts for answering these questions based on historical studies of modular designs. Hatton shows for example that size-complexity-fault frequency relation is not linear and there are some medium sized components that exhibit higher reliability compared to both smaller and larger components [Hat97]. With regard to aggregation at system level, Hamlet et.al. propose a theory for compositional calculation of reliability metrics based on component metrics. Nevertheless,

they contend that the theory needs to be validated in experimental settings [HMW01].

Safety is the ability of a system to avoid harm to people and environment. Hence, a car that never starts may seem to be safe by definition (although not quite reliable!). However, a car that does not start can indeed pose a threat to safety if it happens to stall on a railway crossing. In both cases the car fails to meet its reliability requirements. In the latter case it creates a potential threat to safety. Thus, safety is a property that intrinsically emerges from the behaviour of the system under design and the conditions in its environment.

Since there is no way that software on its own harms people or environment, it is incoherent to allocate attributes such as safety to a piece of software or digital hardware. Software or digital hardware can only be examined in terms of the ways they may contribute towards appearance of hazards. Hazards are failures that may potentially lead to violation of safety. Hence, traditional analysis of system safety starts by considering the potential unsafe scenarios, characterizing the risks for the hazard to take place (both in terms of probability and in terms of severity of consequences), and make a quantified decision on which scenario to consider as one that should never happen - no matter how the constituent components in the system are designed developed or operated.

Traditional analysis of system safety rests on techniques that focus specifically on "things that may go wrong". Fault-tree analysis (FTA) and Failure modes and events analysis (FMEA) are old techniques that grew within the era of building systems from hardware (mainly mechanical) components. One can contrast FTA and FMEA by considering one a top-down and the other as bottom-up. In other words, in FTA analysis one is interested to know given a potential failure in the system (a top level event) what are the combinations of conditions that necessarily cause that event. In FMEA one tries to consider each and every constituent of the system, and trace the effects of errors manifesting in that constituent. An interesting area of work is to extend the traditional safety analysis techniques so that they can indeed be applied to software-based systems too [HNT04, GKR05].

According to aerospace experts most errors are found in the interface between components; either because the original specification was incomplete (had forgotten to specify some aspect), or that it simply made wrong assumptions. A typical case is that one forgets the dependencies between several components, and when one component is updated/changed, the potential changes in other subsystems are not fully considered, or corresponding changes introduced there. An example is an attribute such as measured wheel velocity. If one reduces the number of pulses per rotation the resolution of the measurement is decreased. This might be favourable in terms of costs in the landing gear system, but the change might affect other consumers of the information. So the supplier of the landing gear system may move on to a cheaper realisation not considering the changes implied in the flight control system or pilot information system. The way such changes are propagated in the system are

by administrative processes: meetings, agreements, reviews. Thus, we are interested in studying whether component-based or model-based development paradigms can support the assurance procedures for a system that is upgraded by changing one or more of its components.

2.2 Constructing the correct

The traditional formal development process that can be labelled "correct by construction" has a natural extension in the model-based development paradigm. The claim is: if the design can be formally analysed, one can demonstrate that the known causes of failures that may violate safety, as identified by hazard and risk analysis, have been removed. Such a design model can thus be a sound basis from which automatic code generation can produce high-assurance components. A seemingly opposite approach that may be labelled by the slogan "constructing the correct" amounts to applying formal verification techniques directly to source code. Since this approach is gaining momentum in recent years, we believe that the proponents of model-based development need to consider the impact of these techniques in future development processes.

The "constructing the correct" school is both old and new. Early versions of the idea can be traced back to the works by Hoare and Floyd on formal analysis of programs (late 60's). In those years, the ability to reason about the behaviour of a structured program in terms of assertions at entry and exit points, based on deductions after execution of every program statement was put forward as a fundamental argument for constructing correct programs. One might argue that the early structured languages of 70's were equivalent to the high level modelling notations (e.g. UML family of languages) of today. Thus, early proponents of "constructing the correct" were indeed acting as today's proponents of the model-based formal verification (and thereby correct by construction idea). However, high level programming languages are no longer at the top of the abstraction hierarchy today. Instead, one could characterise the proponents of this approach by goal to find errors in real executable software in an efficient manner. The claim is that in the end what is running in the system to be delivered is the code. So, high assurance systems have to find a way to guarantee a predicable behaviour by the control software.

