FTOS: Model-Driven Development of Fault-Tolerant Automation Systems

Christian Buckl
fortiss GmbH
Cyber-Physical Systems
80805 München, Germany
buckl@fortiss.org

Dominik Sojer, Alois Knoll
Technische Universität München
Department of Informatics
85748 Garching bei München, Germany
{sojer,knoll}@in.tum.de

***Abstract*—The design of fault-tolerant automation systems is a complex task. These systems must not only satisfy real-time requirements but they must also deliver the specified functionality in the presence of both software and hardware faults. To achieve fault-tolerance, systems have to use redundancy. This redundancy is usually achieved by replicating hardware units and executing the application within a distributed system.**

Model-based design tools promise to reduce the complexity of the design process by raising the abstraction level. However, most of the existing tools focus only on functional aspects. Code realizing extra-functional requirements such as fault-tolerance mechanisms, communication, and scheduling is not targeted. However, this type of code makes up the majority of the code of a fault-tolerant real-time system. This paper presents FTOS, a model-based development tool for the design of fault-tolerant automation systems that focuses on code generation of extra-functional requirements and therefore complements existing tools.

I. Introduction

During the last years, the software share in automation systems has significantly increased. The safety of these systems was then only dependent on the mechanical and electrical system components. However, due to the rise of software in this area, nowadays it has also become a safety critical system component. However, software systems in general tend to be more complex then mechanical or electrical systems, because they do not obey to physical laws and therefore *anything* can happen in software and to software. This progression lead to a huge increase in system complexity that has to be dealt with at system design and system reengineering, which is even more frequent in automation engineering than system engineering.
Model-driven development might help to handle this complexity by abstracting from the actual implementation layer to model and metamodel layers. From these layers, huge parts of the system can automatically be generated. Model-driven development originated in software development [22] but it can very easily be extended to systems development and therefore be also used in automation engineering, due to the abstract way of modeling that it provides.
This paper presents FTOS [5], a model-driven development tool that generates software architectures for fault-tolerant automation systems by taking not only functional system aspects into account in its automatic generation process, but also extra-functional system aspects like faults and failures. Section II will reveal the characteristics of the automation domain. The requirements on a model-driven development tool for safety-critical real-time systems in discussed in Section III. Section IV presents FTOS in detail and shows how FTOS can handle the specific requirements of the automation domain. Finally, section V will discuss our future work to further increase the value of the tool.

II. Automation

Automation engineering is a unique engineering discipline because it unites a broad range of scientific subjects and various system views. In the publicly founded research project "Software Platform Embedded Systems 2020" (SPES2020[1]), we analyzed this domain and created a list of its demands on model-driven software development [26].
Automation engineering combines computer science, mechanical engineering, electrical engineering and process technology, which leads regularly to misconceptions and information loss, because each of these disciplines has its own basic principles and assumptions. An electrical engineer will, for example, see a control cabinet as a set of connected control units with fixed physical borders, whereas a computer scientist will think of it as an abstract distributed system whose physical dimension is of no interest.
Another important challenge in the automation domain is the large variety of considered systems: a single programmable logic controller (PLC) is as well an automation system as a whole rolling mill. Automation engineers have to deal with this huge hierarchy of complexity from simple, small systems up to very complex systems of systems, but obviously the development processes at these different layers of abstraction differ tremendously.
Model-driven software development may be of great value for dealing with these domain specific challenges, however solutions have to be found for a set of specific problems that were identified during SPES2020. On the one hand, there are project organizational challenges that have to be solved. Currently, model-driven software development is very strongly

[1] http://spes2020.informatik.tu-muenchen.de/

tied to agile development methodologies, because not only the final product evolves over time, but also the tools [27]. Moreover, the versioning of models is still under research and the use of text-based tools like Subversion [23] cannot meet this requirement, because the semantics of two models may be equivalent even if their syntax differs. Moreover, current modeling techniques like XMI [2] allow references to other files by the line, which can lead to problems that cannot be handled by text-based versioning tools.

Apart from the organizational flaws, model-driven software development lacks some technical features on the other hand. One of the major missing features is the possibility to handle extra-functional system features like timing and fault-tolerance. Automation systems are prime examples for cyber-physical systems, as they interact with the physical world, in which these extra-functional aspects play a very important role. The importance of time is very obvious, but fault-tolerance is also becoming more and more important with the increasing need to certify automation systems to safety standards like IEC 61508 [1].

