Integration of Component Fault Trees into the UML

Rasmus Adler[1], Dominik Domis[2], Kai Höfig[2], Sören Kemmann[1], Thomas Kuhn[1], Jean-Pascal Schwinn[3], and Mario Trapp[1]

[1] Fraunhofer IESE
{rasmus.adler|soeren.kemmann|thomas.kuhn|mario.trapp}@iese.fraunhofer.de,
WWW home page: http://www.iese.fraunhofer.de
[2] University of Kaiserslautern, Computer Science Department
{dominik.domis|kai.hoefig}@cs.uni-kl.de,
WWW home page: http://agse3.informatik.uni-kl.de
[3] Siemens AG, Corporate Research and Technologies
jean-pascal.schwinn@siemens.com

Abstract. Efficient safety analyses of complex software intensive embedded systems are still a challenging task. This article illustrates how model-driven development principles can be used in safety engineering to reduce cost and effort. To this end, the article shows how well accepted safety engineering approaches can be shifted to the level of model-driven development by integrating safety models into functional development models. Namely, we illustrate how UML profiles, model transformations, and techniques for multi language development can be used to seamlessly integrate component fault trees into the UML.

1 Introduction

Embedded systems are of crucial importance to our society. We recognize our dependence in the moments when these systems do not work or produce errors. Headlines in the newspaper about plane crashes or car accidents show how coupled the advantages and dangerous disadvantages of these systems are to each other. Therefore the development of embedded systems comes with a large responsibility. Particularly the development of safety critical systems therefore underlies a series of legislative and normative regulations making safety to one of the most important non-functional properties of embedded systems. One of the main requirements is a sophisticated safety analysis of the system. Particularly in the case of software-intensive embedded systems, their complexity is rapidly increasing and extended analysis techniques are required that scale to the increasing system complexity.

Model driven development is currently one of the key approaches to cope with increasing development complexity, in general. Applying similar concepts to safety engineering is a promising approach to extend the advantages of model driven development to safety engineering activities. First, it makes safety engineering as a standalone subtask of system development more efficient. Second,

and even more importantly, this is an essential step towards a holistic development approach closing the gap between functional development and safety engineering.

This paper illustrates how model driven design principles can be applied to safety analysis techniques in order to enable an efficient analysis of complex systems. In contrast to other approaches it is not the goal to extend existing development approaches with some additional safety properties. Instead, it is the goal to shift full-fledged, established, and well-accepted safety engineering approaches to the level of model-driven development and to seamlessly integrate them to the functional development. In this paper, we use fault trees as an established analysis approach and illustrate how they can be integrated to a model-driven design approach. As a starting point we use component fault trees, which extend standard fault trees with the concept of modularity. In order to shift this approach to the model driven development level, we have defined a domain specific modeling language and according analysis and transformation algorithms. Since safety and particularly certification bodies are usually very conservative, it is very important to enable the use of proven-in-use tools to perform the actual analyses. To this end, model transformations are used to generate exchange formats for standard fault tree analysis tools like FaultTree+ [9]. By this means, the flexibility and expressiveness of the DSML-based fault tree language can be combined with the acceptance of proven-in-use analysis tools.

To tap the full potential of this approach, the safety meta models are seamlessly integrated to the functional meta models in the second step. To this end the functional models have been extended using a view concept, i.e., it is possible to define a safety view for each component using DSML-based component fault trees. By this means, safety becomes an integral part of software and systems development, safety modeling tasks can (partially) be delegated to developers, consistency checks between safety and functional models can be automated, and safety models are automatically reused together with the owning component. And even most importantly, it is sufficient to define the safety models of the single components, the overall safety models can then be generated automatically using the component safety models and the functional architecture models. In most cases, the complexity consequently increases linearly with the number of components so that the approach scales to large systems. Regarding first applications in industry, the advantages of model-driven development can be successfully extended to safety engineering. The combination of modularization, reuse, and automation can tremendously increase the efficiency of safety analyses of large systems. The remainder of this paper is structured as follows. First, we present related work. Second, we introduce component fault trees. Third, we present a more formal definition of CFTs based on a generic meta-model for components. Based on, we explain our multi language integration for integrating CFTs into architectural design models. Finally, we summarize our approach and discuss its benefits for industrial usage.

2 Related Work

Many approaches exist that try to integrate safety analyses and design models in order to reduce effort by automatically transforming the integrated model into a classical safety analysis such as fault trees. These approaches can be divided into three classes: Semantic Enriching, Fault Injection, and Failure Logic Modeling.

