Bachelorarbeit

Konzeption und Entwicklung eines Werkzeugs zur Spezifikation und Analyse von Safety-Modellen auf Basis von Enterprise Architect

Jan Reich

Datum: 10/2013

Fachbereich Informatik, Technische Universität Kaiserslautern
Fraunhofer Institut für experimentelles Software Engineering Kaiserslautern

Konzeption und Entwicklung eines Werkzeugs zur Spezifikation und Analyse von Safety-Modellen auf Basis von Enterprise Architect

Bachelorarbeit

von

Jan Reich

31.10.2013

Erstprüfer:	Prof. Dr.-Ing. habil. Peter Liggesmeyer, AG Software Engineering: Dependability
Zweitprüfer:	Dr.-Ing. Mario Trapp, Fraunhofer IESE
Betreuer:	M. Sc. David Santiago Velasco Moncada, Fraunhofer IESE

Erklärung

Hiermit erkläre ich, Jan Reich, dass ich die vorliegende Bachelorarbeit mit dem Thema

„Konzeption und Entwicklung eines Werkzeugs zur Spezifikation und Analyse von Safety-Modellen auf Basis von Enterprise Architect“

selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

.....

Jan Reich,

Kaiserslautern, den 31.10.2013

Kurzfassung

Im Kontext des Projektes *Software Platform for Embedded Systems 2020* (SPES2020) wurde die konzeptionelle Grundlage dafür geschaffen, die Erstellung und Analyse von Safety-Modellen in den Prozess des System Designs für sicherheits-kritische und software-intensive eingebettete Systeme zu integrieren. Außerdem wurde ein generisches Austausch-Format für Safety-Modelle entworfen, das den Modell-Austausch zwischen verschiedenen Modellierungs- und Analyse-Werkzeugen ermöglicht. Um die erforschten Konzepte für die Praxis attraktiv zu machen, müssen diese allerdings in die industriell benutzten Werkzeuge integriert werden. Diese Arbeit beschäftigt sich mit der Entwicklung eines Prototyps für das Modellierungs-Werkzeug *Enterprise Architect* (EA), der die Erstellung und Analyse von Safety-Modellen innerhalb dieses Werkzeugs ermöglicht. Die konkret implementierte Analyse-Methode ist die Fehlerbaum-Analyse mit *komponenten-integrierten Fehlerbäumen* (C²FT). Es wird gezeigt, dass in EA modellierte C²FTs in das generische Safety Austausch-Format überführt und anschließend mit den Fehlerbaum-Analyse-Werkzeugen des *Fraunhofer IESE* und der kommerziellen Software Isograph *FaultTree+* analysiert werden können. Der Hauptteil der Arbeit besteht aus der Entwickler-Dokumentation der eingeführten Schichten-Architektur und deren konkreten Implementierung für EA. Zusätzlich wird das bestehende C²FT-Meta-Modell evaluiert und im generischen Safety-Austausch-Format dahingehend erweitert, dass zukünftig sowohl weitere Safety-Analyse-Techniken wie z.B. *Failure Mode and Effect Analysis* (FMEA) in das Format integriert werden können als auch eine Safety-Analyse durchgeführt werden kann, die unabhängig von den verwendeten Analyse-Techniken ist.

Abstract

In the context of the project *Software Platform for Embedded Systems 2020* (SPES2020) the conceptual foundations have been researched that allow to integrate the creation and analysis of safety models into the system design process for safety-critical and software-intensive embedded systems. Furthermore, a generic safety exchange format has been developed that enables the exchange of safety models between different modeling and analysis tools. In order to make the researched approach applicable in practice, it has to be integrated into those tools that are commonly used in the industry. This thesis deals with the development of a prototype for the modeling tool *Enterprise Architect* (EA) that is able to create and analyze safety models within this tool. The considered analysis technique is the *fault tree analysis* with *component-integrated fault trees* (C²FT). It's demonstrated that C²FT models that are created in EA can be transformed into the generic safety exchange format and subsequently analyzed with the fault tree analysis tools of the *Fraunhofer IESE* and the commercial software *FaultTree+* by Isograph Company. The greater part of this thesis consists of the documentation for developers including the introduced layered architecture as well as its implementation for EA. In addition, the existing C²FT meta-model is evaluated and the generic safety exchange format is extended in a way that on the one hand, further safety analysis techniques like *Failure Mode and Effect Analysis* (FMEA) can be easier integrated into the format and on the other hand, safety analysis can be performed for safety models that incorporate different analysis techniques at the same time.

Table of Contents

1 Introduction

According to [1], 98% of all produced microprocessors are installed in embedded systems. These embedded systems are used in various application areas and are omnipresent in our daily lives: the majority of automobiles and airplanes are controlled by embedded systems. In the medical sector they are used for example for the monitoring of vital functions or for carrying out irradiation therapies. Furthermore, embedded technologies are extensively used in devices for consumer electronics and the power supply industry. This implies that special attention should be paid to the development of embedded systems and their software.

Numerous examples demonstrate in its own right that embedded systems are safety-critical in many respects and can have an impact on the life and limb of people. Due to the steady increase in the complexity of such systems new techniques need to be developed constantly that allow to master the complexity, to deliver the required product quality, and to satisfy the safety requirements simultaneously. There exist international standards like IEC61508 [2] or ISO26262 [3] that propose techniques for anchoring safety engineering tasks into the whole development process. This includes for example the consistent realization of safety analyses throughout the development process. The safety analysis proved difficult particularly for complex system architectures; therefore, appropriate techniques need to be employed to keep the complexity as low as possible.

The research project *Software Platform Embedded Systems 2020* (SPES2020) was started in 2009. One of the research tasks was to perform research for the development of concepts for software-intensive embedded systems applicable across industry domain boundaries. According to [4], the main objective of this project was to master the steadily increasing complexity of these systems by providing an integrated concept ranging from requirements down to the resulting program code, independent of the specific domain. This includes the identification of similarities between embedded systems from different domains, and a high capability for the automation of process tasks in tools. The essential result of the project was that *model-based design* (MBD) and *component-based software engineering (CBSE)* are key components for overcoming the complexity challenge mentioned above. Although MBD and CBSE are well suited for the reduction of system development complexity, their principles were not considered in traditional safety analysis techniques. In order to enable the seamless integration of the established technique *fault tree analysis* with the SPES2020 concepts, traditional fault trees were extended to the concept of *component integrated fault trees* (C²FT), which follow the principles of CBSE and introduce formal rela-

tions between design components and *component fault trees* (CFT) for the first time.

The follow-up project SPES_XT uses the results of SPES2020 and addresses some new requirements with respect to the integration of safety analysis techniques. In first place different safety analysis techniques are needed on arbitrary system hierarchy levels as well as combined in one failure model in order to allow dependable and adequate statements about system safety. In addition to that, a tool architecture should be developed that decouples the safety models from the usage of specific frontend modeling and backend analysis tools. The vision of SPES_XT is that specific failure models of different frontend modeling tools can be transferred into and from one safety exchange format on which the backend analysis tools can operate on. This allows the use of arbitrary front end modeling tools as well as a better optimization of the analysis algorithms.

In the course of SPES_XT, the Fraunhofer IESE implemented such a tool architecture, which currently supports the safety analysis of CFTs with the algorithms of the commercial software FaultTree+ from Isograph Company and incorporated an own implementation of analysis algorithms for CFTs. The only supported front end modeling tool is *MagicDraw* (MD) from NoMagic Company. The MagicDraw implementation was primarily developed as a constantly evolving prototype in order to evaluate the results of SPES2020 in practice.

Enterprise Architect (EA) from Sparx Systems Company is widely accepted across the industry in architecture modeling of embedded systems and is preferred by some customers over MagicDraw; EA should therefore be integrated as a frontend modeling tool into the tool architecture of SPES_XT, too.

The main focus of this thesis rests on the implementation of this new front end to serve as a prototype for the integration of further front ends into the tool architecture, thus an integral part of this thesis is to provide sufficient documentation to satisfy the desired prototype character.

Before the integration of the EA front end could happen, the existing safety exchange format's meta-models, developed according to the results of SPES2020, had to be evaluated on the basis of the experience made with the available MagicDraw implementation. The source code of the IESE analysis back end, which had used the former meta-models, had also to be changed accordingly.

After that, UML profiles have been created in EA, which represent the safety exchange format's failure meta-models on the front end side. In order to add more complex functionality and enable the connection to analysis back ends,

an EA add-in has been developed on the .NET platform. Before that, conceptional considerations had to be made on where the dividing line between front end side and the safety exchange format implementation should reside.

The next step has been the transformation of the front end failure models into the safety exchange format as well as the development of the connection to the analysis back ends, so that analysis tasks were able to be invoked from within the modeling front end tool. Eventually, UI dialogs inside the EA add-in had to be implemented, which present the results of the supported safety analysis operations to the user in a comprehensive and attractive way.

In order to explain how the objectives of this thesis have been achieved, the following structure has been chosen:

Section “Theoretical Foundations” gives an introduction on state-of-the-art system design principles and describes the impact that these principles had on the meta-models of fault tree analysis.

With this foundation, section “Layered System Architecture” gives an overview of the relevant requirements of SPES_ XT and describes the resulting layered architecture consisting of front end layer and model transformation layer. This includes the functional contents of the layers and their communication with one another. In addition, the challenges for the integration of analysis back ends are discussed.

The documentation of the layer architecture’s implementation with the front end EA is the content of the next section "System Documentation". Initially, the failure meta-models and abstract concepts of the current version of the safety exchange format implementation are presented. Then, the design of the model transformation layer’s implementation is described. The second part of this section is the description of how the front end EA has been integrated. This includes the created UML profiles as well as the design of the EA add-in. The section ends with crosscutting design aspects applicable for both front end layer and model transformation layer and an overview which back end tools and analysis operations have been integrated so far.

The last section of the thesis consists of the evaluation of the developed prototype as well as some possible directions for future work. In addition, it contains a conclusion that discusses how well the thesis’ goals have been achieved.

2 Theoretical Foundations

This section gives an introduction to the theoretical foundations on which this thesis is based. Section 2.1 depicts the state of the art in the architectural design of software-intensive embedded systems. Then, section 2.2 introduces fault tree analysis and explains how far integration into architectural systems has been accomplished.

2.1 Architectural Design

Due to the high complexity that is inherent to today's embedded systems and therefore also to their development, high efforts have to be invested in order to master this complexity. On top of that, safety criticality adds an extra level of complexity to the development, because several additional activities need to be integrated in the whole development process to be able to satisfy safety requirements. For large systems, the adherence to development methods that apply proven principles helped to satisfy these requirements in the past. These principles and methods for handling complexity are explained with more detail in the following.

- Modular Decomposition

Modular Decomposition is "the process of breaking a system into components to facilitate design and development" [5].

This means that a divide and conquer approach is applied, which identifies clearly defined units of the system. The interface of such a unit is represented by ports, which can be connected and thus define the relationships between units in a clear manner. Its advantage is that developers don't need to have an overall understanding of the system that should be developed, but can concentrate only on that part of it, which is relevant for their specific development task. This technique is also known as *distributed development*. Ideally, when modular decomposition is applied correctly, the development of a system's components can in theory be happen completely in parallel. However, this is achieved in practice often only to a certain degree.

- Integrated Views

A *View* is "a representation of a whole system, from the perspective of a related set of concerns" [5].

In particular in embedded systems several different views can be defined according to this definition, e.g. functional or logical views. In addition, non-

functional properties like safety, which is considered in this thesis, can be expressed as views. A system's separation into views has the benefit that different aspects of it can be treated separately. However, in most cases, elements can be part of different views simultaneously, because the underlying system is the same. In order to enhance traceability between views, they have to be integrated with each other accordingly. Note that the given definition can also be applied for components instead of whole systems.

- Interface Abstraction

When modular decomposition is applied, components are refined into subcomponents which are themselves further refined into subcomponents and so on. Thus, a hierarchy is created for the purpose of reducing complexity. This complexity reduction is only achieved, when a limited amount of information is needed to understand a specific component in the hierarchy, i.e. when subcomponents should be used in this component, they have to be modeled as black boxes omitting their concrete realization, which is not needed to model the realization of the component itself. Instead, only the subcomponents' interfaces need to be exposed to understand the behavior of the component. This approach is called interface abstraction and enables, apart from complexity reduction, reuse, the easy exchange of a component's realization, when its interface stays unchanged, and the division of labor.

A typical example for the need of both modularization and interface abstraction can be seen in the industry, when a supplier company delivers a realized component to the OEM Company for integration into the overall system. The OEM doesn't need to know all realization details for integrating the component with other components, when a complete description of the interface is available.

- Component-Based Software Engineering and Model-Based Design

The three explained principles for handling complexity are the foundation for the development method *Component-Based Software Engineering* (CBSE) (see [6] for a comprehensive characterization of the method) that is a commonly used method in state-of-the-art development of embedded systems in general. Thus, it has also become the proposed development method in the context of the SPES2020 project.

Apart from CBSE, the *Model-Based Development* (MBD) method in general also adheres to the mentioned principles. According to [7], MBD focuses on the minimization of redundancy during the development of software systems by creating abstraction models that can express domain-specific problems in a much clearer way than classical approaches are capable of. Usually, the executable source code for the target platforms is generated from the models. In addition, the applied formalisms in MBD provide a very high potential

for automation. For example, changed requirements can be traced down automatically to the affected architectural elements, because there is a formal relation between them.

The considered language in this thesis is the semi-formal *Unified Modeling Language* (UML) that supports modular decomposition and partially the notion of views by default. Interface abstraction and integrated views can be integrated into the UML by its extension mechanism, the so-called UML profiles, that will be of interest in this thesis and therefore covered in section 4.2.1 in more detail.

2.2 Fault Tree Analysis

This section introduces the safety analysis technique *fault tree analysis* (FTA) and its relation to architectural component models.

2.2.1 Technique Description

FTA is a deductive failure analysis technique that is suggested by several standards as part of functional safety assessment, e.g. the international standard IEC 61508 [2]. Its aim is to perform a top-down search for causes of a specific dangerous failure that can occur, typically depicted as the top event in a model. The causes are modeled by so-called basic events and represent atomic sources for failure. The idea is to relate these basic events logically by using so-called gates that represent Boolean operators such as AND, OR, NOT or XOR.

Two kinds of analysis could be performed in FTA, namely qualitative and quantitative analysis.

The most used and best known qualitative analysis operation in FTA is the minimal cut set computation, which provides minimal cut sets for coherent fault trees (FT) or prime implicants for non-coherent fault trees. A fault tree is coherent, if it doesn't contain any NOT gate. Both minimal cut sets and prime implicants are minimal combinations of basic events that cause the analyzed top event to occur. This analysis type is typically used for the identification of those basic event sets with the smallest number of events, because in the extreme case, they represent a single point of failure in the system. Thus, they have to be considered with priority in system design in order to build safe and reliable systems.

Quantitative analysis can be performed, if probability distributions can be assigned to all basic events in a fault tree. In this case, the minimal cut sets computed in the qualitative analysis can be additionally ordered after their occurrence probability. Due to the fact that probabilities for basic events are often not known exactly, e.g. when they represent software failures, it's also

possible to choose random probability values from defined boundaries. In this case, the analysis is called semi-quantitative.

2.2.2 Fault Tree Model Evolution

The fault tree theory has been considered in many industry domains since its invention by Bell Telephone Laboratories for a rocket-launch control system in 1961 [8]. Because of the fact that traditional fault tree theory had been a well-researched area, it was also applied for software code analysis since the 1980s [9].

As described in section 2.1, the complexity in embedded systems development is successfully handled by the application of the development methods CBSE and MBD. However, with traditional FTA, the principles inherent to CBSE and MBD are not adhered to, so all their benefits are lost, because the continuous model synchronization between component model and FT model requires huge additional effort in this case. Thus, the principles need to be applied for FT models as well.