The second major technical challenge for model-driven software development of automation systems is closely connected to the large variety of systems and disciplines, which constitute the automation engineering domain: model-driven software development has to be able to handle different views on a modeled system, for example software, hardware and mechanical views. This handling has to go beyond just "showing" the system, but there have also to be mechanism to trace the impact of modifications in one view to the other views.

This paper deals with the technical challenges identified in this section. Based on these general requirements one can derive six requirements to tools for model-driven software development, as identified in section III.

III. Requirements on the Tool

This section identifies and discusses the main requirements on tools used for the model-based development of fault-tolerant real-time systems. As already stated in the introduction of this chapter, various tools are available for the development of embedded systems. Several of them are discussed in [5]. However, these tools focus mostly on the application functionality. Code realizing extra-functional aspects in a fault-tolerant, non-monolithic real-time system has to be implemented manually. This code is necessary to realize fault-tolerance mechanisms, communication within the distributed system, I/O operations, scheduling, and process management. The main reason, why the generation of such code is not covered by existing tools, is the platform dependency of these mechanisms. The realization depends on the used operating system and hardware and cannot be implemented using platform independent programming languages like ANSI-C. Due to the great heterogeneity of used hardware and operating systems [24], [19], it is not possible to implement a code generator that supports a priori all possible combinations. This leads to the first requirement:

Requirement 1: The code generator must be expandable (even for the application developer) to support additional platforms and arbitrary programming languages.

Providing simple means to expand the code generation is the first step to get a useful generation tool for fault-tolerant real-time systems. However, an easy expansion of the code generation alone is not sufficient, since the initial modeling language can not cover all possible mechanisms that one might want to realize utilizing the development tool. For instance, the tool supports the most important fault-tolerance mechanisms such as active and passive replication or rollback recovery. However, there are of course many other mechanisms that might be suitable as well. To support a new mechanism, the modeling language must be expanded to allow the specification of the required information. The same is true for the automatic generation of I/O operations. If in different projects the same device is used repeatedly, it might be reasonable to add generation functionality to the code generator to support this device. In case the class of the device is not supported in the current version, one might need to add a new device class with certain parameters in the modeling language. Therefore, the second requirement that must be satisfied by the tool is:

Requirement 2: The modeling language must be easily expandable.

It is important that expansions of the modeling language or the code generator must not affect the existing parts of the code generator. By allowing the expansion of meta-model and code generation functionality, the code generator can be adjusted to the requirements of the company or developer group using the tool. Such groups comprise typically different stakeholders, such as real-time system experts, hardware experts, safety manager and domain experts. This fact can be exploited similar to the approach in component-based approaches [28], [4]. The responsibility for different aspects of the code generation functionality can be assigned to dedicated experts. To support this approach, the third requirement must be satisfied:

Requirement 3: The code generation functionality must be separated into modules to allow an independent implementation of solutions for different aspects.

In addition, the multitude of different experts causes problems. Having different backgrounds and using different approaches, the interaction and communication between different experts plays an important role. Models are a natural connection factor. To ease the communication process, the used models must be simple, intuitive, and unambiguous. This requirement is even more strengthened by the fact that the models are used for extensive code generation. Many modeling languages such as UML [21] lack the precision and rigor needed for extensive code generation [17]. This leads to the next requirement:

Requirement 4: The specification/modeling language must have explicit and unambiguous (execution) semantics.

Since this requirement is very extensive, this issue is discussed separately in [5]. Explicit and unambiguous models reduce the probability of design errors. However, the tool must support means to prove the correctness of the system. These proofs

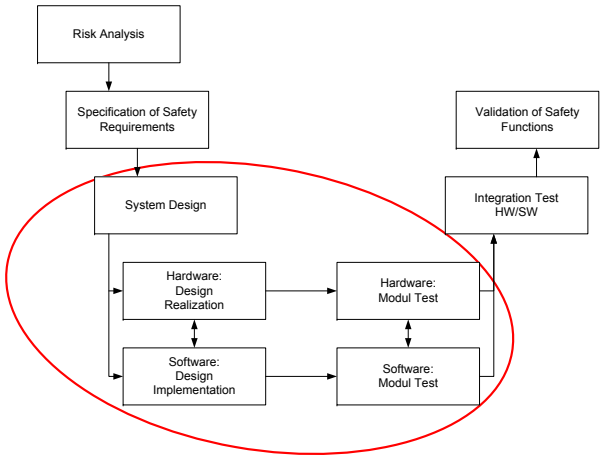

Fig. 1. FTOS: Relevant Development Phases

must be integrated in two dimensions: model validation and code verification. For the first issue, extensive tests must be employed:

Requirement 5: Tests have to be integrated in the tool to check the validity of the model.