Semantic Enriching annotates additional safety-relevant semantic information to design models such as the role of a class in a fault tolerance mechanism, the safety requirements of the annotated entity, or failure modes and their likelihood of occurrence. The assumption for approaches belonging to this category is that adding *semantic* information to entities of the system model is sufficient for the deduction of a safety analysis model and requires less effort, than modeling the safety behavior manually. In UML models stereotypes, tagged values, and constraints are used for this purpose, usually. In [7], such annotations are used to mainly identify redundancy mechanisms. They concentrate on recurring safety analysis model constructs and automatically construct parts of the safety analysis model at a high level. The annotations presented in [4] can be applied to all elements of the design model. The used annotations are generically defined to cover the whole design space and to allow a detailed deduction of the safety analysis model. This coverage and the high detail of the safety analysis model is in this approach achieved at the expense of a large number of different annotations.

The advantage of Semantic Enriching is that safety-relevant information is specified in the language of the design engineer or programmer and within the same model as the analyzed system. This approach supports thereby the consistency of the design and the safety analysis model and increases the communication between the system developer and the safety engineer. However, the main disadvantage of approaches belonging to this category is the need for a high degree of detail in the safety analysis model complicating the process of annotating the model entities. Additionally, the annotations are usually not using the widely used semantics of common safety analysis models such as fault trees and additional knowledge for using the semantics is needed. This makes them difficult to apply, they are time-consuming, error prone and unfamiliar for safety engineers and certification bodies.

Fault Injection requires a formal or an executable design model. Undesired behavior is modeled along with the system model and model checkers or simulations, for example, are used to find inconsistencies between the model and the safety requirements. This procedure allows a high degree of automation and a strong correctness of the results. Using model checking for fault injection was invented by Liggesmeyer in [14]. More examples, can be found in [1], [10] and [3]. The drawback of approaches belonging to this category is the limitation of the system model on formal models. Furthermore, approaches of this category do not solve the problem of finding appropriate failure modes or support humans to think about conceptual faults that are not in the model and were not specified before.

The approach presented in this paper, belongs to the third category: **Failure Logic Modeling** (FLM) modularly defines the failure propagation of modules of the system in parallel to the data-flow through the system similar to a standard safety analysis. The disadvantage of FLM is that the annotation is still a manual task and very similar to a manual safety analysis. However, in our opinion this is also its advantage against Semantic Enriching and Fault Injection. The used development model is not constrained to be executable, as in fault injection, and the advantages of semantic enrichment that come from the combination of the system model and safety analysis model can be achieved without the drawback of applying a complicated and unfamiliar set of annotations. In failure logic modeling, the safety analysis can be performed in the way the safety engineer is familiar with, but modularly and in parallel to the design model, which helps to handle complexity and increases the communication between safety and system engineering. Examples of Failure Logic Modeling can be found in [17], [8] an [16]. The most similar approach to this paper are the Hierarchically Performed Hazard Operation and Propagation Studies (HiP-HOPS), which annotate subsystems in Matlab/Simulink with propositional formulas, which are mathematically equivalent to fault trees [16]. The approach has already been integrated into EAST-ADL [2], which extends the UML/SysML. In contrast to the approaches as discussed before, we propose in this paper an approach that uses model-based concepts to integrate the fault tree model with UML models. In this way, safety becomes model-based analyzable in the model-based development with UML models. The safety analysis model is then an integral part of the design model and the advantages of model-based development are transferred to the safety analysis model.

3 Introduction to CFTs

Fault tree analysis is a deductive, top-down method that analyzes the causes of a hazardous failure of a complex system. Fault trees offer thus a breakdown with regard to the hierarchy of failure influences. The root of the tree is called TOP EVENT and the leaves are called BASIC EVENTS. The decomposition of the top event into the basic events is defined by the remaining nodes which are logical operators. However, this kind of hierarchical decomposition is not sufficient when dealing with complex technical systems. In the functional design, the complexity of these systems is typically handled by an appropriate component concept that allows the decomposition into manageable components. To facilitate this architectural decomposition also for the fault tree analysis, the concept of component fault trees (CFTs) has been proposed in [12].

CFTs feature a decomposition approach that is used in a similar way in modern software design notations: subcomponents appear as "black boxes" on the next-higher level and are connected to the environment via their interface. The output interface is given by so-called OUTPUT EVENTS and the input interface is given by so-called INPUT EVENTS. Further, a CFT comprises so-called BASIC EVENTS that represents internal faults. The CFT relates every output event to

its internally caused basic events and its relevant input events. This decomposition is modeled with logical operators as it is done in the decomposition of a top event to a set of basic events in a common fault tree. Similar to a component, a CFT can be instantiated several times in other CFTs. In their environment, these instantiations are called "subcomponent" or "CFT instances". These instances can be composed by connecting input and output events. In- and output events of a CFT appear as subcomponent events when they are instantiated on the next-higher hierarchy level. The CFT graph as a whole is defined by its edge connections and the mappings from subcomponents to the corresponding CFT model describing their "internals". The complete CFT can be evaluated qualitatively and quantitatively like a classical fault tree (see [11] for a formal specification of CFTs, [12] for a description of the BDD-based and compositional evaluation algorithm used for CFTs; UWG3, a tool for modelling and evaluation of CFTs, can be found at [6]).