An important improvement of traditional FTs addressed the problem that their structure had been flat, i.e. they did not have a reduction of complexity through modularization. This resulted in FT models that were hard to use together with their related component models. As a result, *Component Fault Trees* (CFT) were introduced by [10], which were able to reflect a modular structure in parallel to the basic structure of components. This was achieved by the introduction of both input events, which represent transfer elements that allow the division of a flat fault tree into modular sub trees, and special components, which allow the representation of these subtrees.

“Component Fault Trees cannot be directly integrated with the generic *Component Model* [...] because they do not support *Interface Abstraction*.” (p. 59, [11]). Thus, the next improvement step was taken to *Component Integrated Fault Trees* (C²FT), which were proposed in [11]. The C²FT model defines a formal relation between the component model and the CFT model, so failure modes of fault trees are formally mapped to their functional counterparts. This ability allows for consistency checks and traceability between both views. In addition, the components can be reused together with their CFTs because of the formal relation.

3 Layered System Architecture

This section addresses the layered system architecture, which originates from the requirements of the SPES_XT project. In section 3.1, the most important requirements are outlined. Section 3.2 presents the resulting architecture from a high level perspective describing the interface to the system's environment as well as a summary of the system contents. The following sections 3.3 and 3.4 present structure, runtime behavior and interfaces of each layer in more detail. At last, section 3.5 describes different appearance types of back end tool APIs as well as typical problems occurring with their integration.

3.1 Primary Requirements

Because of the fact that the SPES_XT project addresses requirements of several different areas, those relevant for this thesis are described in the following.

The flexibility for tooling choices should be improved for stakeholders.

In the last decade, the application environment in the area of safety modeling and analysis for embedded systems has evolved to a variety of solutions by different vendors. According to the consensus in the industries, these solutions usually support several different analysis techniques, but in most cases have their strength solely in one technique, which forces companies to use tools from different vendors in order to get the best results. Most of the available tools don't support a common exchange format for failure models, so the model exchange between different tools is aggravated.

A key objective of SPES_XT concerning safety development is the seamless integration of safety analysis with system design. As a consequence, joining pure architectural modeling tools and pure safety modeling and analysis tools would reduce the dependability of stakeholders on specific tools significantly. As a consequence, architectural modeling tools must be extended to include the ability to create failure models and the exchange of failure models between different tools must be substantially simplified. The second task is in its entirety beyond the scope of this thesis, but the developed prototype paved the way for coping with this task in the future.

The consideration of those findings allows the flexibility to choose the best tool set in a specific context; for example, OEM integrators and suppliers in the automotive industry usually use different tool sets for the same analysis

technique, but the missing exchange format inhibits this degree of freedom, when much manual rework should be avoided.

It should be possible to combine heterogeneous safety analysis techniques within one failure model.

Because of the fact that the complexity as well as the required safety level of the developed systems and the components have grown, traditional safety analysis techniques like FTA have reached their limits. Thus, the necessity arose to research modular variants for analysis techniques like *Markov analysis* to be applicable within the SPES Modeling Approach. Every technique has its own separated failure meta-model, so it has not been possible by now to model the components of a system with different failure models and to perform safety analysis for the overall system then. Hence, a new approach needs to be introduced, which abstracts from specific analysis techniques and is able to be used for overall safety analysis.

From an industrial point of view, the expertise in the application of specific analysis techniques also plays an important role, because a company having the choice between different techniques with comparable results will obviously choose the variant that requires less time.

Apart from the available expertise, the system's required safety integrity level dictates the process of safety development including the mandatory safety analysis techniques, which can be more than one, dependent on the level, i.e. it's likely that several techniques *must* be applied simultaneously. An example for such a standard is ISO26262 [3], which covers safety aspects in the automotive industry and includes four (automotive) safety integrity levels (ASIL) with different analysis technique recommendations per level.

3.2 Big Picture

The proposed architecture for fulfilling the requirements stated in the preceding section introduces two intermediate layers between architectural modeling tools and safety modeling and analysis tools as shown in Figure 1.

The architectural solution's context consists of three parts, namely the *modeling front ends*, the *safety exchange layer* and the *analysis back ends*. To avoid naming ambiguities, when the terms *modeling front end* and *analysis back end* are used throughout the thesis, this includes the actual application sold by the vendor as well as the provided application programming interface (API) the *safety exchange layer* can make use of.

A selection of available tools for architectural modeling and safety modeling and analysis is also shown in Figure 1. The safety analysis tools are connected to the analysis techniques they are able to handle [12] [13] [14] [15].

Note that, the architecture's implementation in section 0 only deals with those elements having a bold label, while the regular labeled elements are either already integrated like the MagicDraw front end or likely to be integrated in the future. In addition, the distinction between modeling and safety analysis tools should not be understood as a sharp separation, because it's conceivable that a safety analysis back end tool also has modeling capabilities. In this case, the tool's usage context within the specific project needs to be taken into account.

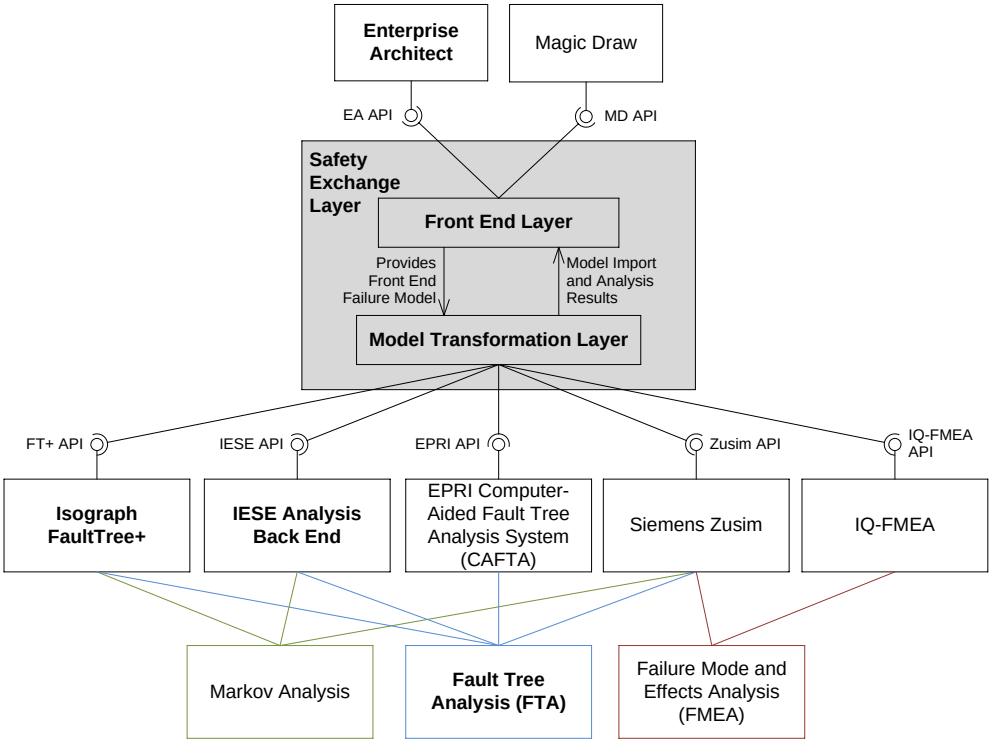


Figure 1 System Context Diagram

The *safety exchange layer* contains the two layers *front end layer* (FEL) and *model transformation layer* (MTL).

The FEL is mainly responsible for providing the additional functionality in order to integrate safety modeling tasks directly into the specific front end application. This is usually done by means of UML profiles and extension mechanisms of the front end applications (e.g. add-ins in Enterprise Architect or plugins in MagicDraw), which allow the definition of complex constraints for the stereotypes that exceed the capabilities of UML profiles. If safety analysis is also required, a connection to the MTL has to be established through which the front end failure models can be transferred to the MTL and analysis tasks can be triggered.

The MTL includes the core component for achieving model exchangeability between different tools: the safety exchange model. The purpose of this collection of meta-models is to incorporate all relevant properties of the supported safety analysis techniques so that a failure model modeled in any tool can be transformed into and from the safety exchange model. These transformations are provided by model transformers which have to be written for each technique-modeling tool combination. The specific UML profiles from FEL have to be created according to the previously defined safety exchange model. The second responsibility of the MTL is the communication with back end analysis tools in order to perform safety analysis tasks.

The benefit of connecting modeling front ends by implementing front end layers and model transformers for them is drawn from the fact that analysis support is almost directly given, because connected back end analysis tools only depend on the safety exchange model.

3.3 Front End Layer

This section describes the functional decomposition of the FEL and the interaction structure of the identified functional units resulting from the FEL requirements.

3.3.1 Front End Layer Requirements

As suggested in section 3.2, the FEL consists of the implementation of the safety modeling functionality and the triggering of analysis tasks. The layer's main requirements, which have to be taken care of, are the following:

The integration of new analysis techniques and failure meta-model changes should be doable fast.

Bearing in mind that research produces newly created or changed meta-models steadily, it's very likely that the FEL has to be changed frequently reacting to alterations. As a consequence, developers need to have a repository that contains frequently used functionality and helps to implement higher-level functions faster, because the basic functions just need to be parameterized for the specific context. In addition, this allows to concentrate rather on the actual changes than on the implementation of the basic functions

The common property of modularity and the support for hierarchies with failure meta-models in the context of SPES2020 need to be abstracted from to further facilitate changes and additions.

Modeling and analysis issues should be separated.

There are two usage scenarios conceivable for the *safety exchange layer*: The first one consists only of the creation of failure models in the front end. The second one adds the ability to analyze the created models, too. Since the analysis issues are optional, the separation between both tasks has to be reflected in the FEL in order to enable independent deployment of both parts.

3.3.2 Functional Decomposition

This section describes the functional decomposition of the FEL. Its purpose is to identify the fundamental functional units, which are necessary to provide the desired behavior. Note that the modeling front end EA and its extension mechanism have had a major influence on the identification of the individual units.

The functional decomposition of the FEL is shown in Figure 2. It consists of four functional packages, namely *Modeling*, *Analysis*, *Profile Functionality* and *Model API Façade*. The package structure has been chosen according to the FEL primary requirements explained in the preceding section.

3.3.2.1 *Model API Façade Package*

The APIs for modeling tools allow programmatical access to the failure models created within the modeling tool. Because of the fact that the considered tools EA and MD support the UML as a whole, the offered APIs are very generic and their application in concrete domains is difficult. In the SPES modeling approach, the usage of only a small subset of UML elements, specifically *Component*, *Part*, *Port*, *Connector* and *Association*, has proven best for the definition of the UML profiles. The reason is that these elements incorporate the principles of CBSE and modularity by default. As a consequence, the *Model API Façade* Package constitutes a façade to the API, which concretizes the generic functionality for the typical usage of the used element subset. In the SPES context, these typical usages are model retrieval, e.g. the retrieval of all instances that were instantiated in a component, model modification, e.g. the synchronization of the port creation for instances to its classifier, and model visualization, e.g. the reflection of the port synchronization in all existing diagrams.

The introduction of this façade facilitates the usage of the provided API and provides a reusable and basic function repository, which is supposed to be used from all other packages in order to implement quickly new functionality or changes alike.

3.3.2.2 *Profile Functionality Package*

The most important unit of this package is the *UML Profile/Diagram Definition*. For EA and MD, these definitions can be created declaratively from within the specific modeling tool.

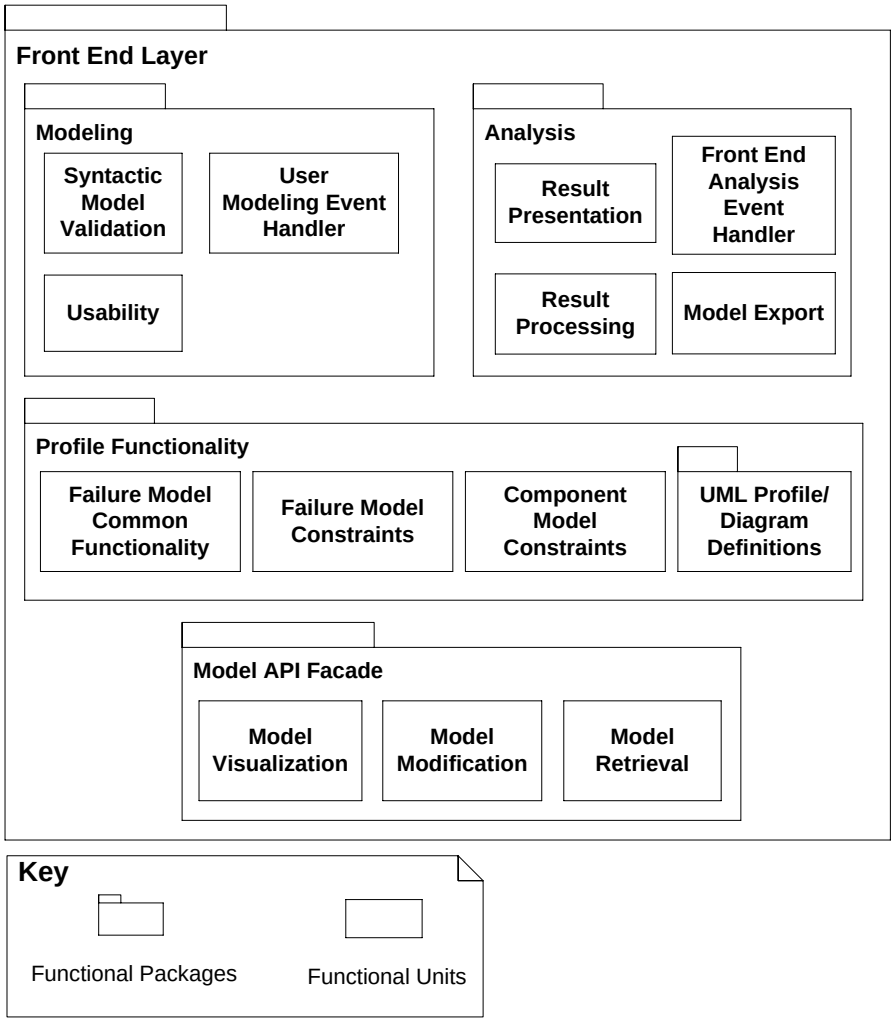


Figure 2 Front End Layer Functional Decomposition

Since the definition of stereotype constraints directly in the profiles is not flexible enough, they have to be expressed by means of a more powerful mechanism. Although the *Object Constraint Language* (OCL) [16] indeed encompasses the capabilities to do so, the complexity of OCL itself is a limiting factor. The chosen approach in this thesis for defining complex constraints is by implementing them in code. The units *Component Model Constraints* and *Failure Model Constraints* have been distinguished to highlight

the fact that the first one only contains constraints for the central meta-model in SPES, while the second one incorporates the constraints for all supported analysis techniques. The constraints are mostly syntactical restrictions, e.g. rules how model elements are allowed to be connected together.

Due to the heterogeneity requirement concerning failure meta-models and the resulting abstraction from specific analysis techniques in the context of SPES, all failure meta-models share some behavior, e.g. the handling of their interface failure modes, the which is put in its own unit to avoid duplication of identical behavior.

3.3.2.3 *Modeling Package*

This package's responsibility is to react to modeling events triggered by either the user or the modeling tool, e.g. creating and deleting elements or selecting modeling commands from custom menus. These modeling actions have to be checked for syntactical correctness according to the defined meta-models. The rule set for the syntactical validation is defined by means of constraints in the *Profile Functionality Package*.

The usability unit contains functionality to facilitate the modeling activity. This includes for example the automation of tasks with several intermediate steps like the synchronization for port changes in a component with all its instances as well as the simplification of tasks which are tedious to achieve by default like the parameterization of a CFT.

3.3.2.4 *Analysis Package*

The *Analysis Package* incorporates all functional units which are related to safety analysis tasks.

This includes the export of front end failure model representations, which is a mechanism for selecting and optionally serializing the relevant combination of failure models and parameters for specific analysis tasks. Serialization is necessary, if the MTL doesn't have direct access to the front end data model.

The unit *Front End Analysis Event Handler* reacts to the triggering of analysis events by the user and controls the analysis process, i.e. it delegates the serialized failure model and the desired analysis operation properties to the MTL, which returns the analysis results after the analysis execution.