In addition, there must be high assurance in the generated code. In fault-tolerant system, especially the correct implementation of components taking into account the fault hypothesis is important. The same functionality might be implemented in different ways, if other assumptions on faults are used. Therefore, it is necessary to verify the generated code:

Requirement 6: The tool should include formal methods to ensure a high assurance in the generated code.

IV. FTOS

This chapter presents a holistic overview of the approach and gives an introduction into basic concepts of FTOS. FTOS is used to model the system and to generate the code related to extra-functional aspects. The phases of a standard development process that are supported by FTOS are depicted in Figure 1. The designer is supported during system design by providing a specification language. FTOS validates the models and generates a tailored run-time system that provides the functionality for fault-tolerance mechanisms, communication within the distributed system, scheduling, and I/O operations. The components realizing the application functionality have to be implemented by the developer. A number of tools are available that can support the developer in this task.

This chapter is intended to provide an overview of FTOS and to identify the development steps. It starts by identifying the requirements that have to be satisfied by FTOS to achieve the intended goals. Based on these requirements, the basic concepts of FTOS are discussed and the different development steps are presented. At the end of the chapter, two applications are presented that are used to illustrate the introduced concepts.

A. Template-Based Code Generation and Development Steps

This section enumerates the concepts used to satisfy the requirements stated in the previous section. Furthermore, the resulting tool is presented and the different components are explained.

The main requirement is an easy expandability of the code generator with respect to generation functionality and modeling language. The first requirement can be solved by leveraging template-based code generation [25], [11], as it was pointed out in the context of the predecessor of FTOS called Zerberus [9]. The concept of a template-based code generation is depicted in Figure 2. Instead of having one monolithic code generation kernel that encapsulates all the code generation functionality, template-based code generators consist of a code generation core and templates encapsulating the generation functionality. A template can realize a certain aspect of the fault-tolerant real-time system or can be used to combine further templates to finally form a complete run-time system. Thus, the input of the code generator consists not only of the model, but also of a set of templates. The task of the code generator is to analyze the model, select a suitable set of templates and to adapt these templates to application requirements.

The advantages of this approach are obvious: new templates can be added easily. These templates can realize new aspects of the system, e.g. support new hardware or a new fault-tolerance mechanism, but can also be used to generate code in a different target language. It is even possible to generate natural language to provide necessary documentation. Another advantage of this approach is that the complexity of the code generator can be reduced significantly. This is in particular very important when using the code generator for safety-critical system development. Very often, code generators are by far more complex than the generated programs. Thus, the certification of a code generator becomes usually too expensive. In addition, any changes of the code generator to expand the code generation functionality lead to the necessity of a new certification of