A good overview of the recent trend in program verification can be found in articles by Havelund, Visser, and co-authors [HV02, VHBP00] where it is clearly emphasised that the purpose of investigating verification techniques for program code is not that design verification is fruitless. Rather, it is recognised that many software engineers instead of detailing construction decisions in a design prefer to write the code directly. Also, tools that support design models e.g. UML allow the modeller to include code fragments in a UML model. Hence, verifying the code is promoted as an efficient assurance method. This

approach is followed for several programming languages, among them Java, C, Ada and hardware design languages e.g. [CCO02, BCC+02, CCG+04].

2.3 Dependable assemblies of correct components

Based on the short review above we believe that upgrading dependable systems based on updating and replacing components is an area that model-based development needs to address in the future. The overall methodology can be sketched as follows. Each developed component need to carry with it information about its effect on overall system behaviour with respect to extra-functional properties. These could be interfaces that capture timing behaviour of the component, or interfaces that tell what behaviour can be expected from a component if the assumptions on its environment are invalidated by external (unintended) faults. The latter can be used in a similar manner that FTA/FMEA analysis is carried out on assemblies of hardware components. How the interfaces are derived and how they are used in an upgrade process is an interesting field of study, but it seems that there will be potential use for both paradigms, correct-by-construction using model-based development, and constructing-the-correct where the source code of a component is assured to satisfy its contracted interfaces by its team of developers. In either case, the remaining step would be to assure that given certain interfaces the assembly of (upgraded) components satisfy its system level properties, e.g. safety-related requirements. This requires proof techniques that build up incremental proofs based on earlier analyses and thus achieve the overall assurance in an effcient manner.

None of the tools and environments for model-based development are mature enough to satisfy the above needs. We use the details of three case studies to illustrate this gap and hopefully provide a benchmark for future studies. Two of the case studies concentrate on the needs related to the automatic generation of code. Since there is more maturity on the tool front in those respects, we will report on these in less detail. The third case study, a sanitised example of an unmanned vehicle that was provided by Saab Aerospace, serves to illustrate the need for tracing intents to formally verified component code. 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.

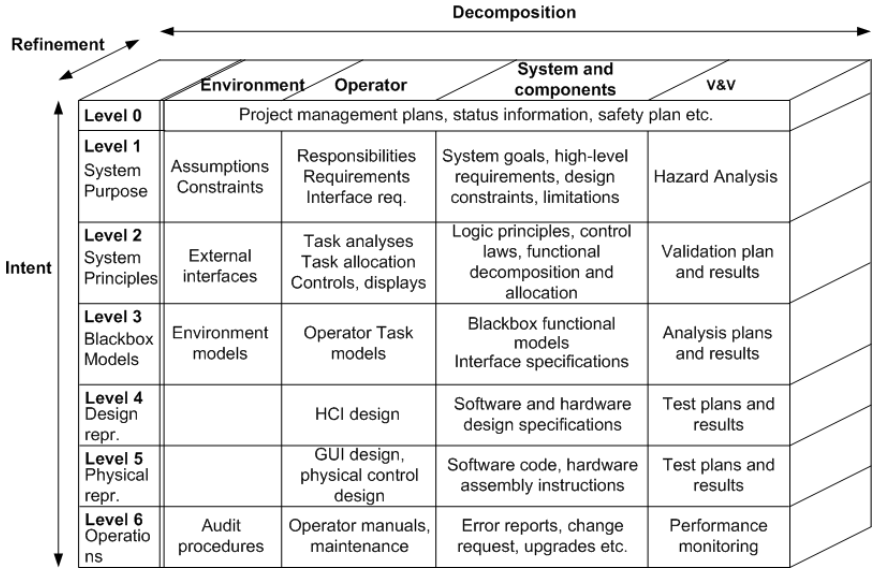

Fig. 1. The structure of an Intent Specification [Cor03a]

3 Intent Specifications

Intent Specifications [Lev00] 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 Fig. 1). 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 [GHA02].