Analysis results need to be presented to the user in a comprehensive way, so specific UI dialogs for each analysis operation are located in the *Result Presentation* unit. In real systems, the sheer data size of the analysis results can be huge, so it's necessary to represent them with an appropriate data

structure to enable performant result retrieval needed for visualization. The storage of results directly within the failure model is especially useful, when analysis tasks have a long computation time or when the failure model is reused but not delivered entirely because of intellectual property hiding.

3.3.3 Layer Interface and Interaction Structure

The purpose of this section is to present the FEL's interface to its environment and the interaction structure of the functional units identified in section 3.3.2 during the execution of primary use cases. These issues are visualized in Figure 3 by means of a UML Component Structure Diagram.

3.3.3.1 *Front End Layer Interface*

A key requirement for the proper operability of the FEL is the presence of an API for the modeling tool. The API serves two purposes: Firstly, it provides programmatical access to the modeling tool's data model. This data model is not supposed to be accessed directly by any component inside the FEL except the *Model API Façade* component (see section 3.3.2 for details). Secondly, it asynchronously notifies registered entities of modeling and analysis events, which are either user-triggered or triggered by the modeling tool itself. These events are delegated to *User Modeling Event Handler* and *Front End Analysis Handler*, which implement the logics for the execution of the tasks associated with the events.

The FEL provides two interfaces which are required by the MTL. When safety analysis tasks should be executed, the FEL acts as *Safety Analysis Service Consumer* and has to delegate those tasks to the MTL, including the relevant parameterized failure models. The delegation happens asynchronously, which means that several analysis tasks can be performed in parallel.

The other interface provides direct access to the serialized front end failure model. It is separated from the safety analysis task delegation, because exported failure models could also be used separately, e.g. for the pure exchange of models between tools or for their persistent storage on the file system.

3.3.3.2 *Use Case Execution*

In Figure 3, the components colored in dark grey and light grey illustrate exclusive participation in modeling and analysis activities, respectively, while the red colored components incorporate behavior used in both use cases.

- Safety Modeling

During the whole modeling process, *User Modeling Event Handler* component receives events, which signal the modifications that the user made to the front end's data model through UI or custom modeling commands.

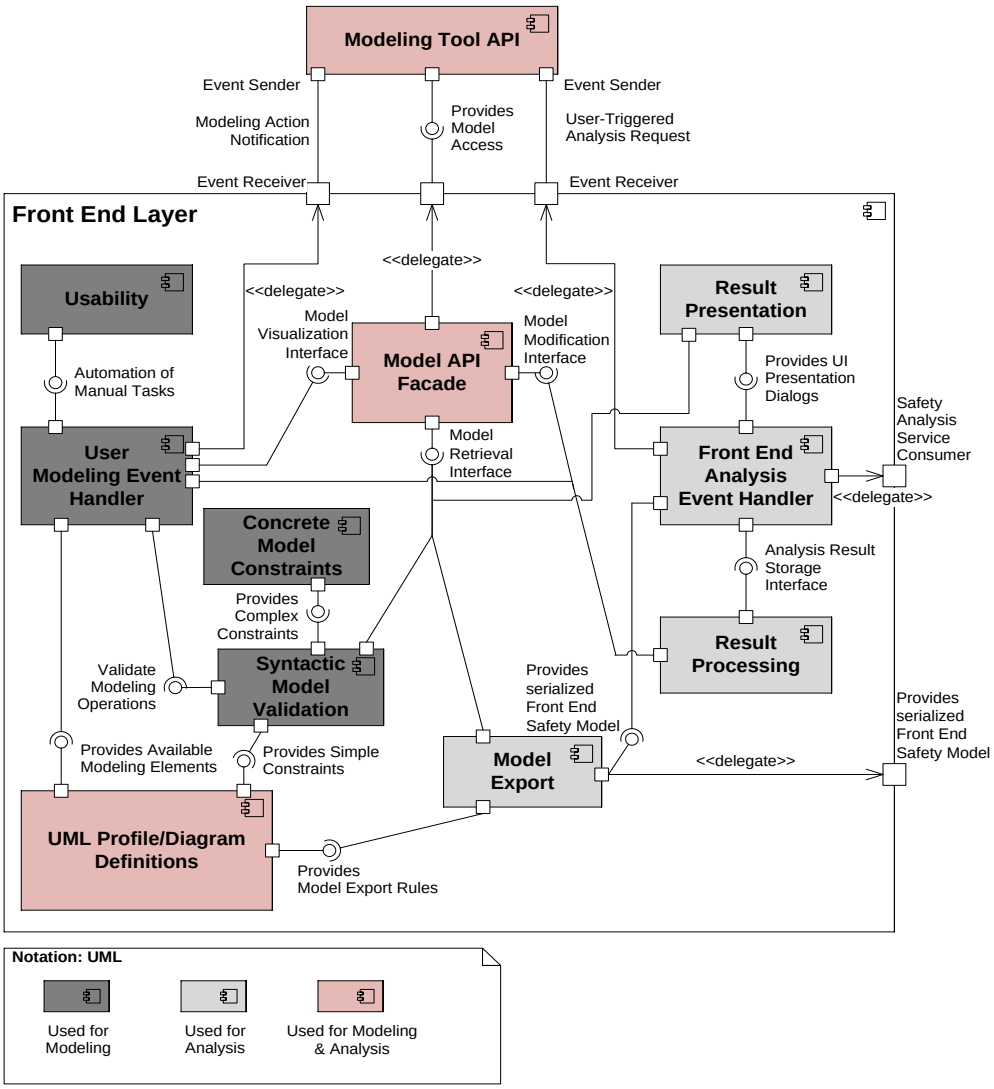


Figure 3 Front End Layer Interaction Structure

In the first case, the modification's syntactical correctness has to be checked against the available constraints. Note that in Figure 3, there isn't any distinction between component model constraints and failure model constraints for the sake of simplicity.

In the second case, the invoked custom modeling commands have to be directly executed by the FEL, i.e. that the modeling tool's internal data model is modified programmatically. The required information for executing the commands is taken from the *UML Profile/Diagram Definitions* component.

- Safety Analysis

When safety analysis tasks are triggered, *Front End Analysis Event Handler* is notified of it. Subsequently, the relevant front end failure model representation is prepared and sent to the MTL, which performs the actual analysis. The use case continues, when the MTL returns either the result of the analysis task or any occurred errors. The results are stored in the front end failure model by *Result Processing* component and can be retrieved by *Result Presentation* component which has the knowledge for filling the UI result dialogs with data.

3.4 Model Transformation Layer

This section describes the functional decomposition of the MTL and the interaction structure of the identified functional units resulting from the MTL requirements.

3.4.1 Model Transformation Layer Requirements

The parts of the primary requirements presented in section 3.1, which are relevant for the MTL, are:

The implementation of the safety exchange model should be provided.

Front end and back end adapter implementations should be separated from the safety exchange model implementation.

Due to the fact that the safety exchange model should not depend on possible adapted modeling front ends and analysis back ends, it is necessary to reflect this separation in the layer structure. On the one hand, this facilitates the impact analysis for changes in the safety exchange model and on the other hand, it enables the ability to individually choose those front ends and back ends that should be incorporated in the deployed MTL binary.

3.4.2 Functional Decomposition

This section describes the functional decomposition of the MTL. It is visualized by a UML package diagram in Figure 4.

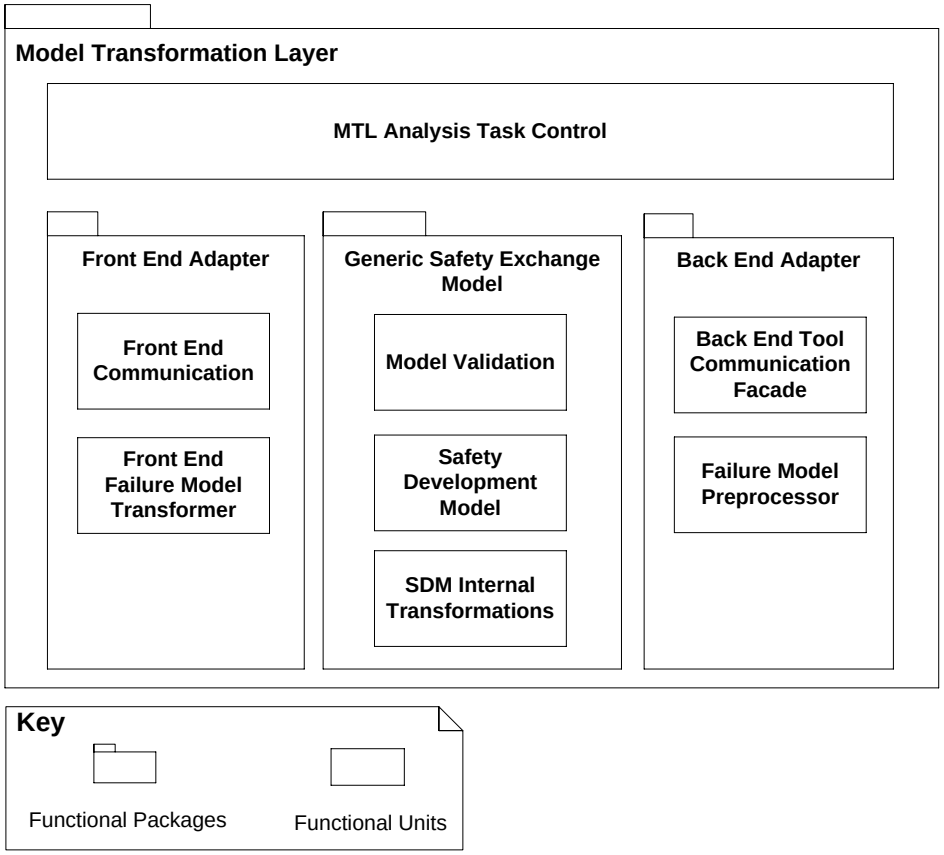


Figure 4 Model Transformation Layer Functional Decomposition

3.4.2.1 *Generic Safety Exchange Model Package*

The most important unit of this package is the *Safety Development Model* (SDM), which is the implementation of the safety exchange model mentioned in section 3.1. Apart from a mechanism to create models in a declarative way, the key objective for the implementation of this unit is to achieve the integration of several safety analysis techniques under a modular perspective. All other units of this package are built around this model implementation and operate on it. In the developed prototype, the chosen modeling framework for generating source code from the declarative defined models is the Eclipse Modeling Framework (EMF) that is Java-based. This is mentioned in this section, because it's an architectural decision that has an impact on MTL development, namely that the other units of the MTL have to be developed in Java, too.

The *Model Validation* unit has the responsibility to check the present failure models for syntactic and semantic rules defined for their meta-model elements. Note that according to section 3.3.2, syntactical validation rules are

also checked in the *Syntactical Model Validation* unit in FEL. Both units have the same underlying rule set, but in order to be flexible concerning pure model import to the MTL without a full-fledged front end adapter, this functionality is duplicated. Nevertheless, the main responsibility of the *Model Validation* unit is to check for semantic rules, such as the check for Boolean loops or instantiation loops in CFTs.

Because of the fact that commercial safety analysis tools do not support the modular variants of failure meta-models developed in the context of SPES_XT by now, it is necessary to be able to transform the modular models into flat models, which can subsequently be directly passed to the respective safety analysis tools. The *SDM Internal Transformations* unit takes care of this task and provides additional functionality performing optimization on the resulting models. An example for such an optimization is the reduction of CFTs. This is possible, because CFTs represent Boolean formulas that can be reduced by the axioms of Boolean algebra.

3.4.2.2 *Front End Adapter Package*

With the SDM implementation as a basis, front end modeling tools are integrated by means of front end adapters, which implement the communication interface towards the FEL and the rules for the transformation from front end specific failure models to the respective failure models from the SDM.

While the *Front End Communication* unit acts as an access point for receiving incoming safety analysis requests and implementing the deserialization of the included failure models, the *Front End Failure Model Transformer* unit implements their transformation to the SDM. Note that there is a difference between this unit and the *SDM Internal Transformations* unit: although both implement transformations, their input models are different. In the first case, the input models are serialized failure models coming from the front end, while in the second case, the failure models are already available in the SDM format.

3.4.2.3 *Back End Adapter Package*

The counterpart to the *Front End Adapter Package* implements the interface to the supported back end analysis tools.

The *Failure Model Preprocessor* unit is responsible for preparing failure models from SDM for the analysis with a specific back end analysis tool. Due to the diversity of such tools' interfaces, the functionality of this unit can be very different. In general, it provides all necessary information for performing the safety analysis.

Related to the above mentioned diversity, the *Back End Tool Communication Façade* is a rather abstract unit, which represents a façade to simplify the usage of a specific back end tool API. It is modeled in the architecture, because a common interface for analysis, which abstracts from the specific issues of a back end tool, should be provided for MTL developers.

3.4.2.4 *MTL Analysis Task Control Unit*

This unit is of rather logical than functional nature, because it only controls safety analysis tasks inside the MTL, i.e. it uses the functionality of the MTL packages to perform the analysis tasks. The introduction of this unit serves for the decoupling of front end adapters from back end adapters.

3.4.3 Layer Interface and Interaction Structure

This section describes analogously to section 3.3.3 the MTL's interface as well as its interaction structure by means of the use case "Safety Analysis". These topics are visualized in Figure 5 by a UML Component Structure Diagram.

3.4.3.1 *Model Transformation Layer Interface*

As depicted in section 3.3.2, safety analysis tasks are not directly executed in the FEL but delegated to the MTL. Therefore, in the MTL, there exist two ports acting as required interfaces for receiving serialized front end failure models and safety analysis requests. While the *Safety Analysis Service Provider* port receives analysis requests from the FEL asynchronously and thus is the main entry point for analysis, it's also possible to import previously stored front end failure models directly into the MTL. This is a lightweight approach for an intermediate step before implementing a full-fledged front end modeling tool extension. In addition, it facilitates the integration of the MTL into environments, where a lot of legacy failure models are present.

The third port of the MTL is the one, which requires the specific back end analysis tool's API. It provides prepared safety analysis requests for the specific back end analysis tool and receives the analysis results.

3.4.3.2 *Safety Analysis Use Case Execution*

As soon as an asynchronous safety analysis task is triggered from the FEL, the *Front End Communication* component is notified and deserializes the attached front end specific failure model. Subsequently, the control is passed to the *MTL Analysis Task Control* which has access to all necessary components in the MTL for executing the safety analysis task. The next step is the transformation from front end specific failure model to SDM, which is followed by a semantic model validation. The imported SDM may be trans-

formed into a semantically equal model by *SDM Internal Model Transformations* component (see section 3.4.2 for necessities for these transformations).

At this point, the model import to SDM has been completed. The model parts relevant for the specific analysis operation are read from the SDM by the *Failure Model Preprocessor* component and subsequently the analysis request is delegated to the back end analysis tool API, which finally performs the analysis task and returns the results. Those results are stored in the SDM and the *Front End Communication* component notifies the FEL through its *Safety Analysis Service Provider* interface that the results or possibly occurred errors are available.