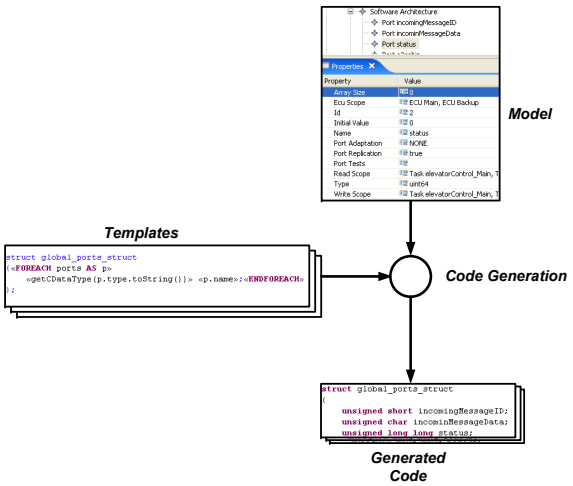

Fig. 2. Principle of Template Based Code Generation

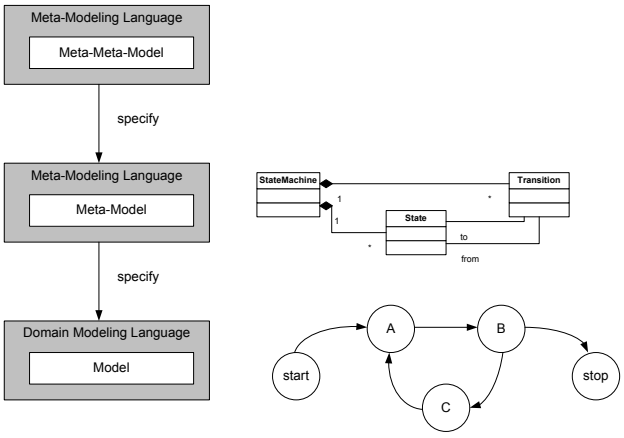

Fig. 3. Meta-Modeling Framework Approach [16]

the whole code generator. In contrast, template-based code generation solves this problem. The complexity of the code generation core can be reduced to a minimum. Furthermore, the addition of new templates only leads to a new evaluation of the affected templates, while other templates can be used without a repeated certification.

In addition, the concept of templates fulfills inherently the request for modularity. System engineers can define a generic software architecture and specify the interfaces between the different components. Experts can use their expertise to realize templates solving specific aspects of the complete system. Safety engineers can identify the important components of the system and add appropriate mechanisms to guarantee safety and reliability.

However, the concept of templates does not address the expandability concerning the modeling language. A solution is the use of a meta-code generation framework. Several of these frameworks are available such as openArchitectureWare [14], AndroMDA[2] or MetaEdit[3]. These code generators allow the definition of modeling languages in the form of user-defined meta-models. The concept of these frameworks is depicted in Figure 3. Based on a meta-modeling language, the developer of the code generator can define a meta-model. In the example of the figure, a meta-model for finite state machines is described. The meta-modeling language is typically based on the class diagram notation and allows the definition of classes, references, attributes, and data types. Based on the meta-model, the application developer can define a concrete model. The support of object oriented concepts such as inheritance [15] and polymorphism [12] is a key factor to simplify the expansion of the modeling language / meta-model. By introducing new sub classes e.g. for a specific device class, the modeling language can be expanded straightforward. In addition, the concept of polymorphism allows this expansion by specifying a code generation function for that specific sub class. Other code generation functions can be left unchanged.

[2]http://www.andromda.org/

[3]http://www.metacase.com/

Instead of augmenting the code generator Zerberus to support meta-modeling, FTOS is based on openArchitectureWare[4] (oAW) [29]. The meta-models used in FTOS are described in detail in [5].

The final two requirements are satisfied by incorporating model validation rules and formal verification. The validation rules are directly included in the code generation process. The verification is more complex. Since FTOS does not focus on a specific target language like Whalen et al. [30], it is not possible to formalize the translation between modeling and target language. Furthermore, the verification must also be suited to support the expandability of the code generator. This dilemma is solved by specifying the formal behavior of a template. Based on this formal description, the formal model of the developed system can be generated in parallel to the original code generation process. Important properties can be verified by integrating formal verification tools. A detailed description of this approach can be found in [5].

The complete tool chain is depicted in Figure 4. Based on meta-models realizing the domain specific language, the developer team can specify the concrete system models. FTOS uses four meta-models to describe the different aspects of fault-tolerant systems. The modeling tool incorporated within oAW allows the specification of the models using graphical notations. Therefore, syntactical errors are excluded by design. Nevertheless, it is necessary to check the semantic correctness of the models. This model validation is realized in the next step. The validated models are then combined to one model. In addition to the mere combination, supplementary information is computed in this step to simplify the code generation. Some validation rules test the interaction between different models. These rules are checked after the model transformation. Finally, the validated, expanded, and combined model is used for code generation. The code generator selects appropriate templates to solve application aspects and adopts these templates to application requirements. The result is a tailored run-time system including mechanisms for scheduling, inter-process communication, fault-tolerance mechanisms, and synchronization. Furthermore, user implemented code that realizes the functionality for the application, such as a controller function, is embedded into the generated code. In parallel to this code generation, a formal model of the system is generated that can be used for formal verification. The individual steps are explained in more detail in the following section.