In the following, we exemplify the modeling of CFTs with the running example of this article. It is a system for charging the battery of a car with an electric power train. This system *charging* has to fulfill the safety requirement that the car must not start-up when the male connector is plugged, since otherwise the charging cable may crack and electrocute somebody. For detecting the plugging of the male connector, the system comprises a hardware component *ProximitySensor*, a hardware component *VoltageSensor*, and a software component *PlugDetection*.

The hardware components send respectively a signal to the component *PlugDetection* that indicates whether the male connector is plugged or not. The component *ProximitySensor* outputs the boolean signal *proximity* that is supposed to be *true* when the connector is plugged and *false* when the connector is unplugged. The component *VoltageSensor* outputs a signal *voltage* that is supposed to be *true* when the plug connection is energized and *false* otherwise. If both signals are *false* then the component *PlugDetection* sets its Boolean output signal plugged to *true* else it sets signal plugged to *false*. In this way, the component minimizes the risk of violating the safety requirement due to an undetected plugged connector.

Figure 1 shows the CFT *PlugDetection* that refers to the detection of a plugged connector based on the signals *proximity* and *voltage*. The CFT comprises two OUTPUT EVENTS (top triangles), two input events (bottom triangles) and three basic events (circles). The event *plugged_omission* refers to the common failure mode OMISSION, i.e., an unexpected absence of a signal when it is required. The event *unplugged_commission* refers to the common failure mode COMMISSION, i.e., an unintended provision of a signal when not required. The input event *Proximity_FN* captures a false negative value of signal *proximity*, i.e., the value is *false* although a connector is plugged. Accordingly, the input event *voltage_FN* refers to a false negative value of signal *voltage*. A false negative value of both input signals leads to an undetected plugging of the connector. Hence, the CFT defines that the conjunction of the input events causes both output events. As the outputs events can also be caused by internal faults of

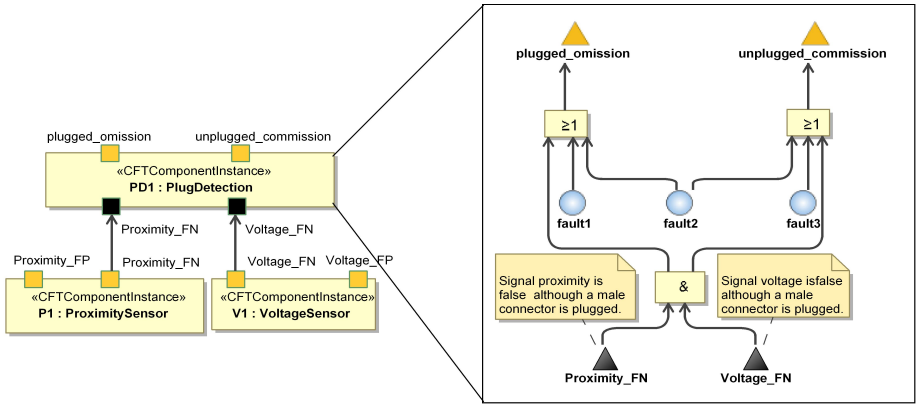

Fig. 1. Example for a component fault tree (CFT)

the component *PlugDetection*, the CFT comprises the basic events that refer to these internal faults. It has three basic events in order to distinguish between different kinds of faults. Basic event *fault_1* refers for the class of faults that causes only a omission of signal plugged. Basic event *fault_2* refers to faults that cause an omission and a commision of signal plugged. Accordingly, it is connected to both OR gates. Basic event *fault_3* refers to faults that cause an comission of signal plugged. Please note that the identifiers *fault_1 fault_2* and *fault_3* are anonymized due to NDA reasons. In practice the names should be expressive and semantically meaningful.

The CFT *ProximitySensor* that describes the failure behavior of the proximity sensor comprises an output event *Proximity_FN* that refers to a false negative value of the measured signal *proximity*. The CFT *VoltageSensor* that describes the failure behavior of the voltage sensor comprises an output event *Voltage_FN* that refers to a false negative value of the measured signal *voltage*. As illustrated in Figure 1, instances of the CFTs *ProximitySensor* and *PlugDetection* are composed by connecting output event *Proximity_FN* with input event *Proximity_FN* in order to define the overall CFT graph of the system. Instances of CFTs *VoltageSensor* and *PlugDetection* are composed by connecting output event *Voltage_FN* with input event *Voltage_FN*. This means that the fault trees belonging to output events *Proximity_FN* and *Voltage_FN* are adhered to the fault trees belonging to output events *plugged_omission* and *unplugged_comission*.