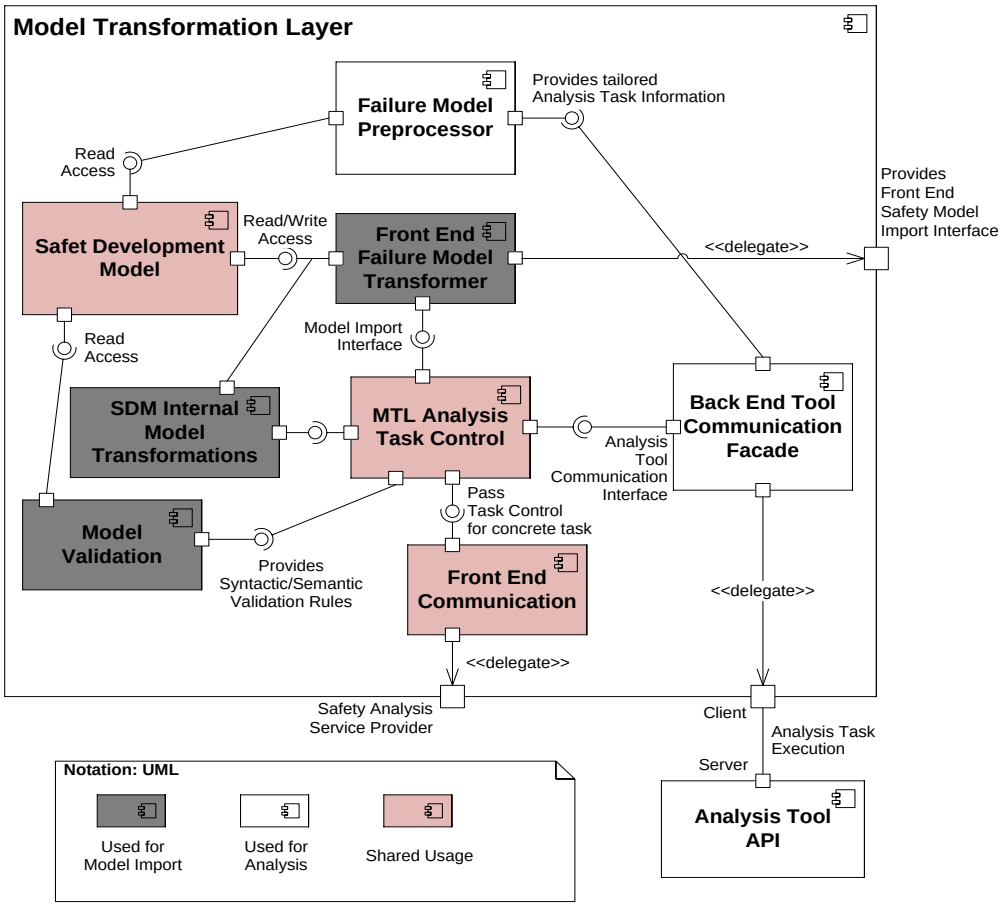


Figure 5 Model Transformation Layer Interaction Structure

3.5 Analysis Back End Integration Challenges

As shown in Figure 1 in section 3.2, there is a variety on safety analysis tools available on the market. In order to understand the differences in the API usage of those tools as well as some typical problems, this section compares the characteristics of the APIs which had already been connected to the MTL before this thesis: the IESE Analysis Back End and FaultTree+.

The most important property of an API is the platform on which it is deployed by its vendor. This has implications on the communication between the MTL and the API. The problems that need to be tackled in this context are:

- Effort for communication implementation

  The IESE API and the MTL are both deployed for the Java Platform. This highly facilitates the integration of this analysis back end, because communication across binaries is easy in Java. On the other hand, the FaultTree+ API is deployed as a dynamic link library (DLL), so extra effort has to be put in the communication between Java and DLLs.

- Error Handling problems

  Especially, when asynchronous analysis requests are sent across platforms, error handling is aggravated on the MTL side. Reasons for that can be for example: a poor documentation of the analysis tool API; the errors are not propagated properly to the service consumer across platforms or they are not signaled at all and end up in an application crash.

Apart from platform issues, the back end tools' APIs differ greatly in usage comfort. This depends primarily on the degree of affinity between the SDM and the meta-model of the specific back end analysis tool. Because of the fact that the IESE analysis back end was developed for the SDM by design, it is able to directly receive failure models from the SDM and analyze them. In contrast, the FaultTree+ API has to be called once for each model element's transmission, which makes the parallelization of analysis tasks difficult and decreases performance because of the communication overhead.

4 System Documentation

With the layered system architecture described in section 3 as foundation, this section deals with the documentation of the developed prototype's implementation accomplished in the course of this thesis. The chosen tools to be connected to the model transformation layer are the front end modeling tool Enterprise Architect and the IESE implementation and FaultTree+ as analysis back ends for performing FTA.

4.1 Model Transformation Layer

This section presents the current development stage of the *safety development model* including its failure meta-models, the applied abstract concepts and its support for analysis. Subsequently, the design of the model transformation layer is documented including some development hints for important tasks.

4.1.1 Safety Development Model

As mentioned in section 3.4.2, the architecture's key component is a safety exchange format that is able to handle multiple safety analysis techniques, even mixed in the same failure model, as well as multiple safety analysis back end tools. These requirements have been satisfied in the course of this thesis by improving the previously implemented *Safe Component Model* (SCM), which was proposed in [11]. The resulting integrated meta-model is called *Safety Development Model (SDM)* and is this section's topic. Section 254.1.1.1 describes the abstract concepts and failure meta-models that have been integrated into the SDM by now. Section 4.1.1.2 explains those parts of the SDM that have relevance, when failure models should be parameterized and analyzed. Finally, section 4.1.1.3 gives development hints that include the used principles for extending the SDM. The current development stage of the SDM implementation is shown in Figure 6.

4.1.1.1 *Failure Meta-Models and Abstract Concepts*

The meta-models of the SCM that were affected by improvements were namely the *generic component model* (GCM) and the *component fault tree model* (CFT). Furthermore, four new meta-models were added: the *safety aspect component model*, the *failure propagation model* (FPM), the *structural propagation model* (SPM) and the *abstract connection model*.

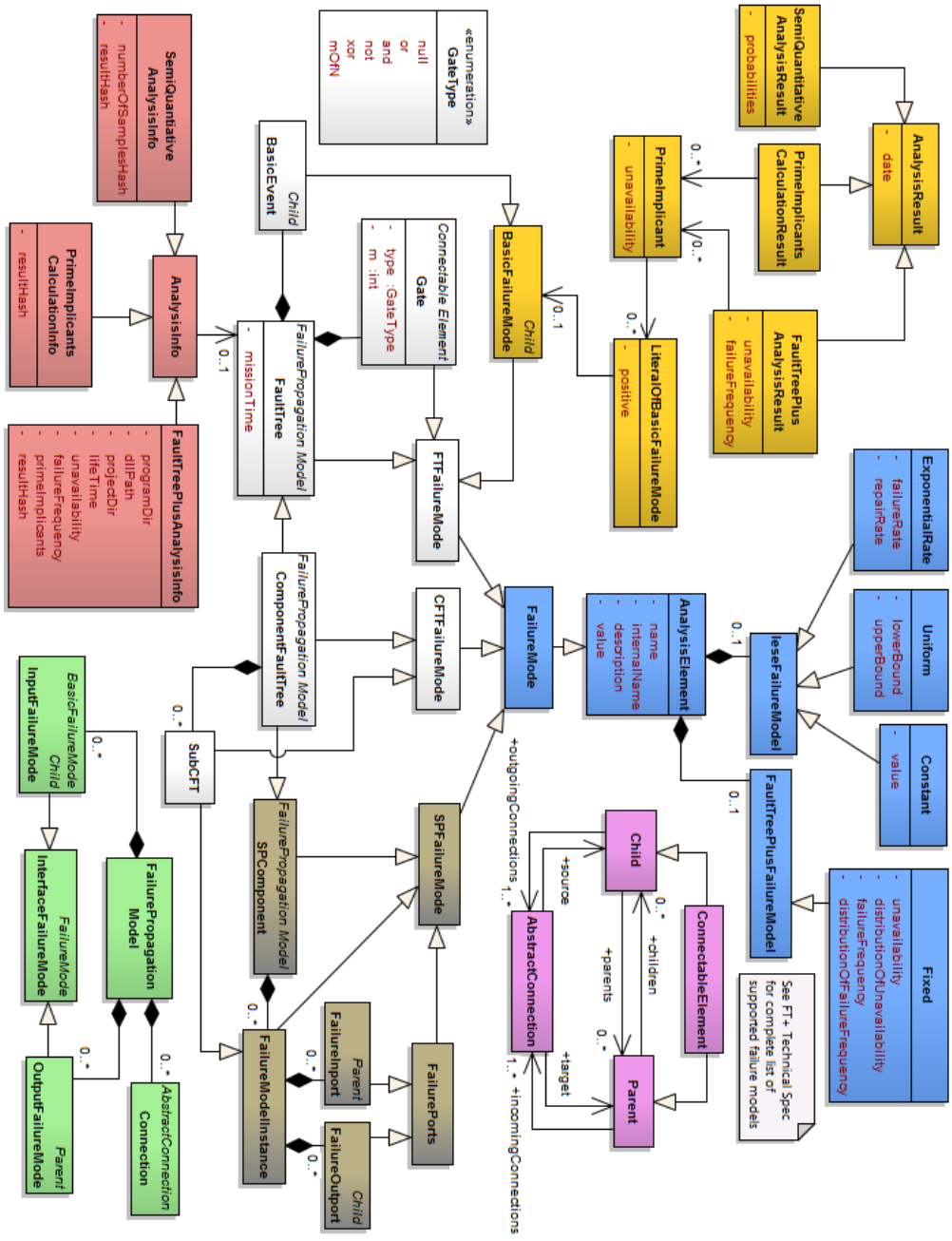


Figure 6 Safety Development Model (SDM) Implementation

- Failure Propagation Model

Because of the fact that a "*Failure View* has been defined to aggregate the commonalities of all integrated kinds of fault trees" ([11] : 69), the notion of a failure view has to be re-thought when multiple safety analysis techniques

should be taken into consideration, because SPES_XT requires an abstraction from specific failure meta-models like CFTs.

Thus, the FPM has been introduced, which represents this abstraction and is shown by the green colored stereotypes in Figure 6. It offers the possibility to associate component models with failure models, independent of specific techniques. In order to achieve this independence, the stereotype *FailurePropagationModel* only exposes the failure model's input and output failure modes, which all specific failure model stereotypes like *FaultTree, ComponentFaultTree* or *SPComponent* get by inheriting from it. Thus, the ports of a component can be associated to interface failure modes without the knowledge of the used technique inside the failure model.

- Safety Aspect Component Model

The GCM proposed in [11] doesn't support an instantiation concept for components. As a consequence, reuse is prohibited in two different ways: On the one hand, without instantiation it's not possible to model more occurrences of one component as part of another component. This is for example needed in a technical system using two identical power units for redundancy. On the other hand, the instantiation concept allows the development of more general and powerful components, when their instances are allowed to reuse only subsets of the provided ports of the component. In order to keep consistency between components and their instances after changes or additions, synchronization mechanisms have to be implemented in both directions.

The resulting overall model in terms of SDM is called *Safety Aspect Component Model*, which includes an instantiation concept as well as the association of the component with an abstract failure model and the component's ports with abstract interface failure modes. Note that the current implementation of SDM does not include component integration due to the fact, that no analysis operations exist by now that are able to take advantage of the associations between components and failure models.

- Abstract Connection Model

The third concept that has been abstracted from in the SDM implementation is the connectivity of model elements shown by the purple colored stereotypes in Figure 6. The idea behind it is that each model element like *Gate*, *BasicEvent* or *InputFailureMode* is restricted by its theory concerning its connectivity. A failure model element can be connected in three ways: Either it has only incoming connections, meaning it's a *Parent*, or it has only output connections, meaning it's a *Child*, or it has both types of connections, then it is called a *ConnectableElement*. These restrictions can be applied for the according stereotypes by extending the respective abstract class. Note that

the stereotype *AbstractConnection* is only once extended, namely from stereotype *Connection* in FPM.

- Structural Propagation Model

As mentioned in section 3.1, SPES_XT also requires the possibility to model heterogeneous failure models that use different techniques both horizontally (at the same hierarchy level) and vertically (at different hierarchy levels), thus a mechanism has to be provided that allows the modeling of failure propagation through instances of failure models that are modeled with different techniques. This approach paves the way for performing safety analysis for whole systems regardless of the used techniques to analyze the associated components. An example for the need of the mentioned heterogeneity can be seen in the industry, when different suppliers of system components use different techniques, but the OEM has to prove safety for the integrated system, though.

The introduction of the SPM, whose contents are shown with the brown colored stereotypes in Figure 6, solves this issue with the creation of stereotype *SPComponent* which extends stereotype *FailurePropagationModel*, so it can be mapped to a component like all other failure meta-models, too. In addition, it can own *FailureModelInstances* that have failure inports and outports. Note that failure inports and outports are instances of the respective interface failure modes from the instantiated failure model. The benefit of this distinction is again the use of different interface failure mode subsets in different contexts.

Since the SPM encapsulates the instantiation concept for failure models, it can be reused in other failure meta-models. This is for example implemented in the SDM for CFTs. For this reason, in Figure 6, stereotype *ComponentFaultTree* inherits from stereotype *SPComponent* and stereotype *SubCFT* inherits from stereotype *FailureModelInstance*.

- CFT and FT Models

The SDM implementation for FTs and CFTs is shown in Figure 6 with the white colored stereotypes.

Only the specific elements *BasicEvent*, *Gate* and *GateType* for *FaultTree* and *SubCFT* for *ComponentFaultTree* had to be modeled, because *ComponentFaultTree* extends *FaultTree* and therefore inherits all FT model elements. It's important to note that all other concepts like interface abstraction, instantiation and connectivity restrictions could be integrated by only extending the respective abstract classes *FailurePropagationModel*, *SPComponent* and *Child, Parent* or *ConnectableElement.* This highly facilitates the integra-

tion of new failure meta-models into the SDM in the future, because the applied concepts are already available in an encapsulated way.

4.1.1.2 *Analysis Support*

The SDM parts that have been described in the preceding section included abstract concepts required by SPES_XT and the actual failure meta-models for CFTs and FTs. This section deals with the SDM parts that are necessary for the support of different safety analysis operations and analysis back end tools, namely the parameterization of failure models, the consideration of back end tool specific properties and the representation of analysis results.

- Model Parameterization

The concept for the parameterization of any failure model element is demonstrated in Figure 6 by the blue colored stereotypes. When an arbitrary model element should be parameterized it has at least to extend the most general stereotype *AnalysisElement*. An *AnalysisElement* is currently capable of having a parameter model for both IESE back end (*IeseFailureModel*) and FaultTree+ back end (*FaultTreePlusFailureModel)*. A parameter model is defined by a failure distribution. Figure 6 shows *ExponentialRate*, *Uniform* and *Constant* distributions for IESE back end, but only *Fixed* distribution for FaultTree+ back end, although all FaultTree+ failure distributions are supported.

Note that normally, failure distributions are only assigned to basic failure modes in FTA, but in theory, they could be assigned to any element extending *AnalysisElement*, e.g. *FaultTree*. This issue is an example of the fact that the SDM is designed for being flexible concerning parameterization. The price for this flexibility has to be paid by the MTL developers, who have to take care that such parameterizations like in the example don't happen.

The introduced abstract classes *FailureMode* as well as its subclasses *FTFailureMode*, *CFTFailureMode* and *SPFailureMode* don't have a specific responsibility at the current stage of development, but they were included with respect to the possible need in the future to distinguish between the elements of different failure meta-models.

- Back End Tool Specific Properties

Before a parameterized SDM failure model can be handled to a back end tool for analysis execution, it has to be packaged together with attributes that describe back end tool specific issues. Such a package is represented in Figure 6 by the abstract stereotype *AnalysisInfo*, which is colored in red. The stereotype for a specific analysis operation with a specific back end tool has to be extended from *AnalysisInfo*.

This has been done for the analysis with FaultTree+, including some parameters to enable the connection to the FaultTree+ application (*programDir* attribute) and its API DLL (*dllPath* attribute) as well as the attributes *unavailability, failureFrequency*, *primeImplicants* and *resultHash*, which are typed data structures that include the results after the analysis execution.

The classes *PrimeImplicantsCalculationInfo* and *SemiQuantitativeAnalysisInfo* represent the supported analysis operations for the IESE back end. Note that quantitative analysis is also supported, but that is equal to a semi-quantitative analysis, where all basic failure modes have a constant failure distribution.

Concerning FTA, the elements, for which quantitative and qualitative analysis can be performed, are output failure modes and gates. The attribute *resultHash* has a special responsibility for all *AnalysisInfo* extensions, because it maps analysis results to output failure modes and gates, which both are extended from *FailureMode*, so *resultHash* attribute's type *Map<FailureMode, AnalysisResult>* is again an example for the flexibility of SDM.