B. Code Generation Process

1) Modeling: The modeling activities in oAW are based on the Eclipse Modeling Framework (EMF) [10]. This modeling framework is used both for the definition of the meta-models and for the definition of the concrete models. The previous section already indicated that it is useful to split up the models into sub-models to describe the different aspects separately. This technique increases the simplicity of the models and

[4]http://www.openarchitectureware.org/

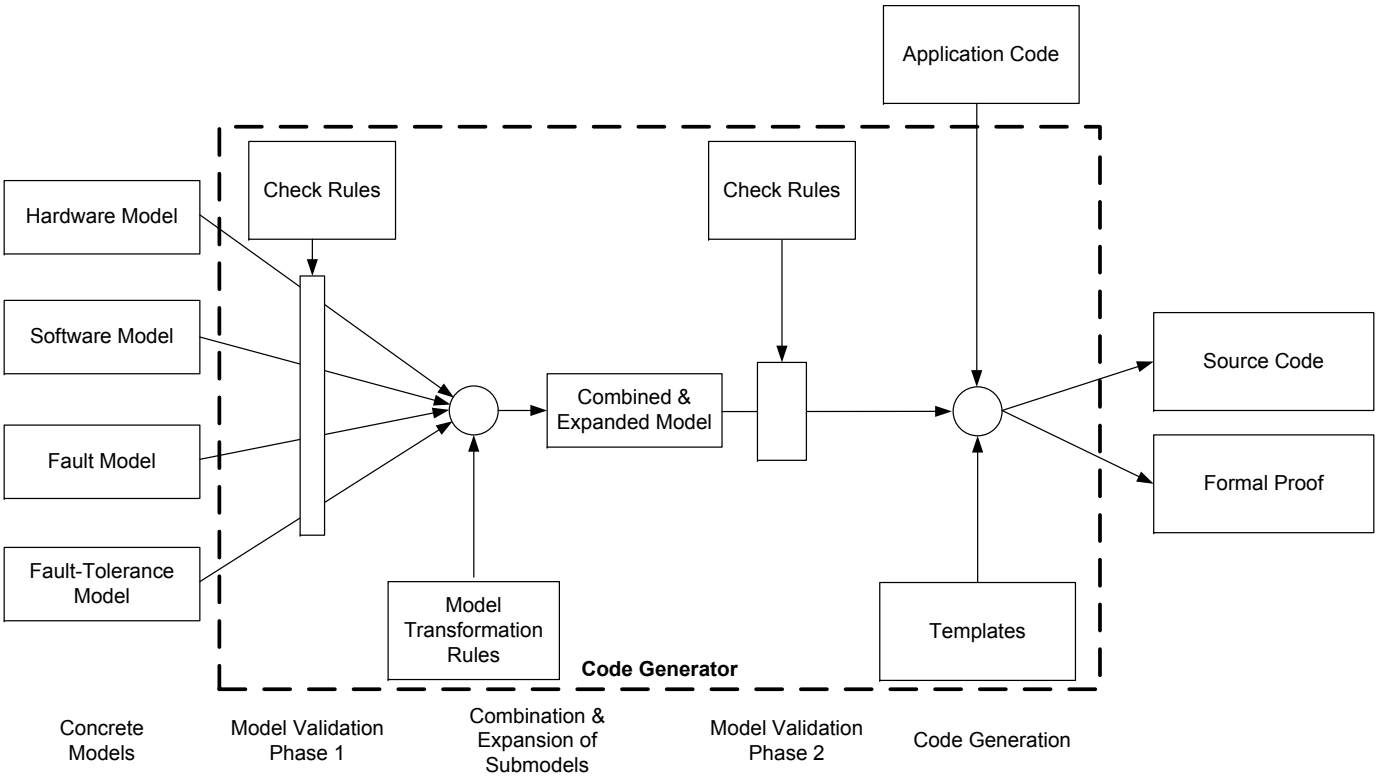

Fig. 4. Code Generation Process

the separation of concerns. A detailed description of the used meta-models and their relation can be found in [5].

2) Validation: Semantic design errors have to be detected in early design phases. Therefore, several tests are formulated to check the validity of the specified models. The specification of these tests is supported in oAW by offering the validation language **CHECK**, an equivalent to the object constraint language (OCL) [20] available for UML. Tests in CHECK are specified as formulas in First-Order Logic. One example for a test is depicted in Figure 5. This test checks whether each communication point (port) is read by at least one software component (actor). Other examples are tests to check the reachability of all application modes or to ensure the absence of constructs in the model that could introduce non-determinism.