4 Towards model-based CFTs

The aforementioned component fault trees enable developers to formally specify the failure-propagation of their developed systems, and enable formal analysis techniques. CFTs are on the most abstract level hierarchical components with parts and connections between them. We therefore define CFTs based on a generic meta-model for component based software development that resembles

the most important principles of component based development CBD (cf. Figure 4). This meta model was build based on the UML modeling language.

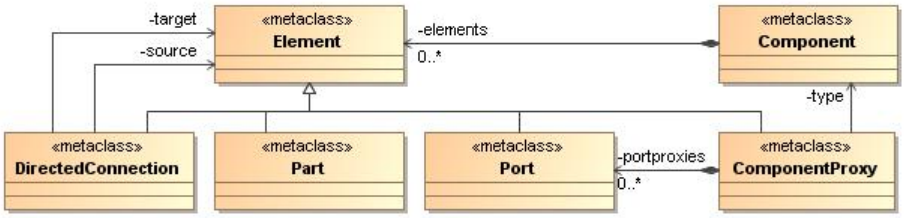

Fig. 2. Component Meta model

The meta model consists of COMPONENTS, which define component types, COMPONENT PROXIES, which represent component instances, PORTS, and DIRECTED CONNECTIONS. All are well known concepts in CBD. Components consist of ELEMENTS, which may be COMPONENT PROXIES, CONNECTIONS, or generic PARTS. Generic PARTS are abstract placeholders for more specialized language constructs therefore and need to be specialized by more concrete meta models.

CFTs are defined based on this abstract meta model (cf. Figure 3). This approach is helpful for language integration, which will be described in the next section. CFTs define a specialized type of components, CFTCOMPONENT, and specializes generic ports to INPUT- and OUTPUT EVENT PORTS . GATES and EVENTS are specializations of the abstract element PART. GATES combine events with each other. Two predefined gate types are common to all calculation backends, these are the predefined AND and OR gates. EVENTS represent faults, which are the core of fault trees. After identifying faults and their propagation logic, the next step is the definition of counter-measures to mitigate or tolerate the faults. Because classic (C)FTs do not support the modelling of measures, in practice the workaround is to model a failing measure as a basic event and relate it to the fault or the fault-subtree with an AND-GATE. The semantic of this would be that a fault propagates, whenever the fault occurs AND the counter-measure fails, too. The disadvantage is that one cannot distinguish original faults from failing counter-measures. Having a meta-modelled DSML it is straight forward to add a modelling element called MEASURE. This makes it not only possible to distinguish faults from measures graphically, but also to use the extended semantic information for additional analyses. One analysis could be to reason, whether every fault is covered by at least one measure. Figure 3 shows the complete meta model of Component Fault Trees, which is the abstract syntax of the CFT language.

Sometimes, for specific application domains, or for enabling more expressive fault trees, tool vendors provide proprietary gates as extensions to standard fault tree gates. These types are valuable, and therefore must not be omitted by modeling frontends. Additionally, it is necessary to integrate new domain specific

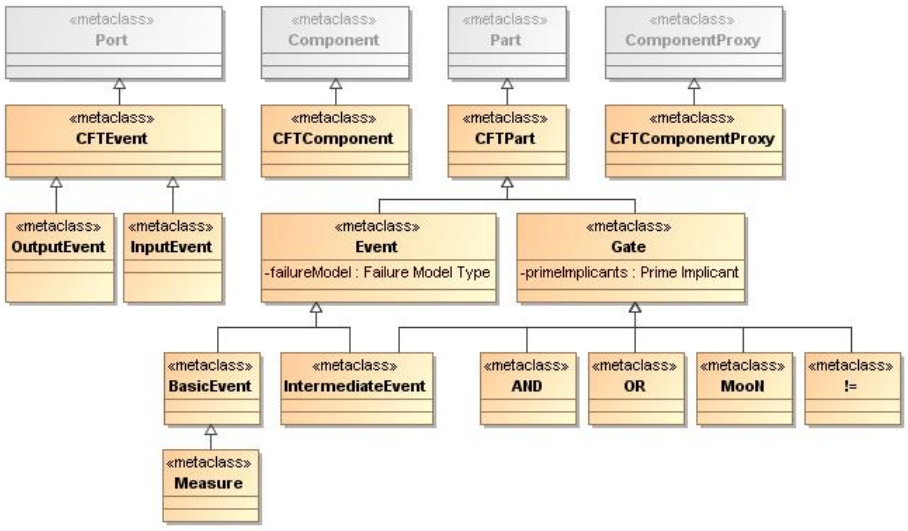

Fig. 3. CFT Meta model

languages with existing tool chains. For these reasons, the concrete syntax of CFTs is realized as a UML profile. This makes it possible to use UML modeling tools for developing component fault trees, and it enables us to inherit the UML profiling mechanism for vendor specific language adaptations. For this reason, the meta model has been converted to a UML profile (cf. Figure 4).