- 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.

In the unmanned vehicle case study below, the tool used for implementing an intent specification was SpecTRM (Specification Toolkit and Requirements Methodology), a commercial tool from Safeware Engineering [Cor03a]. It is a document-oriented tool that focuses on system requirements and specification. The tool 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 [HL96]. The primary goal of the language were readability and reviewability, completeness with respect to safety, and assisting with system safety analysis of the requirements [LW04].

SpecTRM provides simulation and some static analysis of the SpecTRM-RL models and also a limited support for traceability. By executing the models and simulating the system, the engineer can study the behavior of the system before the actual implementation. The formal analysis tools availiable are robustness and determinism analysis. By analysing robustness it is meant that SpecTRM checks if the modelled system has a specified response for every sequence of inputs to the system. By analysing determinism it is meant that the tool checks if several behaviors are specified for the same sequence of inputs. However, both of these analysis tools are conservative, thus generating false alarms as possible examples of nondeterminism and nonrobustness.

4 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 [LH94] 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 FTA and FMEA mentioned in Section 2.1, and component-level and system level verification have 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.

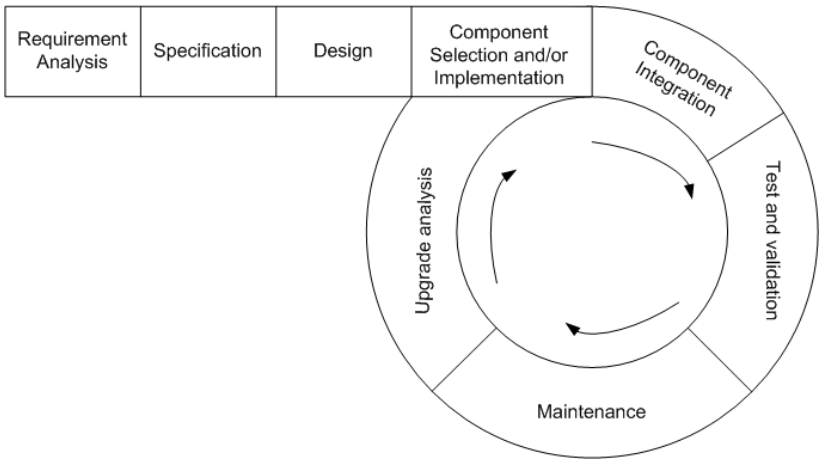

Fig. 2. The Sigma development process model

Elmqvist [Elm03] presents the Sigma development metaphor as a model for system level upgrades based on component updates (see Fig. 2).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.

4.1 Example: Unmanned Vehicle

The unmanned vehicle is controlled by the Remote Vehicle Control Unit (RVCU). The vehicle operates inside a closed area (see Fig. 3) 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.

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 [FBT97] 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.