3) Model-To-Model Transformation: The next step after the model validation is the combination of the different models. The resulting model is used for the code generation. To simplify the code generation, further information, which is already implicitly contained in the models, can be computed and added explicitly. One classical example is the handling of references: if the model contains an unidirectional reference, it might be useful to also add a reverse reference. This approach decreases on the one hand the error-proneness of the initial model, since directed references are in contrast to bidirectional references much easier to maintain for the developer. On the other hand, the code generator benefits from the bidirectional references. Model-to-Model (M2M)-Transformation is supported in oAW by offering the functional programming language **EXTEND**. In Figure 5, the result of the M2M transformation is depicted for a communication point. The M2M transformation computes the number of software components using the communication point and the number of relevant electronic control units. Further examples for the M2M transformation are discussed during the presentation of the used models in [5].

4) Code Generation: The code generation is based on templates, as described before. Templates represent the actual code generation ability and can be added easily. A template can be used to solve a certain aspect or to combine the results of different templates to form a run-time system. Most templates are platform dependent in the sense that they offer a solution only for a certain combination of hardware, operating system, and programming language. Therefore, also the correct selection of adequate templates is necessary.

oAW uses for the implementation of templates the `XPand` language. This language is very simplistic. It offers the statements `DEFINE` to declare a new code generation function and `EXPAND` to call other generation functions during the code generation. An important feature of oAW is the support of polymorphism to guarantee an easy expansion of the code generation ability. To specify the control flow of the code generation, the commands `FOR`/`FOREACH` and `IF`/`ELSE` can be used. The `FOREACH` statement is used to generate code for each object of a certain type that is declared within the model. Finally, the commands `FILE` and `ENDFILE` allow the

management of the generated files.

The code generation technique itself is simple: the adaptation of the templates to the model is performed using a technique similar to preprocessor macros. Text sequences between the different XPand commands are directly copied to the generated files and variables allow the access to objects and their attributes. A description of the code generation technique can be found in [8]. Figure 5 shows one example for a template that is used to generate code for the declared communication points.

5) Verification: Formal verification is in particular of great importance for fault-tolerant systems to verify the correct implementation of mechanisms with respect to the fault hypothesis. Two components realizing the same functionality may be implemented in different ways, if the fault assumptions differ. However, the fault assumption of the system that should be developed does not necessarily match the fault assumption used for the implementation of the available templates. It is therefore necessary to assure the interoperability of different components for a specific application context. The main idea to solve this issue is to use a formal description of the components behavior. As presented in [5], the meta-model provides means to describe the behavior of components in the presence of faults. Because the formal description must be specified by template developers that have typically no expertise in formal verification, it is necessary to limit the required knowledge. This goal is achieved by using BoogiePL [13] for the specification. BoogiePL is actually an intermediate language for program analysis and program verification and resembles imperative programming languages. Developers have to learn very few concepts in order to be able to implement a formal specification. Based on this specification, the code generation tool generates a formal model of the complete system. This formal model can be used to verify certain properties of the system by using a SMT (Satisfiability Modulo Theories) solver. More details on this approach are given in [5].

6) Code Generation Result: Usually, the generated files contain source code for an arbitrary programming language. But since oAW is not restricted to one specific output language, it is also possible to generate documents in natural language. This can be useful, if documents for certification issues or user-manuals are required. Currently, FTOS provides templates for the generation of executable run-time systems for two different platforms. These systems include code for the timely-correct execution of the application, for process management and scheduling, as well as communication (interprocess, interprocessor) functionality. In addition, the selected fault-tolerance mechanisms are realized by the run-time system. The actual code realizing the application functionality, like control functions, is not covered by the tool and has to be implemented by the developer. An overview of the generated code can be found in [5]. A concrete example for the whole process is depicted in Figure 5.

C. Standard Compliant Fault Modeling

As presented before, FTOS consists of different meta-models for modeling different system aspects. One of them will be presented in more detail in this section.
In industry-driven environments like automation engineering, academic safety analysis techniques like model-checking are most of the time not feasible because they are usually very expensive. In this environment, *safety* is established by proving compliance with a generally accepted safety standard, like IEC 61508 [1]. These standards do not require the absence of any risk in a system but they only require that the risk should be as low as reasonably possible. This leaves the decision, which concrete safety measures should be applied, to the system developers.
FTOS also supports this paradigm by giving developers the possibility to model system faults that comply to the system faults specified by IEC 61508. During the code generation process, the set of modeled faults is automatically evaluated and techniques are chosen that are able to detect the occurrence of all modeled faults. When one of these techniques discovers a fault at runtime it is able to change the system's fault configuration, which triggers the fault handling mechanisms of FTOS.