- Analysis Result Representation

As described above, analysis results are mapped to output failure modes and gates within SDM. The stereotype representing the result of a specific analysis operation is *AnalysisResult*. All yellow colored elements in Figure 6 are related to the representation of analysis results. Analogous to *AnalysisInfo*, *AnalysisResult* has to be extended for each analysis operation. The extended class is responsible for the storage of the operation specific results, which are finally read by the MTL's front end adapters and sent to the front end application's for the presentation to the user.

4.1.1.3 *Development Hints*

This section describes the process for adding a new analysis technique to the SDM as well as the integration of a new analysis operation for an analysis back end tool.

- Adding a new analysis technique to the SDM

    1. Create an EMF ECore Model in Eclipse for the new technique

    2. Model stereotypes for the specific elements for the technique (like the white stereotypes in Figure 6) and make sure that the stereotype representing the technique like *FaultTree* extends stereotype *FailurePropgationModel*.

3. Define the connectivity for each stereotype that is able to be connected like *BasicEvent* or *Gate*. This is done by extending from *Child*, *Parent* or *ConnectableElement* stereotypes (purple stereotypes in Figure 6).

4. If the new technique should support instantiation as it's the case with CFTs, add a stereotype with name *Sub<TechniqueName>* extending stereotype *FailureModelInstance* from SPM and make sure that the technique stereotype like *ComponentFaultTree* also extends *SPComponent.*

- Adding a new analysis operation to the SDM

  1. Extend *AnalysisElement* with a stereotype for the back end tool (like *IESEFailureModel*) which serves as base stereotype for all parameter sets that are needed for this back end tool (see blue colored stereotypes in Figure 6). In FTA, these are the failure distributions for the basic failure modes.

  2. Extend *AnalysisResult* for the new analysis operation and include attributes for each result value

  3. Extend *AnalysisInfo* for the new analysis operation and create at least a *resultHash* attribute that has the type *Map<FailureMode, AnalysisResultSpecialization>*, while *AnalysisResultSpecialization* is the stereotype defined in 2.

  4. Generate Java classes from the EMF ECore Model and use them in the MTL development as it will be described in section 4.1.2.

4.1.2 MTL Design Documentation

This section first gives some general information of the MTL implementation structure. Section 4.1.2.1 explains the design decisions that have been taken during the MTL development process. Section 4.1.2.2 provides a step-by-step list of how the MTL is extended and finally, section 4.1.2.3 presents an example that shows which MTL implementation units are in action at what time during semi-quantitative analysis of a CFT

Due to the fact that the implemented prototype in this thesis only deals with two analysis back ends and FTA as analysis technique, the rather generic structure for the MTL presented in section 3.4.2 could be concretized considerably. A major influence on the concretized MTL structure has been the existing front end implementation for MagicDraw as well as the implementation of the IESE analysis back end. Although both implementations were aimed to match the *Safe Component Model* (SCM) proposed in [11], the

overall structure could be reused. In order to be able to join the EA adapter consistently with the existing MagicDraw adapter, the actual MTL implementation presented in this section has a structural difference to the proposed MTL structure from section 3.4.2, namely that the packages in the MTL are not separated by front end adapter, safety exchange format and back end adapter, but rather by the analysis techniques.

The resulting implementation structure is visualized by means of a UML package diagram in Figure 7. The packages colored in blue represent packages that were already implemented at the start of this thesis. The reflection of the model change from SCM to SDM in those packages was implemented by some colleagues from Fraunhofer IESE. All used arrows in the diagram are UML usage relations and when package names are addressed in this section, the prefix “de.fhg.iese” is left out because of the fact that all MTL implementation packages reside in “de.fhg.iese” package.

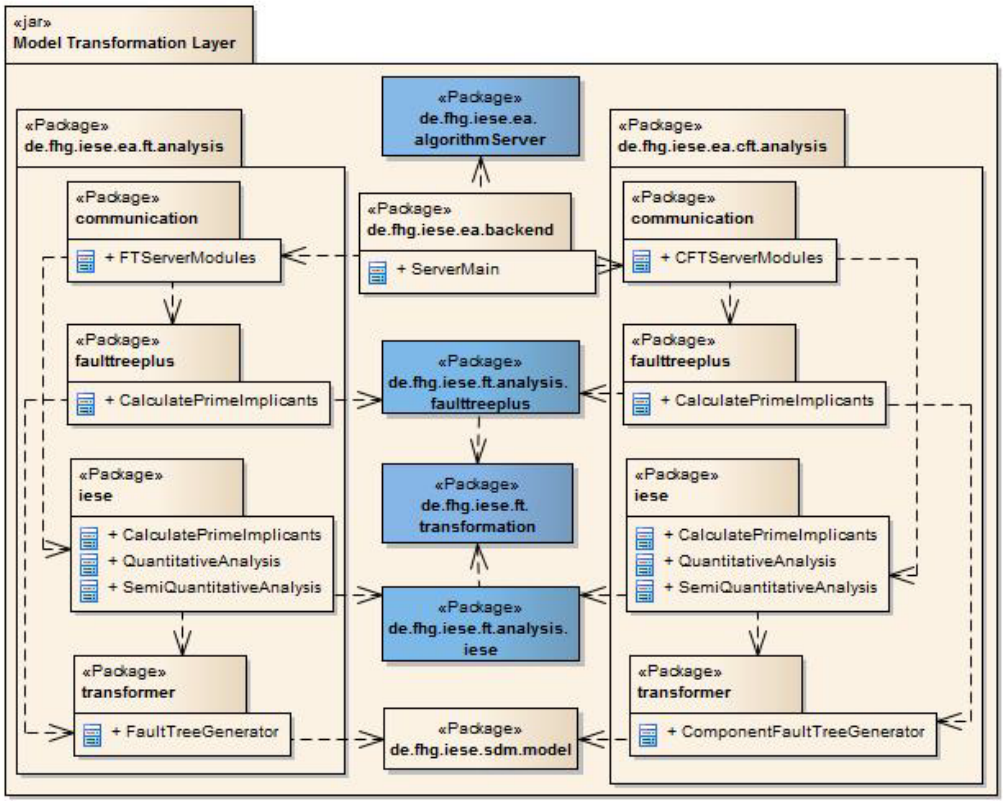


Figure 7 MTL Implementation Structure for FTA with EA

4.1.2.1 *Design Decisions*

By virtue of each analysis technique being encapsulated in a separate package and that it should be possible to deploy the back end with different sets of techniques, the package *ea.backend* has been created. Its purpose is to provide the main class for starting the analysis server and specifying the techniques that should be included.

- Client-Server Communication Library

The so-called server modules for each technique are stored in a separate class like *FTServerModules*. They are part of the communication library *ea.algorithmServer* which allows comfortable TCP message exchange between .NET and Java. The library's characteristic feature is that messages can be exchanged between named modules that additionally specify the behavior for the reaction to an incoming data message in the *run(AsMessage msg)* method of the class *AsModule*. The usage is comfortable, because the return value of the *run* method is automatically sent to the matching module in .NET. Note that a pair of modules, one in .NET and one in Java, is identified as one communication channel, if they have the identical identifier string on both sides.

An incoming analysis request from the FEL includes a XML serialization of the failure model, which should be analyzed. The information, which analysis operation should be performed with which analysis back end, is encoded in the server modules themselves, e.g. when prime implicants should be calculated for CFTs with FaultTree+, the analysis request is triggered inside the .NET module named "cft_primeimplicants_ft+_mod" and the message automatically arrives in the matching Java module with the same name as parameter.

- dom4j XML Library

The included XML serialization of the failure model needs to be parsed, for which the library dom4j [17] has been made use of, because the native mechanisms for parsing XML in Java are not very comfortable. dom4j also provides mechanisms to use the XPath language [18], which allows efficient navigation through XML documents in an intuitive way and therefore is an advantage in terms of code readability and code maintenance. There is made extensive use of this feature in the failure model transformers which are also described in this section.

- Concurrency in MTL

Each analysis operation is encapsulated in a class like *CalculatePrimeImplicants* in *ea.ft.analysis.*iese package, which inherits from *java.lang.Thread*,

because this paves the way for parallel execution of several analysis operations in the long run. One could imagine this scenario in a distributed environment where the IESE algorithm server assembly is provided on a remote server for simultaneous usage of several customers.

In order to synchronize the main thread with the analysis operation thread at the analysis end, the class *LinkedBlockingQueue* from *java.util.concurrent* package has been used. One can put function calls in the queue, which automatically executes them and puts their return values in the queue, when the function has finished executing. The key concept is that *LinkedBlockingQueue.take()* function suspends the calling thread as long as no return value is available.

- Failure Model Transformation

The transformation of failure models delivered from the front end into its SDM representation is implemented in the transformer packages. As mentioned in section 3.4.2, the SDM is implemented by means of the EMF framework, which builds usable java packages from the declaratively created models. The package for the SDM implementation is sdm.model, whose classes are accessed by the failure model transformers extensively. During the transformation, hash maps are created for a mapping between identical model elements in both representations, which increases the transformation efficiency, because each model element is only transformed once. This is useful because of the fact that multiple instantiation of failure models in different hierarchy levels is possible and would otherwise lead to redundancy. The mapping keys are EMF objects representing the SDM elements which are mapped to “Globally Unique Identifiers” (GUID) [19] assigned to model elements in EA. Note that GUIDs are the used mechanism in EA for uniquely identifying model elements.

- Back End Adapters

As depicted in Figure 7, the analysis operation classes can make use of the back end adapter packages, which are *ft.analysis.faulttreeplus* and *ft.analysis.iese* in the developed prototype. They provide classes for each analysis operation and encapsulate the behavior for the analysis with a specific back end tool. In addition, their interfaces are similar, which makes their usage straightforward for developers.

The back end adapter packages themselves make use of the *ft.transformation* package. It provides the functionality to perform internal model transformations within the SDM, e.g. a flattening of CFTs to FTs or a reduction of CFTs. In addition, it performs semantic and syntactic model validations during transformation like Boolean loop detection. Above all, the flat-

tening is important with respect to FTA, because the analysis back ends only support analysis for flat fault trees.

4.1.2.2 *Development Hints*

The development tasks concerning the MTL, which are likely to be of interest in the future, are presented in the following as step-by-step recipes. This includes the addition of a new analysis technique as well as a new analysis operation for an existing technique.

- Adding a new analysis technique

    1. Create a package for the analysis technique like *ea.ft.analysis*

    2. Create package *transformer* in it and implement the transformation class for the failure meta-model transformation from its serialization to the SDM model

    3. Create package *communication* with a *<FailureModel>ServerModules* class, which will consist of the communication modules. Subsequently, register this class in *ServerMain* class inside package *ea.backend* in order to make the created modules accessible in the deployed assembly.

    4. Start adding analysis operations

- Adding a new safety analysis operation for an existing technique

    1. Create a new class for the analysis operation under the respective analysis back end package, for which this operation should be available like *CalculatePrimeImplicants* in *ea.ft.analysis.iese* package

    2. Create a new *AsModule* instance in *communication* package of the respective analysis technique and make use of the class created in 1.

4.1.2.3 *CFT Analysis Example*

In order to provide a continuous example, which visualizes the mapping between the implementation units and the tasks they accomplish, the semi-quantitative analysis use case with CFTs has been chosen. In Figure 8 that part of the execution taking place in the MTL is shown, while its execution in the FEL will be covered in section 4.2.2.4.

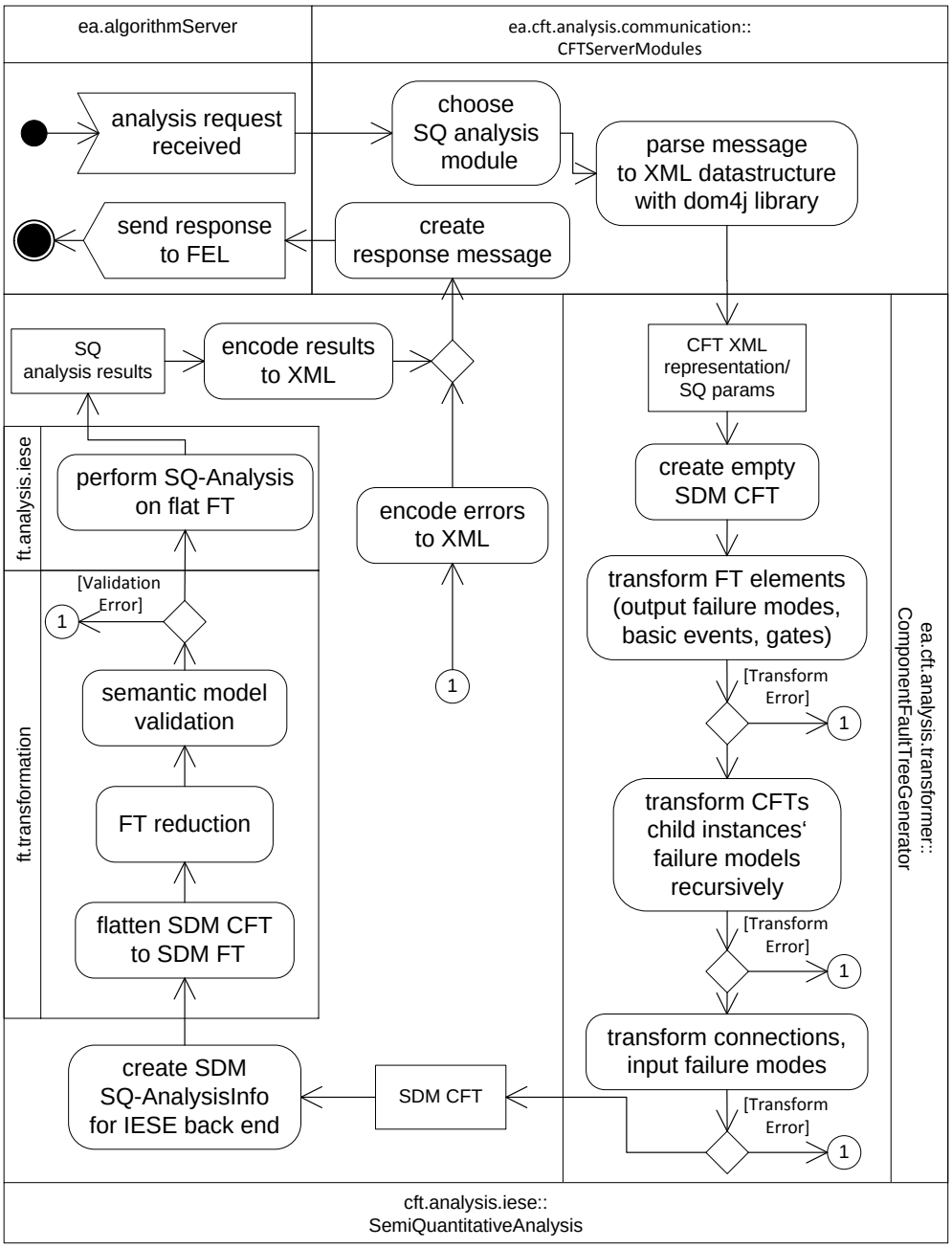


Figure 8 CFT Semi-quantitative Analysis Example in MTL

Initially, analysis request is received by the server module for semi-quantitative analysis with the IESE back end. After the message has been parsed, the CFT XML representation is transformed step by step into the SDM representation. If any error occurs during the transformation, it is stopped and the error is propagated back to the FEL, where the user is noti-

fied of it. Otherwise, the SDM *SemiQuantitativeAnalysisInfo* is prepared. Subsequently, the SDM CFT representation is flattened to a FT and validated semantically. If the validation succeeds, the FT is analyzed by the IESE back end algorithm for semi-quantitative analysis. Its results are encoded to XML and packaged in a message that is sent back to the FEL by the provided communication library.

4.2 Enterprise Architect Front End

This section starts with a description of the EA tool architecture and subsequently describes the FEL's implementation for EA. This includes the developer documentation for both the EA UML profiles that have been created according to the meta-models from the SDM and the add-in that incorporates the additional functionality used for enriching the capabilities of the EA profiles.