Our DSML approach for CFTs is based on the separation of modelling and calculation. Thus, the step of code-generation for functional DSML is replaced by a model transformation step from our model to a proprietary, tool specific format for analysing with an external calculation back-end. The challenge on the one hand and one huge advantage of our DSML approach on the other hand was the demand for supporting several back-ends with vendor specific language extensions. The need for rapid tailoring and adaption of our DSML arose with the realization that the analytical expressiveness of the different back-ends are not equal. Some gate-types such as m-out-of-n-gates cannot be directly transformed to the back-end language of other tools. Therefore we decided to apply the aforementioned profile based approach together with tailored transformation rules for each tool. By adding profiles to the regular CFT meta model, the language is tailored to a specific tool. The generic set of transformation rules for CFTs is specialized as well to yield tool specific transformations that transform to vendor specific target models. This way, a heterogeneous and non-standardized set of backend tools is made useable through one UML-based frontend, without disabling tool specific extensions.

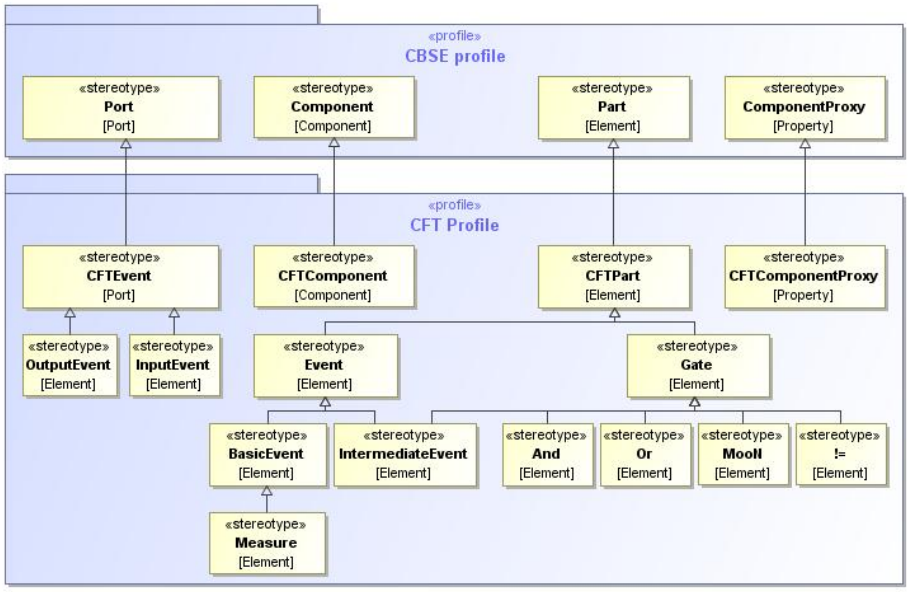

Fig. 4. CFT UML profile

5 UML profiling as DSML design

The visualization of the elements, names and their relations of the concrete syntax seems to be almost identical to the abstract syntax of the CFT language (cf. Figures 3 and 4). Nevertheless by defining the concrete syntax of the CFT language as UML profile, many benefits of model-driven development approaches are available to the specification of fault trees. However, this does not change the fact that the modeling language for describing component fault trees is still isolated. The first advantage is to create safety models in the same tools that are used for creating the architecture and design of the embedded system.

Moving from this promising step further to model-driven development, we need tightly integrated models that store all relevant information for a developed systen in one location. This enables developers to model relations between different system aspects and prevents redundant storage of information in several repositories. This way components and their ports for example need to be stored only once, in one repository. Languages that focus on specific aspects of the developed system - for example on CFTs and UML-based architecture definitions, can refer to this common location. This makes it possible to reuse modeling elements in several modeling languages, and keep them synchronized automatically. The common limitation when applying MDD approaches, the replication of information can be easily mitigated. As an example, the name of components are replicated in two modeling languages, the UML and the CFT DSML. This

replication of information leads to inconsistencies as projects grow and get more mature, and introduce therefore significant defects into models over time.