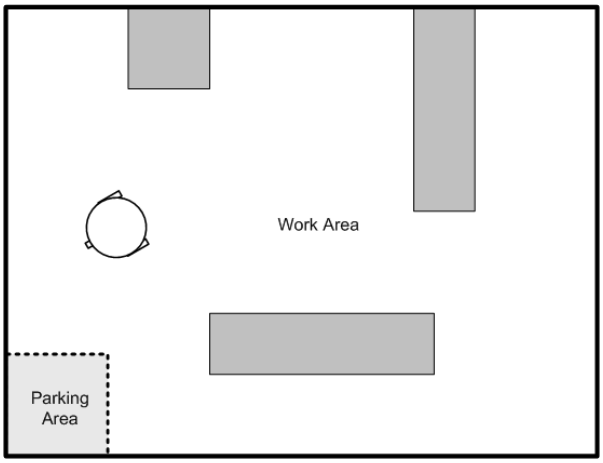

Fig. 3. A possible environment with obstacles for the unmanned vehicle

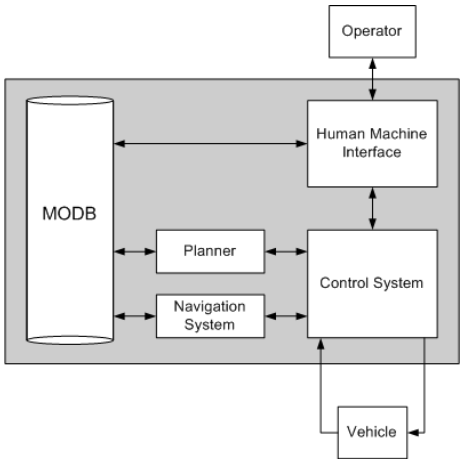

Fig. 4. The RVCU architecture

The five components of the RVCU are presented in the architectural model in Fig. 4:

- 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.

5 From Specification to Design

The third level of intent specifications described in Section 3 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 [Tec03, BG92]. 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 [SS98]. 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 [HNT03]. 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 (version 1.0.14) 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 as hypertext links. A more sophisticated support for traces would be appropriate in order to make the tool suitable for industrial use.

One could question what is the use of an intent specification tool when there are already more mature requirements engineering tools (such as CORE [Cor03b] and DOORS [Tel03]) available on the market. The reason we thought a tool like SpecTRM was interesting in the context of the case study was that the tool was supposed to support the hierarchy of models described in Figure 1 that is well-suited for the safety-critical systems development. Also, earlier experience with the SpecTRM-RL language had proven to be positive in terms of communication with non-experts and thereby ease of validation of the early requirements. In contrast to the structure of the above level 3 models, that are quite understandable to engineers from different disciplines, the software engineering oriented tools like DOORS that build upon the object-oriented notation have so far been less accessible to other engineers. However, the SpecTRM tool was not able to provide convincing support to the involved engineers to justify its use in the current status.

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 2003 version of the Prover plug-in was insufficient for our purposes.

We conclude that a tool environment that aids the developer beginning with a specification language such as SpecTRM-RL down to verified code is needed but is not fully available today. The unmanned vehicle example was

indeed tried on a tool chain that combines SpecTRM and Esterel Studio, but with lack of automatic translation between the tools, any of the tools could be replaced by another alternative. A case study by Leveson and Weiss also address the use of SpecTRM and intent specifications in model-based development [LW04]. However, their focus is on reuse at the software behavioral specification level and they do not address the issue of generating and verifying code.

A positive aspect of the SpecTRM implementation is its use of the Java-based Eclipse [Ecl03] environment that allows plug-in translators to be added conveniently. A trial plug-in, 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 [ILo04] and Visual State from IAR systems [IAR04]. 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 the style 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 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 [Gra04].

6.3 Airbag software

Rhapsody was also tested as a candidate for design and code generation of 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 [Tec03] was studied for code generation in this part of the application. Another useful feature of Scade was the formal verification support with the Prover plug-in that was tested to a limited extent on the crash algorithm model. Scade is a tool that is based on the language Lustre [HCRP91] with a formal semantics, and has a history of usage in European aerospace applications (the code is generated by a DO-178B certified compiler that makes the tool an appropriate candidate for safety-critical applications).

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 bag systems. The low cost constraints of the ROM violate 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 this 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 fairly 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. None of the tools however, have a component-based support. This is perhaps not expected from tools like Esterel and Scade that have a state-based or data-flow oriented style, close to the environment that other engineering disciplines are used to (state machine or function blocks). But the enhancement from object-oriented tools like Rapshody to component-based modelling is needed if compositional analysis of upgrades is to be supported.

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. UML diagrams syntactically host both styles of modelling, but the semantic gaps still need to be addressed in tools that aim to support development of high-assurance 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 component-based high-assurance systems.

7 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 NFFP. 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 plug-in translator was developed by Erik Sundvall. The state of practice and the problem definition from the aerospace sector has benefited from numerous discussions with Lars Holmlund, Rikard Johansson, and Jan-Erik Ericsson from Saab AB, whose cooperation is gratefully acknowledged.