In order to develop extensions for EA, one has to be familiar with its internal, layered architecture, which is shown in Figure 9.

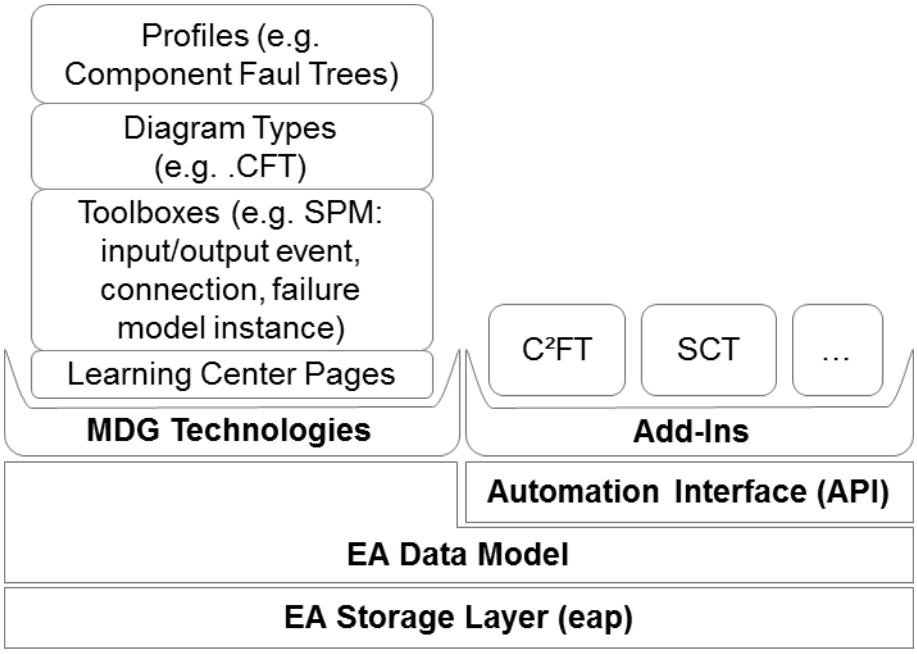


Figure 9 Enterprise Architect Tool Architecture

The key component of this architecture is the *EA Data Model*, which can be accessed by two mechanisms:

On the one hand, the model can be extended in a declarative way by *Model Driven Generation (MDG) technologies*, which are EA means to implement

UML profiles. In addition, several other artifacts like diagram types and their toolboxes or user help pages can be incorporated into a MDG technology. On the other hand, one can make use of so-called add-ins, which offer programmatical access to the *EA Data Model* through the *Automation Interface*. The add-ins have to be written in the .NET platform and are used for enriching MDG technologies with additional functionality.

The *EA Data Model* is typically stored in EA project (eap) files. They are based on the Microsoft Jet 4.0 database engine and because of that, they are equal to the MS Access '97 mdb format [20].

Note that many of the EA-specific issues and terms described in this section are described in more detail in [21]. In addition, there exist two extremely useful e-books which originated from the EA user community and give advanced information about the development of EA extensions [22] and the internal structure of the *EA Data Model* [23].

4.2.1 UML Profiles in EA

4.2.1.1 *MDG Technology Creation*

This section shows the typical process of the MDG technology creation. In order to visualize the artifacts created during this process, the central model of the SPES modeling approach is used, namely the component model. The steps for creating a new MDG technology are:

1. UML Profile Definition
2. Toolbox Profile Definition
3. Diagram Profile Definition

The first step in the process is the definition of a UML profile in EA. In general, profiles are UML's extension mechanism for expressing domain-specific meta-models. A profile is defined by creating new stereotypes, constraints and tagged values for UML meta-classes. The EA component model profile is shown in Figure 10.

- Meta-classes and tagged values

In EA, meta-class elements have the stereotype <<*metaclass*>> and stereotype elements are marked with the «» image next to the element name. Both of them are modeled as classes within EA. Connections with black-filled arrowhead symbolize meta-class extension, while connections with white-filled arrowhead symbolize stereotype extension.

Tagged values can be modeled by one of two mechanisms, either as class attributes in a stereotype or as UML uni-directional associations between stereotypes. It proved best to use the association mechanism for tagged values referencing other stereotypes, while tagged values of simple types like integer are defined as attribute. Note that tagged values can only be inherited from stereotypes which extend a meta-class.

- Special EA attributes

There exist some special attributes for meta-classes and stereotypes in EA, whose names have to start with an underscore. These are are used for defining the appearance of stereotypes in diagrams (*_image, sizeX, sizeY, _lineStyle)* and rules how EA treats stereotype elements internally (*_defaultDiagramType, _metatype, _instanceMode, _instanceType*). The most powerful of them is the *_image*-attribute, because it allows defining an arbitrary stereotype appearance through a XML-structure with the use of a simple scripting language called *ShapeScript*. Examples for shape-scripted

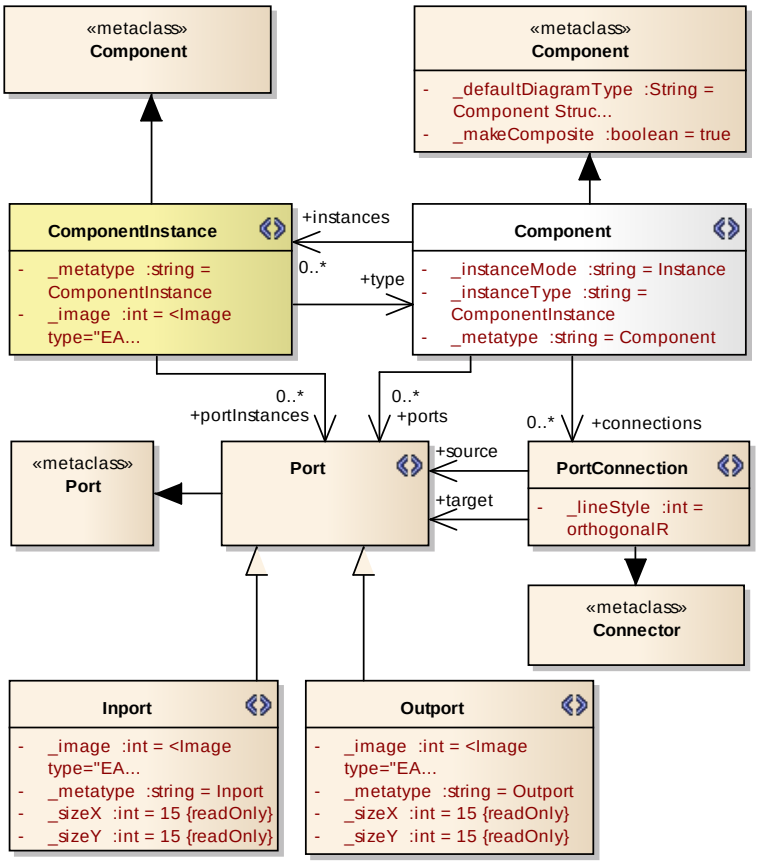


Figure 10 EA Component Model Profile

appearance are given in Figure 11 for the stereotypes *Inport* and *Outport*. Note that MDG technologies can also include images that are accessible from shape scripts. Another important attribute is the *_defaultDiagramType*-attribute, because it defines the name of custom diagram type in the form *<DiagramProfileName>::<DiagramStereotypeName>*, which is automatically created and attached to the stereotype, when it's dragged from a toolbox.

- Diagram Types

The second step in the MDG technology creation is the definition of custom diagram types. They are modeled in EA by diagram profiles and toolbox profiles, where from special meta-classes and attributes can be made use of. This is shown exemplarily for the component internal structure diagram in Figure 11. Normally, diagram and toolbox profiles have to be modeled in separate diagrams, but for space issues, they are shown in one image.

A toolbox profile consists of at least one toolbox page specifying attributes for each stereotype that should be available for modeling. By specifying a stereotype extending the meta-class *ToolBoxItemImage*, custom images can be used to further describe the stereotype's intentional purpose directly in the toolbox.

Diagram profiles usually consist of only one stereotype (*Component Internal Structure)* extending an EA-specific meta-class representing an UML diagram (*Diagram_CompositeStructure*). This is in most cases the UML component structure diagram in this thesis, because it serves best for modeling architectural structures due to its built-in support for hierarchies and modularization, which are of main interest in the SPES_XT meta-models. The most important attribute of each meta-class *Diagram_<UMLDiagram>* is *toolbox*, whose value is the name of that EA-diagram, where the toolbox profile is defined in, in Figure 11 its value is "InternalView".

The creation of profile packages is facilitated by the usage of a wizard so-called "Profile Helpers", which provides dialogs that simplify the creation of stereotypes, toolboxes and diagrams by listing all possible stereotype attributes including the EA-specific ones and their possible values.

Finally, the MDG technology has to be put together. During this task, all relevant profiles, diagrams and toolboxes and optionally other resources (see [21] for more details) are compiled into a single XML file, which can be imported manually in other EA installations or, as it is done within this thesis, directly embedded into an EA add-in.

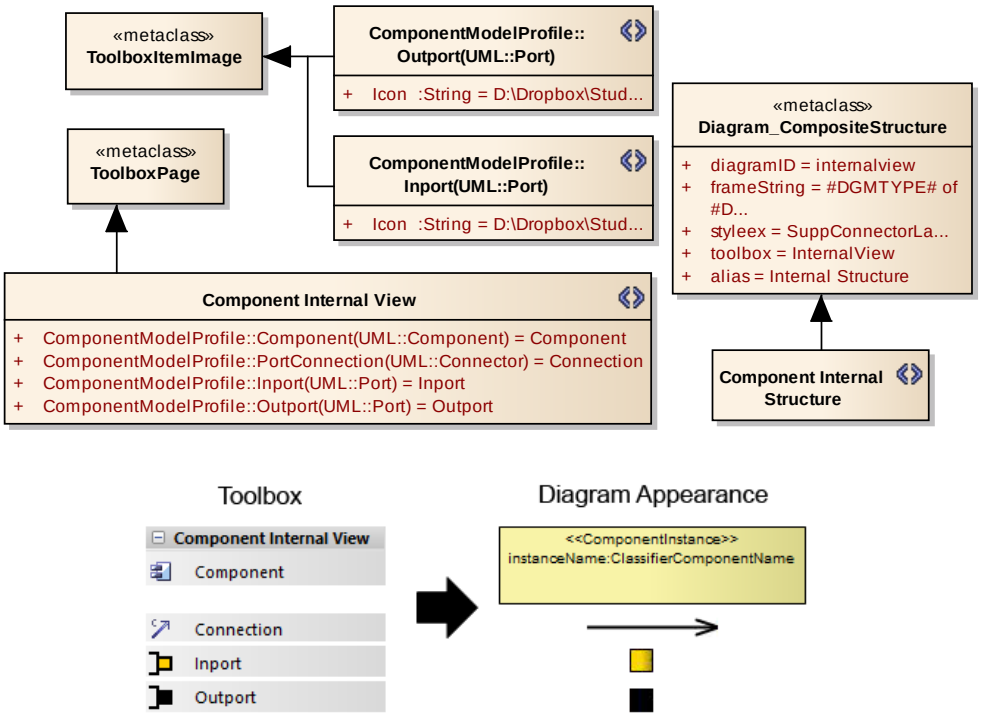


Figure 11 Custom Diagram Type in EA

4.2.1.2 *C²FT EA Profiles*

The safety analysis technique considered for this thesis is fault tree analysis. In order to support the modeling of FTs, CFTs and C²FTs, three profiles had to be developed. In addition, a profile has been created for SPM. This won't be covered in this section, because the used EA-related concepts are similar to the other profiles. For the interested reader, the SPM profile can be found in Figure 23 in appendix section 6.1.

The EA component model profile, which has already been shown during the description of MDG creation, was created according to the safety aspect component meta-model described in section 4.1.1. Some of the modeled tagged values are actually superfluous and modeled only for understandability, because they are accessible through EA's automation interface by default. These include *ComponentInstance::type* and *PortConnection::source/target.* Note that the tagged value *Component::instances* stores references to the instances of other components, that it's owning and not to its own instances.

Although the UML provides default support for instantiation through the meta-classes *InstanceSpecification* and *Property*, this was not practicable in the component profile, because of the fact, that an interactive navigability

through the component model is required and both of them are not composite, i.e. they aren't containers for further elements. In contrast, the EA-representation of the meta-class *Component* provides the needed functionality by its *_makeComposite*-attribute and it has therefore been chosen for both stereotypes *Component* and *ComponentInstance*.

The second created EA profile is that for the CFT meta-model, which is presented in Figure 12. It supports modeling of both FTs and CFTs.

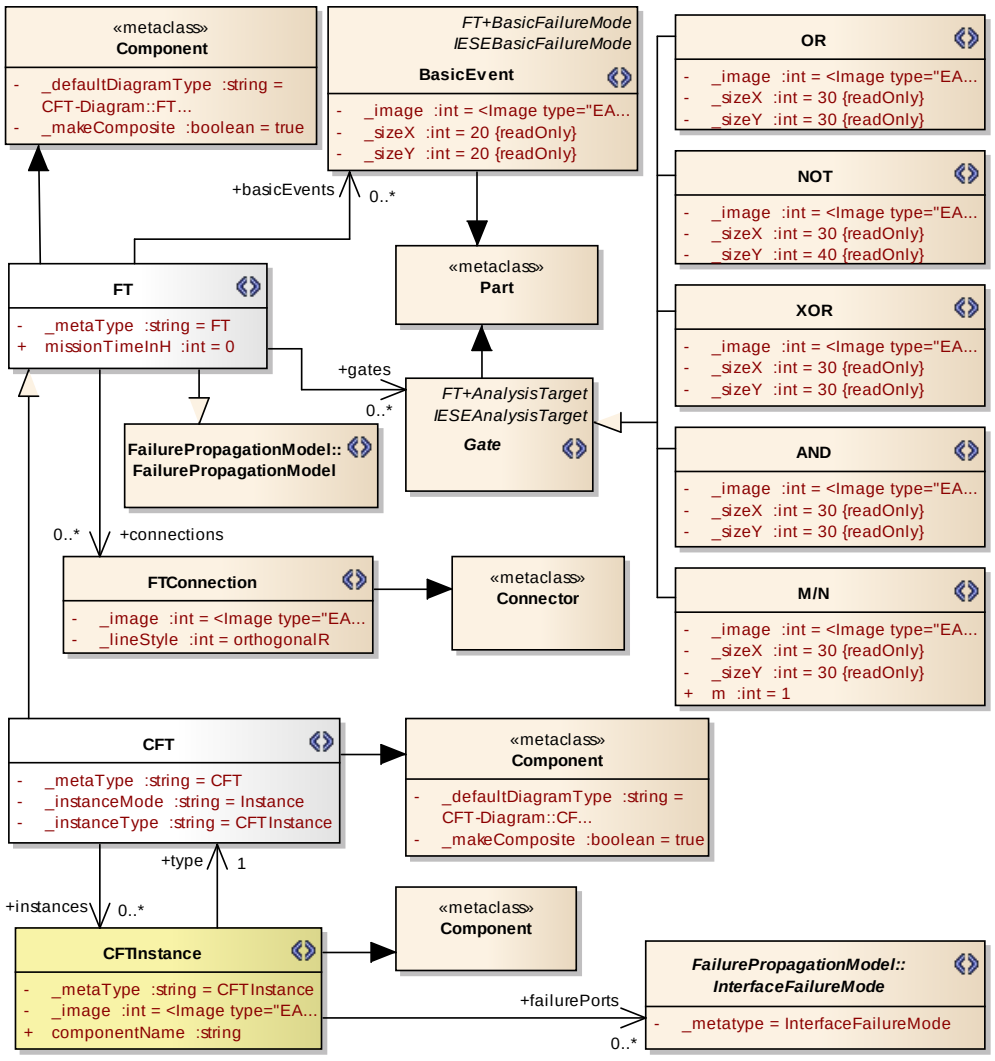


Figure 12 EA CFT Profile

For stereotypes *Gate* and *BasicEvent* the meta-class *Part* has been chosen, because they need not be composite and are always instantiated inside a parent component, which is exactly the natural notion of *Part*. *CFT* stereo-

type inherits from *FT* stereotype and adds instantiation capabilities to it, i.e. CFT stereotypes are capable of owning instances of other CFTs, while FTs are not. The CFT part of the profile is similar to the EA component profile. Note that both stereotypes *CFT* and *FT* extend the meta-class *Component* with one difference: The *_defaultDiagramType*-attribute stores references to a different custom diagram in each case, which goes back to the different sets of elements provided in each toolbox (see Figure 21 in appendix for all other created toolboxes).