The solution to this situation is multi language integration, which is still a challenging and uncommon topic. With multi language integration, modeling languages or DSMLs are integrated in a manner that they refer to the same elements. For example, if a developer changes the name of a component, all references in other modeling languages are updated as well. In the following, we describe the application of our approach to language integration of CFTs and UML-based architecture definitions.

A common scenario regarding the UML for practitioners is to use only a subset of the UML for developing software, which may be viewed as a UML based DSL. Normally, this language tailoring is done through profiles, which define new language elements based on existing meta classes and add (OCL) constraints. This yields a DSL with a UML-based concrete syntax. Optionally, the abstract syntax of the DSML may be specified as a meta model.

Our UML based DSML supports only very basic concepts, which are COMPONENTS, PORTS, and RELATIONS. Relations are specialized into generalizations, aggregarions, compositions, and directed, as well as non-directed relations. Component types may consist of component proxies that provide an instance of a defined component type. Other aspects of the UML are not considered in the following. Our multi language integration approach supports two different types of coupling: referencing and harmonization. For our example, we use the tighther harmonization approach. Referencing however is applicable for situations, in which a more light-weight integration is necessary.

Referencing supports the integration of incompatible meta models with each other. This technique does not require modification of any of the meta models that are integrated, and therefore enables integration of existing tools and MDD approaches even when they cannot be modified easily. Here, meta model integration is performed through explicit references, mapping functions, and synchronization. Model elements are explicitely mapped from one modeling language to another one through mapping functions. These functions define links between both meta models, and are kept in an independent model. A concrete realization of a referencing based approach was published in [13], which illustrates this concept of language integration in greater detail.

Considering our multi language integration example, the set of functions $f \in F$ define explicit mappings from CFTs to UML, while $f^{-1} \in F^{-1}$ define mappings from UML back to CFTs. The shown mapping functions ensure that port names are synchronized properly between components - relevant model elements are passed as parameters to the mapping functions. Additional functions would cover the remaining properties of ports, e.g. they would ensure that ports are deleted synchronously, or they would handle different language elements. All language elements that are not explicitely known to mapping functions are ignored. Since mapping functions only extract relevant model information, they are resistant to meta model extensions. Another benefit, which is illustrated in as well is the ability to provide 1:n synchronization - in CFTs, one UML port

may be represented by several input or output events, which represent different error modes. In the example, there are unexpected values and high delays for port A. This situation is also easlily resolved by mapping functions, because the mapping is performed for each pair of conforming model elements.

The referencing based approach is applicable in situations, in which it is not possible to modify the meta models and/or tools that are integrated with each other. Meta models of both languages remain unchanged, and therefore are not affected by the integration. This also holds for existing transformations, model checking approaches, and validations.

In our example, two DSMLs are integrated, which both may be modified for integration. This enables application of our harmonization approach, which provide a tighter, and more efficient integration of languages than referencing. It modifies the meta model of both languages but ensures that the integrated meta models remain compatible to those of the original languages. Figure 5 illustrates our approach with a generic meta class as example. Here, first the common base class g is created that holds common properties of classes e and f. Joint properties, such as the *proxies* property are moved to the common base class. Afterwards, the user visible meta class h is created, which is used for modeling and contains all properties of all integrated classes.

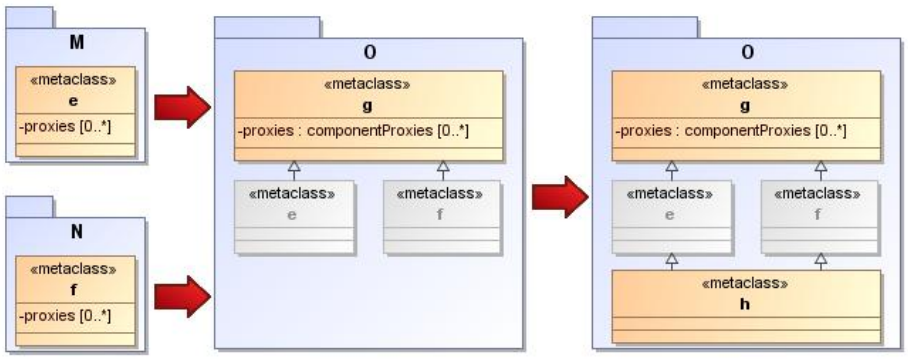

Fig. 5. Meta model harmonization example

This algorithm ensures a conforming integration of two DSMLs - its detailed and formal description is the following:

- If two language elements e and f were distinct in the originating meta models M and N, there are two distinct language elements e' and f' in the combined meta model O.
- All language elements e' in the integrated meta model O have at least all properties of their originating language elements e from each originating meta model.

– All language elements e' in the integrated meta model O are compatible to their source type (i.e. also have the type) e from their originating meta model M.