References

[ALR01] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical report, University of Newcastle, 2001.

[AT05] J. Hansson A. Tesanovic, S. Nadjm-Tehrani. *Embedded System Development with Components*, chapter Modular verification of reconfigurable components. Springer-Verlag, 2005.

[BCC+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min'e, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation: complexity, analysis, transformation*, pages 85–108. Springer-Verlag New York, Inc., 2002.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[CCG+04] S. Chaki, E. Clarke, A. Groce, S. Jha, and H.Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[CCO02] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Flavers: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–, 2002.

[Cor03a] Safeware Engineering Corporation. *SpecTRM User Manual*, 2003.

[Cor03b] Vitech Corporation. http://www.vtcorp.com/. URL, December 2003.

[CSSW04] I. Crnkovic, J.A. Stafford, H.W. Schmidt, and K. Wallnau, editors. *Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE)*, Edinburgh, UK, May 2004. Springer Verlag.

[Ecl03] Eclipse.org. http://www.eclipse.org. URL, October 2003.

[Elm03] J. Elmqvist. Analysis of intent specification and system upgrade traceability. Master's thesis, Linköpings University, December 2003. LiTH-IDA-EX–03/074–SE.

[Eri04] A. Eriksson. Model-based development of an airbag software. Master's thesis, Linköpings University, March 2004. LiTH-IDA-EX–04/025–SE.

[FBT97] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):25–40, March 1997.

[GHA02] L. Grady, J. Howard, and P. Anderson, editors. *Safety-Critical Requirements Specfication and Analysis using SpecTRM*. US Software Safety Working Group, February 2002.

[GKR05] L. Grunske, B. Kaiser, and R. H. Ruessner. *Embedded system development with components*, chapter Specification and evaluation of safety properties in a component-based software engineering process. 2005. To Appear.

[Gra04] A. Grahn. Code generation from high-level models of reactive and security-intrinsic systems. Master's thesis, Linköpings University, April 2004. LiTH-IDA-EX–04/030–SE.

[Hat97] Les Hatton. Reexamining the fault density-component size connection. *IEEE Softw.*, 14(2):89–97, 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HL96] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *Software Engineering*, 22(6):363–377, 1996.

[HMW01] D. Hamlet, D. Mason, and D. Woit. Theory of software reliability based on components. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 361–370. IEEE Computer Society, 2001.

[HNT03] J. Hammarberg and S. Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with esterel. In *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. Elsevier, June 2003.

[HNT04] J. Hammarberg and S. Nadjm-Tehrani. Formal verification of fault tolerance in safety-critical configurable modules. *International Journal of Software Tools for Technology Transfer (STTT)*, December 2004. Springer Verlag.

[HV02] K. Havelund and W. Visser. Program model checking as a new trend. *International Journal of Software Tools for Technology Transfer (STTT)*, 4(1):8–20, 2002.

[IAR04] IAR. http://www.iar.com/. URL, March 2004.

[ILo04] ILogix. http://www.ilogix.com/. URL, March 2004.

[Jon95] C. Jones. Patterns of large software systems: Failure and success. *Computer*, 28(3):86–87, 1995.

[Lev00] N. Leveson. Intent specifications: an approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35, January 2000.

[LH94] J. H. Lala and R. E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, January 1994.

[LW04] N. G. Leveson and K. A. Weiss. Making embedded software reuse practical and safe. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 171–178. ACM Press, 2004.

[Sch03] H. Schmidt. Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65(3):215–225, 2003.

[SS98] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998*, volume 1522, pages 82–99. Springer-Verlag, Berlin, 1998.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition edition, 2002.

[Tec03] Esterel Technologies. http://www.esterel-technologies.com/. URL, October 2003.

[Tel03] Telelogic. http://www.telelogic.com. URL, December 2003.

[TNHN04] A. Tesanovic, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of embedded computing*, 2004. To appear.

[VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.