Stereotypes *FT* and *CFT* inherit from *FailurePropagationModel* stereotype, which is defined in the "Failure Propagation Model" profile shown in Figure 22 in the appendix. This profile has been created to automatically supply every new added meta-model of a safety analysis technique with interface failure modes and the ability to map functional ports to them, just by extending from *FailurePropagationModel* stereotype. In addition, the mapping between component and failure model is also formalized by this extension.

Note that the *failure propagation model* is designed to keep flexibility for the specific failure models as much as possible. The idea behind is to provide concepts for them like the ability to have interface failure modes, but they aren't forced to use them. An example for this issue can be seen with *FTs.* They are restricted by the fault tree theory, which defines that fault trees are only allowed to have basic events and output failure modes. This restriction is reflected in the toolbox definition for the *FT-Diagram*, where the stereotype *InputFailureMode* is not provided.

4.2.1.3 *Back End Analysis Profiles*

So far, the defined profiles provide the ability to create models of FTs, CFTs, C²FTs and SPM, but an additional requirement is to perform some kind of analysis on these models. Depending on the sort of analysis, some of the modeled elements require being parameterized. This section deals with the profiles that provide the mechanisms to assign parameters to model elements.

In order to provide parameterization for stereotypes considering different safety analysis operations and different analysis back end tools, separate profiles for each tool have been provided. The drivers for separating modeling profiles from analysis profiles have been:

Support of multiple parameterizations for one stereotype by multiple inheritance

This means that a stereotype like *CFTProfile::BasicEvent* is not parameterized by default. Instead, it *can* inherit from all according back end tool parameterization stereotypes to apply their parameterizations. These

are obviously those for basic failure modes, *IESEBasicFailureMode* shown in Figure 13 and *FT+BasicFailuremode* shown in the FaultTree+ analysis profile in Figure 24 in appendix.

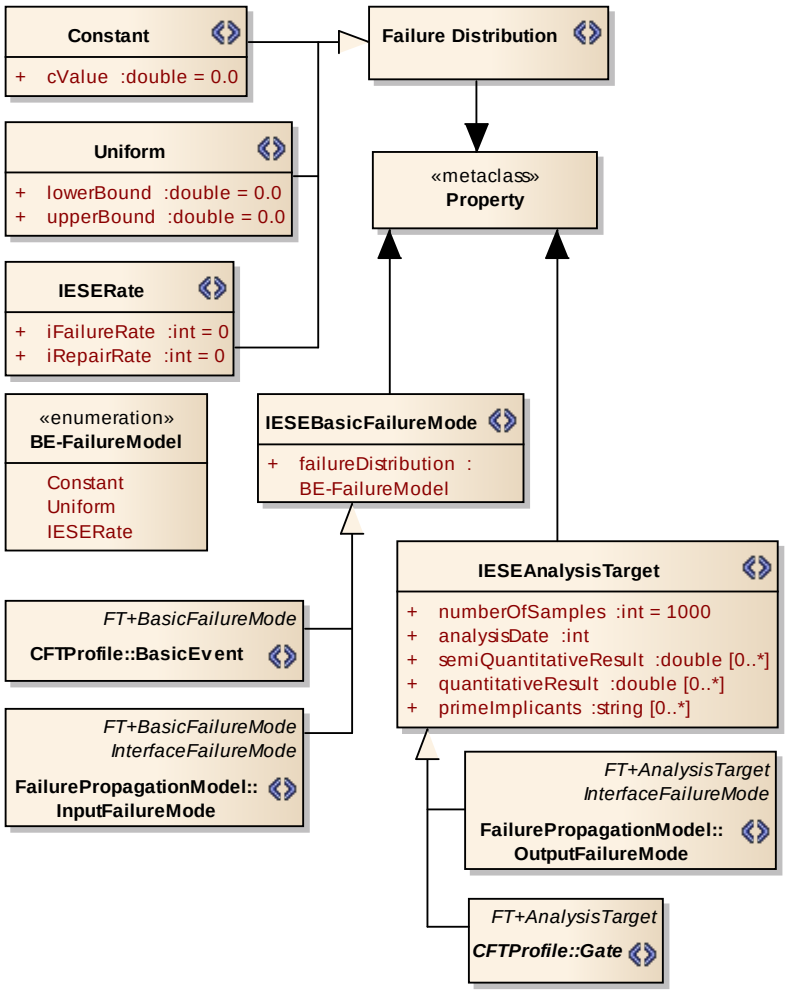


Figure 13 EA IESE Analysis Profile

Separation of concerns improves understandability and therefore simplifies profile maintenance

Separation of concerns is a general principle that improves understandability, because only a limited amount of information that is additionally cohesive is presented at once. Applied to profiles, this means that it would be hard to identify all analysis related contents, if the profiles were mixed up. Furthermore, the stereotypes of the failure metamodel that are used for modeling like *BasicEvent* or *Gate* from *CFTProfile* don't need to be changed *directly*, when for example new failure dis-

tributions are added to the analysis profile. Instead, they receive the additional distributions automatically just by the existing inheritance.

Modeling profiles should be able to be deployed without analysis capabilities

Possibly, the tool will be used in some contexts only for the creation of failure models. Thus, the separation of analysis profiles from modeling profiles minimizes the effort for removing analysis capabilities from modeling elements, namely just the extension relations need to be removed.

Exemplarily, the analysis profile for the IESE analysis back end is shown in Figure 13 (see Figure 24 in appendix for FaultTree+ analysis profile). It is capable of parameterizing the basic failure modes of CFTs and FTs with the failure distributions *IESERate*, which is a exponential distribution, *Uniform*, which is a uniform distribution and *Constant*, which represents a constant failure probability. The parameterization is accomplished by inheriting from the respective analysis stereotype owning the parameters as tagged values. For example, *BasicEvent* from CFTProfile gets a failure distribution parameter by inheriting from *IESEBasicFailureMode*. Note that *IESEAnalysisTarget* also has tagged values for the storage of results of different safety analysis tasks.

4.2.2 EA C²FT Add-In

This section presents the approach for extending Enterprise Architect programmatically as well as the concrete contents of the developed add-in for C²FTs. In addition, it explains the design decisions that need to be understood, when the add-in should be maintained or extended.

4.2.2.1 *General EA Add-In Approach*

The EA Automation Interface, which has already been shown in Figure 9 in section 4.2, consists of an object model (see Figure 25 in appendix) that allows programmatical access to the *EA Data Model* and the registration for EA events such as user interaction with the modeling tool or model changes. This is realized by implementing a so-called add-in for the .NET framework, i.e. a dynamic linked library (DLL) has to be created that references the EA Automation Interface, which is shipped with EA as a DLL called “Interop.EA.dll”. In order to be automatically detected by the EA application, two tasks have to be accomplished:

1. Add a registry key to the Windows Registry under the registry path *HKEY_CURRENT_USER/Software/Sparx Systems/EAAddins* with name <AssemblyName> and value <AssemblyName>. <MainClassNamepaceName>.<MainClassName>.

2. Register the add-in assembly to the .NET framework's global assembly list by invoking the command

   ```
   <RegAsmPath>/RegAsm.exe <AssemblyPath>/<AssemblyName> /codebase
   ```

   on the command line[1].

The specified main class in step one has a special responsibility: Callbacks for EA events have to be defined there. They represent the complete interaction interface for the communication between the EA application and the add-in and can be separated in the following categories:

MDG Events allow it for example to embed MDG technologies directly into the DLL, which is the chosen approach in the C²FT add-in.

- *Add-In Events* notify the add-in for example when custom menu items of the add-in are clicked or when the EA application is closed.
- *Context Item Broadcast Events* notify the add-in when the user changes or double-clicks the selected item.
- *Pre Creation and Post Creation Broadcast Events* notify the add-in before and after an element is created.
- *Pre Deletion Broadcast Events* notify the add-in before an element is deleted.

All events have in common that the notification includes context information by means of the callback parameters so that the add-in can react appropriately to the events. The passed parameters have almost in all cases types, which are defined in the object model.

The object model's central class representing the main entrance point to the "EA Data Model" is *Repository*. It provides properties and methods that allow the navigation through the model as well as the creation, deletion and alteration of model elements and diagrams. Model elements like stereotypes defined in profiles are represented by the class *Element* and can only be created inside packages (class *Package*). *Element* has some collections as properties, which implement the *Collection*-Interface. These are collections for

[1] RegAsm.exe isdeployed with the .NET framework and can usually be found under `<WindowsDirectory>\Microsoft.NET\Framework\<FrameworkVersion>\RegAsm.exe`

the associated tagged values and the elements that are owned by the parent element within the element hierarchy. Note that there are two collections for representing owned elements, namely *Element.Elements* and *Element.EmbeddedElements*. EAs documentation doesn't explain their difference clearly, but a test showed that *Element.EmbeddedElements* stores the same elements as *Element.Elements* plus all owned ports. Packages can contain other packages as well as diagrams (class *Diagram*). Connectors may also be stereotyped elements, but they have their own representing class *Connector*.

The object model also reflects the distinction between the actual model elements (classes *Element* and *Connector*) and their representation in diagrams (classes *DiagramObject* and *DiagramLink*, respectively).

4.2.2.2 *C²FT Add-in Design*

The structure of the developed EA add-in for C²FTs is presented by means of a package diagram in Figure 14, where the majority of elements from the FEL architecture proposed in section 3.3.2 re-emerge. Thus, the explanation of the individual classes' responsibilities are either clear by their names or can be taken from the general explanations from the functional decomposition in section 3.3.2, where examples for these responsibilities are provided, too.

The remainder of the section describes the most important issues of the C²FT add-in design that Figure 14 doesn't show. These are namely the rationale for the used programming paradigm, the supposed technique for extending profiles in the add-in, the used UI dialogs and how common behavior is treated in the add-in.

- Used programming paradigm

EA add-ins are supposed to only consist of behavior that reacts to events, which has two causes: Firstly, the "EA Data Model", which contains the whole information about the models from a currently opened .eap file (=model state), is stored internally in the EA application with a limited access through the EA API. Secondly, the broadcast events that notify add-ins of changes to the model state provide references to the relevant elements of the "EA Data Model" through their parameters.

As a consequence, there is no need to store the model state in the add-in implying that the add-in consequently hasn't any real program state at all. Thus, a rather functional approach has been chosen for the add-in development, i.e. the classes shown in Figure 14 contain mostly functionally related static methods rather than being actual classes in the sense of object orien-

tation (OO). Even though, it is possible to use OO mechanisms like inheritance to abstract from common functional behavior.

- Extending declarative profiles in the add-in

As mentioned in section 4.2.1, MDG technologies containing the declarative profile and diagram definitions can be embedded into the add-in assembly as so-called resources. They can be loaded into the EA application with the add-in event *EA_OnInitializeTechnologies()*. Note that after each technology change the add-in has to be re-built in order to reflect the changes.

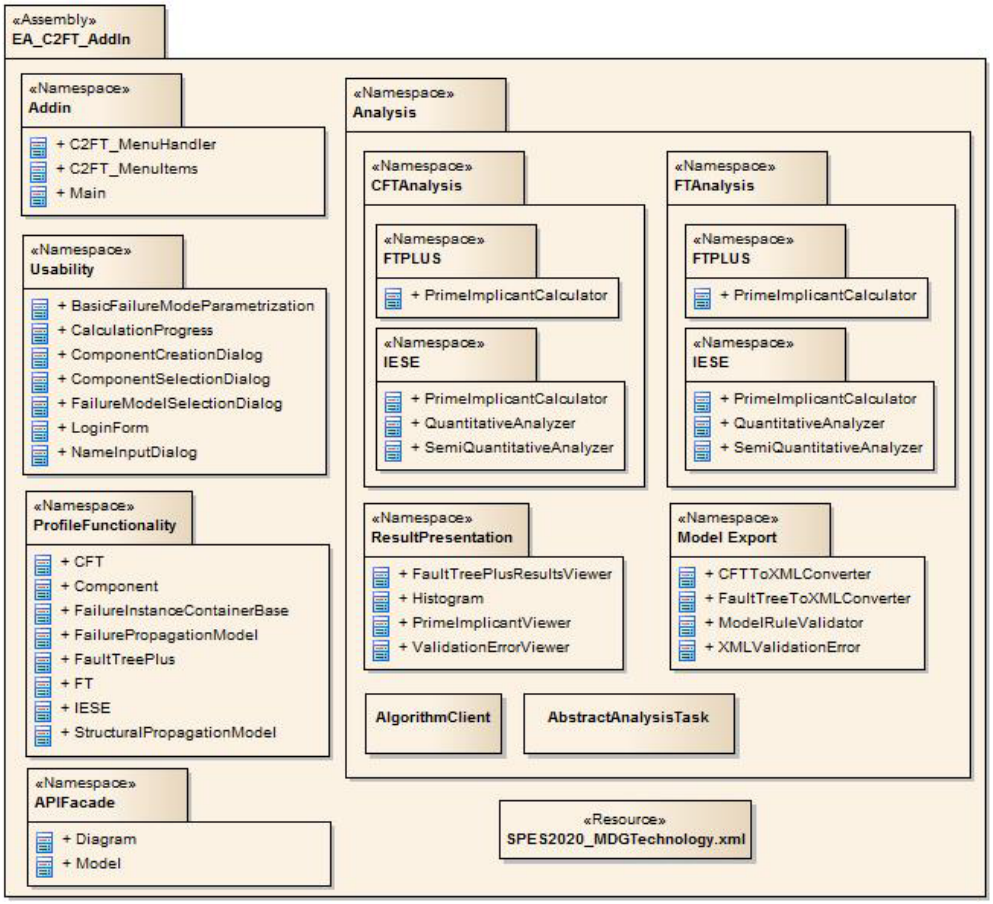


Figure 14 EA C²FT Add-In Structure

In order to extend a declarative profile in an add-in, there has been created one class for each profile. The motivation for this has been the fact that the names of the defined tagged values and stereotypes proved best to be encapsulated in constant variables to make them better maintainable. The profile classes implement the profile specific behavior that cannot be expressed directly by functions from the *APIFaçade* namespace. This namespace rep-

resents a function repository containing profile-independent behavior for modification and retrieval of both model elements and their diagram representations. Examples are synchronization functions for ports between instances and their components or the layout of diagram elements. Note that the profile classes are supposed to make as much use of the *APIFaçade* namespace as possible to minimize code duplication and keep the maintenance and optimization scope small.

- Namespace *AddIn*

This namespace is of special interest, because it incorporates the main class *Main* which consists of the callbacks that react to the EA broadcast events. While the class *C2FT_MenuHandler* defines the hierarchy of the custom menu that is available for the user in the EA application, the class *C2FT_MenuItems* incorporates the callbacks for the broadcast event *EA_MenuClick* for all custom menu commands. Because of the facts that *Main* already contains a lot of callbacks and that the custom menu logic is a responsibility on its own, it has been decided to outsource this logic into separate classes.

- Used UI dialogs

The namespaces *Usability* and *ResultPresentation* only consist of user interface dialogs created with the library for windows forms located by default in .NET framework namespace *System.Windows.Forms.* Especially the classes of the *Usability* namespace are likely to be useful for future add-in extensions, because they have been developed for generic use.

- Common behavior of failure meta-models

With *FailureInstanceContainerBase* and *FailurePropagationModel* in *ProfileFunctionality* namespace, the handling of instances and interface failure modes, respectively, is abstracted from, so most of the work for the integration of a new failure meta-model is done by inheriting from *FailureInstanceContainerBase* and using the functionality defined in *FailurePropagationModel*. Thus, the actual profile classes contain only specific functionality for each respective failure meta-model, apart from hierarchical and interface issues. This approach has been applied successfully for the “CFT” and “Structural Propagation” profiles’ programmatic extensions defined in classes “*CFT”* and “*StructuralPropagationModel”*.