This guarantees that existing profiles, model transformations, validations, and model checkers may still be used with the integrated meta model, which is our definition of compatibility. Integration on the other hand guarantees the following:

– If two language elements e and f were similar in both originating meta models M and N, they are represented by only one element g' in the resulting meta model O. This applies to language elements, relations, and attributes.
– Attributes with conflicting types from originating meta models are not harmonized.
– Attributes with conflicting default values are given a new default value that conform to the integrated meta model.

These definitions may be extended to integrated meta models that originate from more than two meta models. A tighter integration is applicable when meta models of modeling languages are sufficiently compatible to each other, e.g. when they are built on a common meta model. The big advantage of this integration approach is an implicit synchronization across modelling languages. Since modelling languages are now parts of one holistic meta model, it is not necessary to duplicate information anymore. Therefore, information is always accurate and synchronized.

Meta model harmonization is our approach for tight meta model integration. This is not a new technique - both, the UML and the MOF apply this technique to merge language packages that define subsets of their respective meta models. Package merging of one package with another creates the union of both packages, which spans meta classes, attributes, and relations. Resulting meta classes hold all properties of the original meta classes. The existing package merge relation as defined by the UML is ambiguious in some situations though (see [5]). The authors of [5] propose different semantics for package merge, which will also be applied in the following for merging of abstract syntax. When it comes to the concrete syntax, most UML modeling tools do neither support the extending of their meta models, nor do they support the semantics of package merging. Therefore, we map this merging approach to UML profiles. This is a feasible approach, because many DSMLs may be specified as UML profiles, and, as already mentioned, commonly the UML as language for defining behavior and architectures is often restricted by profiles as well.

Integration of meta models and/or UML profiles requires the integration of meta classes, attributes, and relations. We use the following approach for harmonization of source meta models or profiles M and N (in the following, meta classes are used as synonym for meta classes/stereotypes):

– Definition of the harmonized meta model or profile O.

- Harmonization of meta classes/stereotypes from M and N. For every pair of meta classes e and f that are similar in originating meta model, a new meta class g is created that generalizes e and f. Additionally, a new meta class h that specializes e and f is created.
- For realization through profiles only: the stereotyped meta class(es) of g is set to union of the stereotyped meta classes of e and f.
- Harmonization of class attributes for all meta classes $e \in M$ and $f \in N$. Every attribute a of meta class e, for which $a \in e$ and $a \in f$ holds is moved to g, iff meta properties of a can be harmonized (see below). Attribute remains in e or f otherwise.
- Harmonization of regular relations r: Regular relations are handled like attributes, therefore, the rules for attributes apply here as well.
- Harmonization of generalizations g: Generalization relations g between two elements i, $j \in M$ are added as new generalization relations g' between elements i' and $j' \in O$.
- Harmonization of aggregations a: Aggregation relations are handled like attributes.

For the harmonization of attributes, additional meta properties need to be checked to ensure proper harmonization. These properties are the following:

- $defaultValue$ defines the default value of a property. Here, a suitable new default value must be found if default values from both harmonized elements $e \in M$ and $f \in N$ differ from each other. If no default value is found, no default value is assumed for $g \in O$.
- $isComposite$ defines whether a property is contained in its owner or not. If the value of this property differs in source elements $e \in M$ and $f \in N$, a creative decision is to be made by language developers to avoid meta modeling conflicts. this may yield the decision of keeping both properties seperated, because they obviously follow different (static) language semantics, or to decide for one value for $g \in O$.
- $isDerived$ defines whether a property value may be derived from other properties. If the value of this property differs in source elements $e \in M$ and $f \in N$, isDerviced needs to be set to false in $g \in O$, because its value needs to be set manually and cannot be calculated.
- $isReadOnly$ defines whether a property may be modified. If $e \in M$ and $f \in N$ differ, the value for $g \in O$ is false. This may open unexpected privileges that were not given by original meta models, which is necessary in this case.
- *opposite* defines an opposite value for an attribute. If values from both originating meta models differ, again a creative decision is necessary, because there is no automated way for harmonization.
- The *name* property is typically the same and kept during harmonization, the type is set to the harmonized input type g that was created during the second step of our approach.

Language-specific presentations ensure that in each view only properties that are of relevance to the currently used DSML are visible. For example, the safety

engineer that models with CFTs only perceives CFT related properties of the Component modeling elements, software architects only perceive UML related properties. Realization of views is tool specific - in our case, we used the DSL engine of MagicDraw [15] to realize these views.

Meta model harmonization also has implications on the semantics of modeling languages. When a component with an associated CFT is generalized, the sub-components inherit the CFT of the parent component. Therefore, they may add new failure analysis, for example for new ports that are introduced by specialized components. A thorough analysis of these implications for all possible integration scenarios is considered to be future work.