V. Conclusion and Outlook

This paper presented FTOS, a model-driven development tool for fault-tolerant real-time systems, which meets the general requirements of the automation domain towards model-based software development. The tool can be easily expanded both at the front end (modeling language) and back end (code generation facility) to cope with the heterogeneity of automation systems. Formal methods can be included to prove the correctness of the code.

However, some aspects are still left for improvement. One of these aspects is the extension of FTOS by a modeling paradigm for safety requirements. Currently only static and generic safety requirements can be used, which might be two conservative. Typically, safety requirements are highly application dependent properties of system states that differentiate safe ones from unsafe ones. For example, some systems might tolerate transient errors up to a specific time span, whereas other systems might be turn unsafe even when very short transient errors occur. The modeling of these safety requirements would introduce more information into the generation process, which can be exploited to create a system that meets as many requirements as a traditionally developed system.

Apart from safety requirements, software design faults are also a problem that has to be dealt with in model-driven software development. Software design faults in general have been under research since the 1960ies [3], so probably no final solution to them exists. Model-driven software development reduces the probability that developers introduce such errors, but there are still possibilities for them. Therefore we will focus on the automatic generation of measures against the propagation of software design faults at runtime.

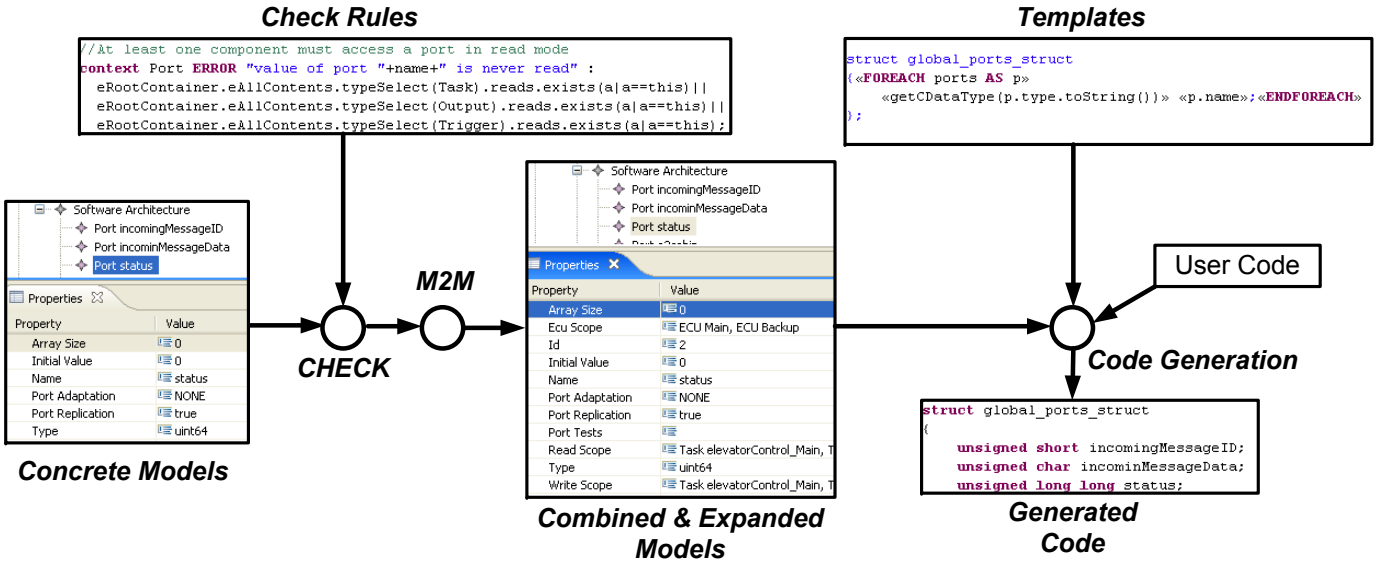

Fig. 5. Code Generation - A Concrete Example

A final improvement that we want to address in the future, will be the definition of a mathematical foundation for safety and fault tolerance that fits well with the safety approaches that are domiciled in industry and academia. Mathematical foundations for safety are quite common in academia [18] but their main problem is that they are hardly applicable by industry for various reasons. Therefore we will be looking for a mathematical description that can be mapped to academic safety approaches like model checking but that can also be mapped to the industrial safety environment, which is mainly driven by the need of standard compliance.

References

[1] Functional safety of electrical/electronic/programmable electronic safety-related systems.

[2] MOF 2.0 / XMI mapping specification.

[3] Tom Anderson, Peter Barrett, Dave Halliwell, and Michael Moulding. Software fault tolerance: An evaluation. *IEEE Transactions on Software Engineering*, 1985.

[4] Colin Atkinson, Christian Bunse, Christian Peper, and Hans-Gerhard Gross. *Component-Based Software Development for Embedded Systems - An Introduction*. Number 3778 in Lecture Notes in Computer Science. Springer, 2005.

[5] C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, TU München, 2008.

[6] Christian Buckl, Alois Knoll, and Gerhard Schrott. Development of Dependable Real-Time Systems with Zerberus. In *11th International Symposium, Pacific Rim Dependable Computing, PRDC 2005*, pages 404–408, Changsha, China, Dec 2005. IEEE.

[7] Christian Buckl, Alois Knoll, and Gerhard Schrott. The zerberus language: Describing the functional model of dependable real-time systems. In *Dependable Computing, Second Latin-American Symposium, LADC 2005*, Lecture Notes in Computer Science, pages 101–120, Salvador, Brazil, Oct 2005. Springer.

[8] Christian Buckl, Alois Knoll, and Gerhard Schrott. Model-based development of fault-tolerant embedded software. In *Second International Symposium on Leveraging Applications of Fomal Methods, Verification and Validation (IEEE-ISoLA)*, pages 113–120, Paphos, Cyprus, 2006.

[9] Christian Buckl, Alois Knoll, and Gerhard Schrott. Template-based development of fault-tolerant embedded systems. In *International Conference on Software Engineering Advances, ICSEA 2006*, Tahiti, French Polynesia, Oct 2006. IEEE.

[10] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Prentice Hall International, Oct 2003.

[11] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[12] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[13] Robert DeLine and K. Rustan M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, Mar 2005.

[14] Sven Efftinge, Markus Volter, Arno Haase, and Bernd Kolb. *openArchitectureWare*.

[15] A. Goldberg and D. Robson. *SmallTalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[16] Institute for Software Integrated Systems. Metagme -gme metamodeling environment. http://www.isis.vanderbilt.edu/projects/gme/meta.html, 2008.

[17] Ian Johnson, Colin F. Snook, Andy Edmunds, and Michael Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[18] Anjali Joshi, Steven P. Miller, Michael Whalen, and Mats P.E. Heimdahl. A proposal for model-based safety analysis. *Proceedings of the 24th Digital Avionics Systems Conference*, 2005.

[19] Edward A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.

[20] Object Management Group. *UML 2.0 OCL Specification*, Oct 2003.

[21] Object Management Group. *OMG Unified Modelling Language Specification*, 2.1.2 edition, Nov 2007.

[22] OMG. Model driven architecture, a technical perspective. Technical Report No. ab/2001-02-04, Object Management Group, 2001.

[23] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008.

[24] Shankar Sastry, Janos Sztipanovits, Ruzena Bajcsy, and Helen Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.

[25] Ajit Singh, Jonathan Schaeffer, and Mark Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distriuted Systems*, 2(1):52–67, Jan 1991.

[26] Dominik Sojer, Christian Buckl, and Alois Knoll. Stand und Anforderungen an eine Werkzeugunterstuetzung zur Entwicklung von Automatisierungssoftware. Technical Report TUM-I1003, Technische Universität München, 2010.

[27] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[28] Clemens Szyperski. *Component Software*. Addison-Wesley Professional, nov 2002.

[29] Markus Voelter, Christian Salzmann, and Michael Kircher. Model driven software development in the context of embedded component infrastructures. In *Component-Based Software Development for Embedded Systems*, number 3778 in Lecture Notes in Computer Science, pages 143–163. Springer, 2005.

[30] Michael W. Whalen and Mats Per Erik Heimdahl. On the requirements of high-integrity code generation. In *Proceedings of the Fourth High Assurance in Systems Engineering Workshop*, pages 217–224, Nov 1999.