6 Conclusion

From a safety point of view, safety analyses are indispensable for ensuring the safety of embedded systems. Particularly for complex, software intensive systems, safety analyses is very time consuming and error-prone. The approaches we illustrated in this paper, have shown in their practical application that applying the principles of model driven engineering to safety analyses yields a promising means to reduce the effort and to increase the quality of the analyses. This is particularly true, if the safety analysis models are seamlessly integrated to the functional development models leading to a higher potential for reuse as well as automated generation and consistency checks of safety models. From a DSML point of view, the integration of safety models has shown the feasibility that the concepts of multi language development can be used to integrate non-functional models to functional development models. The harmonization of the modeling language based on profiles as well as the model transformations developed enables the integrated use of accepted modeling environments and proven-in-use analysis tools. The approach has been implemented and evaluated in context of a cooperation project between Fraunhofer IESE and Siemens-CT. The implementation of the modeling language based on the profiling mechanisms of the UML in combination with additional concepts for the multi language development was fife times more efficient than a traditional development approach (a team that developed a standalone CFT modeler using a traditional development approach required more than five times as much effort as an independant team that using the DSML-based modeler). Besides this speed-up factor, a further very important advantage is the flexibility of the approach. The CFT extensions can be easily modified and extended, which enabled a fast tailoring to the organization-specific and even the project-specific requirements, which has been a crucial advantage over the inflexible off-the-shelf solutions. The modeling language was evaluated by applying to several real world examples of Siemens-CT. The building of safety models according to the functional model was practical in all examples as a reasonable overall structure for the component fault trees was already given. The approach presented in this article therefore provides a very good basis for future work. Regarding safety, this particularly means the inclusion of further important safety models, like hazard and risk analyses or safety

case models. For safety and for the NFP modeling in general, it is very important to further advance the integration over different non-functional models.

References

1. M. Bozzano. ESACS: An integrated methodology for design and safety analysis of complex systems. In *Proc. of European Safety and Reliability Conf. ESREL*, pages 237–245, 2003.
2. P. Cuenot, D. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, R. T. Kolagari, M. Törngren, and M. Weber. Towards improving dependability of automotive systems by using the east-adl architecture description language. pages 39–65, 2007.
3. W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Bde. Boosting Re-use of Embedded Automotive Applications Through Rich Components, 2005. Proceedings, FIT 2005 - Foundations of Interface Technologies.
4. M. A. de Miguel, J. F. Briones, J. P. Silva, and A. Alonso. Integration of safety analysis in model-driven software development. *Software, IET*, 2(3):260–280, June 2008.
5. J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software and Systems Modeling*, 7(4):443–467, October 2008.
6. Essarel homepage. URL: http://www.essarel.de/index.html. Last accessed on 2010/08/02.
7. P. Ganesh and J. Dugan. Automatic Synthesis of Dynamic Fault Trees from UML SystemModels. *13th International Symposium on Software Reliability Engineering (ISSRE)*, 2002.
8. L. Grunske. Towards an Integration of Standard Component-Based Safety Evaluation Techniques with SaveCCM. *Proc. Conf.Quality of Software Architectures QoSA*, 4214, 2006.
9. Isograph homepage. URL: http://www.isograph-software.com/ftpover.htm. Last accessed on 2010/08/02.
10. A. Joshi, M. Heimdahl, M. Steven, and M. Whalen. Model-Based Safety Analysis, 2006. NASA.
11. B. Kaiser, P. Liggesmeyer, and O. Mäckel. A new component concept for fault trees. In *In proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS03), Adelaide*, pages 37–46, 2003.
12. B. Kaiser and A. Zocher. Bdd complexity reduction by component fault trees. In *In proceedings of the European Safety and Reliability Conference (ESREL 2005), Adelaide*, pages 1011–1019. Balkema Publishers, 2005.
13. T. Kuhn, S. Kemmann, M. Trapp, and C. Schaefer. Multi-language development of embedded systems. In *9th OOPSLA DSM Workshop*, Orlando, USA, 2009.
14. P. Liggesmeyer and M. Rothfelder. Improving system reliability with automatic fault tree generation. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 90 –99, 23-25 1998.
15. Magicdraw homepage. URL: http://www.nomagic.com/. Last accessed on 2010/08/02.
16. Y. Papadopoulos and M. Maruhn. Model-Based Automated Synthesis of Fault Trees from Matlab.Simulink Models. *International Conference on Dependable Systems and Networks*, 2001.
17. A. Rugina. System Dependability Evaluation using AADL (Architecture Analysis and Design Language), 2005. LAAS-CNRS.