- Abstraction from analysis tasks

The behavioral “Gang of Four”-design pattern “Template Method” [24] is used for facilitating the implementation of new analysis operations by means of the class *AbstractAnalysisTask*, which is the base class for each imple-

mented analysis operation like *PrimeImplicantCalculator* or *QuantitativeAnalyzer*. It encapsulates validation error handling, the asynchronous delegation of analysis tasks to the MTL and most important, the control sequence for analysis operations, i.e. only the contents of failure meta-model specific functions have to be implemented, not their correct invocation sequence. The functions include for example the serialization for a specific failure meta-model or how analysis results are presented and stored.

In addition, the *AbstractAnalysisTask* catches validation errors that are thrown in the *ModelRuleValidator* class. This class offers the possibility to define several rule sets that can be checked for a given serialized failure model. For example, there is a rule set for CFTs that checks if parameters for the required analysis operation are set correct and if element names don't contain forbidden characters. These are only simple checks but the validator is designed to be very flexible concerning the addition and the reuse of rules. For defining rule sets, it uses the concept of C# delegates, which can be compared to C++ function pointers that are type-safe. The idea is that each rule is defined in one function and a rule set is just an array of delegates to the considered functions. The concept is similar to that of commonly used unit testing frameworks like *JUnit* [25], where the rule sets are similar to test suites and the rules are similar to test cases.

The serialized and validated failure models are sent to the MTL during analysis. For the communication between FEL and MTL, the .NET part of the existent communication library already described in section 4.1.2 is used.

4.2.2.3 *Development Hints*

This section gives two possible procedures for the intended sequence of development tasks for adding new features to the add-in.

- Adding a new analysis operation to the add-in

    1. Add a menu entry to *AddIn::C2FT_MenuItems* by adding a class implementing the interface *IMenuItem*

    2. Register menu entry to menu structure in *AddIn::C2FT_MenuHandler*

    3. Add a new communication module to *Analysis::AlgorithmClient*

    4. Create a class for the analysis operation inheriting from *Analysis::AbstractAnalysisTask* and implement the abstract methods by using the communication module created in 3.

    5. Create or reuse a windows forms UI dialog class from *Analysis::ResultPresentation* to present the results to the user

- Adding a new failure meta-model to the add-in

    1. Create a class for the failure meta-model profile inheriting from *ProfileFunctionality::FailureInstanceContainerBase*

    2. Store there all tagged values and stereotypes defined in the declarative profile

    3. Use *ProfileFunctionality::FailurePropagationModel* and APIFaçade namespace functionality to implement the profiles extended behavior

    4. Create a namespace under *Analysis* namespace for the safety analysis technique, which contains namespaces for each supported safety analysis back end

    5. Create a class in *Analysis::Model Export* namespace for XML serialization of the meta-model in case analysis is required

    6. Add a new model validation rule set for the added failure meta-model to *Model Export::ModelRuleValidator* and define rules for it

    7. Add a menu entry to *AddIn::C2FT_MenuItems* by adding a class implementing the interface *IMenuItem*

    8. Register menu entry to menu structure in *AddIn::C2FT_MenuHandler*

    9. Create callbacks for the needed add-in broadcast events and use the methods from profile class to implement the callbacks appropriately

4.2.2.4 *CFT Analysis Example*

The sequence of the tasks done during the semi-quantitative analysis of CFTs in the FEL is visualized by a UML activity diagram in Figure 15. Its chronological continuation in the MTL has already been shown in Figure 8 in section 4.1.2.3.

Initially, the menu command for semi-quantitative analysis is clicked by the user, which causes the EA API to fire the *EA_Clicked* broadcast event to the add-in, where the callback is called. Since a top event from a currently opened CFT diagram was selected, this CFT model as well as the parameters needed for semi-quantitative analysis are serialized to a XML representation. In addition, a hash value for the model is computed for the XML representation. Subsequently, the *ModelRuleValidator* performs a syntactical validation by checking the rules from the rule set that has been defined for this analysis operation. If the validation produces no errors, the top events tagged value for semi quantitative results is checked for existing results. The

computed hash value for the CFT serialization is unique for the current CFT element topology and parameterization. If results are available and the hash value didn't change since the last analysis of the same top event, the results are simply retrieved from the CFT and presented to the user. Otherwise, semi-quantitative analysis has to be performed again. Therefore, a new analysis task is created, i.e. a message according to the AS*ClientLibrary* protocol is created which includes the CFT XML representation and the target top event. Subsequently, this message is sent to the MTL.

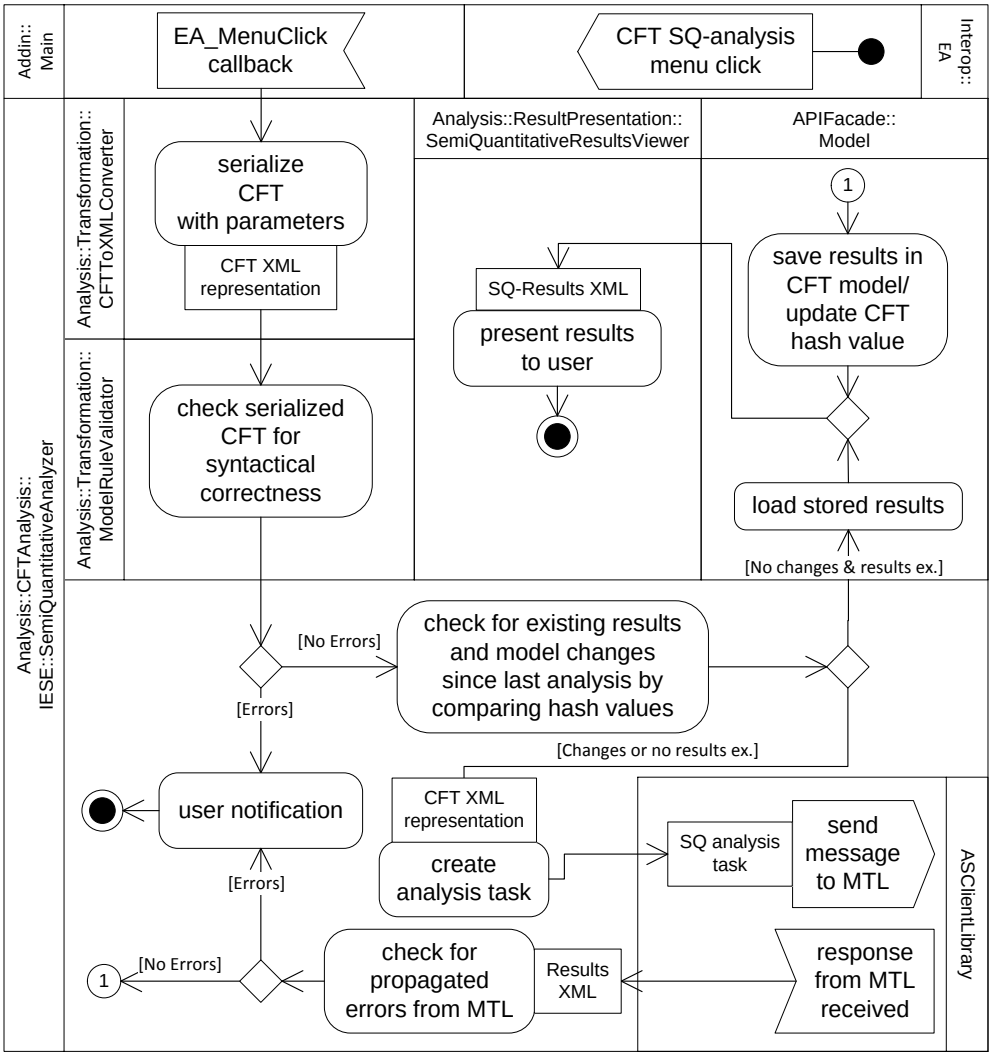


Figure 15 CFT Semi-quantitative Analysis Example in FEL

When the *ASClientLibrary* receives a response from MTL, it is forwarded to the *SemiQuantitativeAnalyzer*, which first checks the result XML structure for defined error tags. If an error is found, the user is notified. Otherwise, the

analysis results are stored in the top events tagged value for semi-quantitative results and then presented to the user.

4.3 Back End Integration

This section describes, which back end analysis operations have been integrated in the course of this thesis and how their results are presented to the user in EA. Furthermore, it gives a brief overview of the required tasks to integrate a new analysis back end consistently with the existing ones in the MTL.

The main focus of the analysis back end integration in this thesis was to demonstrate that the connection to different analysis back ends can be established. However, not all operations that the back ends are able to perform have been implemented.

Concerning FTA, the EA prototype supports both quantitative and qualitative analysis. For the IESE back end, the supported quantitative operation is the top event probability computation for constant, uniform or exponential distributions, while the qualitative one is the computation of prime implicants. With respect to FaultTree+, the prototype is able to compute minimal cut sets as well as top event probabilities with a large range of possible failure distributions like Weibull or Poisson distributions (see [26] for a complete list).

Examples of how the analysis results are visually presented to the user in EA can be seen in Figure 20 in section 5.1 and in Figure 26 in appendix in section 6.1.

In case a new analysis back end adapter should be integrated, the following tasks have to be accomplished:

1. Create a new package named after the convention *de.fhg.iese.<failureModelName>.analysis.<backendToolName>*

2. Create a package inside 1, which is responsible for providing an interface to the API of the back end tool in Java. This step must only be done in the case, when the API is not implemented in Java.

3. Create another package which contains the classes that encapsulate the actual analysis operations and use the java interface from 2. These classes have the responsibility to transform the SDM failure models into a representation that is understandable by the analysis back end tool API, e.g. the flattening in case of CFT analysis with FaultTree+. By convention, the analysis methods in the classes should only take the respective SDM failure models as single parameter to keep usage easy and dependencies minimal.

4.4 Crosscutting Aspects

This section addresses aspects that were considered during the development of the prototype but were also hard to be classified into front end layer or model transformation layer.

- Failure model serialization

For the serialization of failure models, the data structure XML has been chosen, because it supports hierarchies by default and because there are robust libraries available that allow very efficient processing of XML documents, which appears to be extremely useful when considering the size of realistic embedded systems. For a better illustration of how the concrete serialization looks like, a CFT example model has been created and the accomplished analysis operation has been a prime implicants calculation with the IESE analysis back end. The CFT model as well as its serialization and the serialized analysis results can be viewed in appendix under section 6.2. Note that, instead of serialization, it would theoretically be possible to have direct access to the EA repository from Java by means of a Java implementation of the EA automation interface, but this approach has not been applied, because synchronization mechanisms have to be considered, when the “EA Data Model” is read and written by two APIs. Additionally, the *APIFaçade* doesn’t have to be maintained and optimized twice.

- Validation strategy of model serialization and transformation

An important aspect with respect to model serialization and transformation is their validation, i.e. the test whether the failure models that are modeled in the front end are semantically equal to the SDM failure models after transformation. A formal proof at the current development stage is not practicable, because the SDM failure meta-models will likely change due to new research results, so the transformation’s correctness proof would have to be redone for each change.

For this reason, another testing strategy has been chosen: Simple failure models have been created in EA and subsequently analyzed with the IESE analysis back end. Simple means in this context that their complexity has been low enough to verify the results manually. Correct analysis results imply a correct model transformation under the precondition that the analysis back end works correctly. With the intention of achieving an acceptable test coverage for CFTs, the created test models included all available basic modeling elements at least once and especially gates were modeled with different numbers for input and output connections. Special attention was paid to CFT instantiation concerning different hierarchy levels. However, it’s worth noting that only the intended usage of modeling elements was tested, e.g. basic events with input connections were not tested.

- Error Handling

Because of the fact that there exist two different runtime environments, decisions concerning error handling had to be made, namely how errors should be propagated and where they should be presented to the user. This wasn't a problem before the integration of the EA front end, because MagicDraw plugins are also developed in Java, so errors in the analysis back ends could be directly shown to the user by means of Java UI dialogs at those places where they occurred. During the EA integration, the approach has been taken to propagate errors through the TCP communication in order to show them in .NET UI dialogs fitting perfectly in the EA environment. The underlying reason for this decision was that Java UI dialogs certainly popped up when errors occurred, but the user didn't notice it, because the dialog window didn't get the focus. In addition, the user experience within EA gets improved.

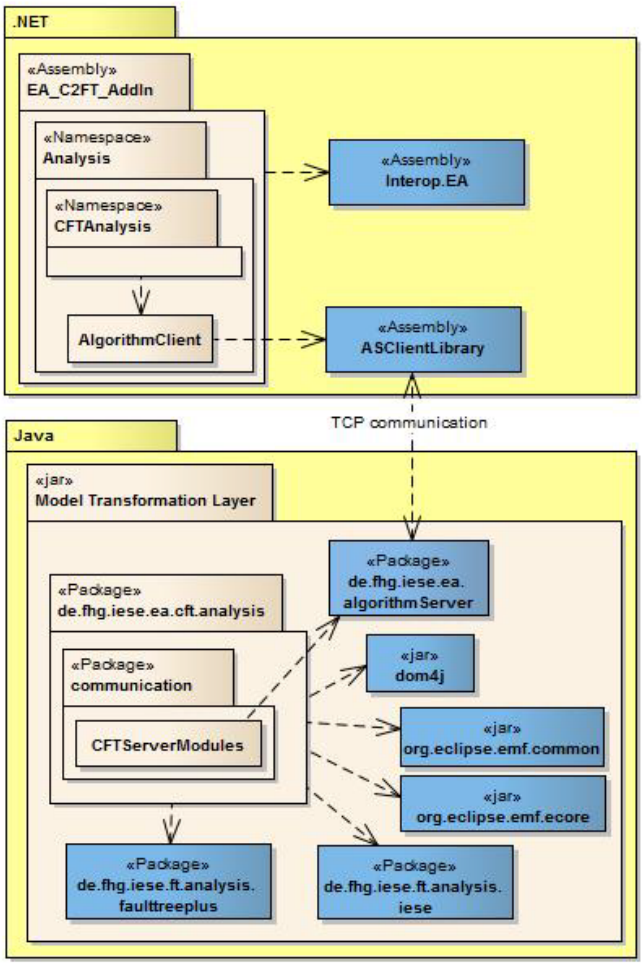


Figure 16 Deployment Setting and Inter-Layer Communication

- Deployment and inter-layer communication

The overall impression of how the communication between front end and model transformation layer takes place and how the developed prototype including external libraries (shown in blue) is deployed is shown in Figure 16. Note that the *Analysis* namespace on .NET side and the *de.fhg.iese.ea.cft.analysis* namespace on the Java side are only shown exemplarily for analysis communication and are by no means complete.

5 Evaluation

The main task of this thesis has been the development of a prototype that integrates the ability to create and analyze safety models into the architectural modeling tool Enterprise Architect. The considered safety analysis technique has been FTA and it should be shown that C²FT models can be modeled in EA and analyzed with the IESE algorithm back end as well as with the commercial FTA software FaultTree+.

The developed prototype's operation is shown by an example system that is created and analyzed in section 5.1. Section 5.2 summarizes the requirements of this thesis and concludes the degree to which they have been fulfilled. Finally section 5.3 contains some possible directions for future work with respect to the SPES_XT requirements that have not been satisfied yet

5.1 Prototype Evaluation

The evaluation of the developed prototype in this thesis is structured in three parts: Initially, the example system's component model is presented with a description of the implemented features for it. Subsequently, the failure models for the system's components are given as well as their parameterization for analysis. Finally, the computed results for the implemented analysis back ends IESE and FaultTree+ are shown.

5.1.1 Component Model

In order to perform the evaluation, an example system describes a *ComponentA*, which is refined into two further components *ComponentB* and *ComponentC*. *ComponentB* and *ComponentC* are leaf components, i.e. they don't contain any instances of other components and therefore their internal structure is not relevant here.

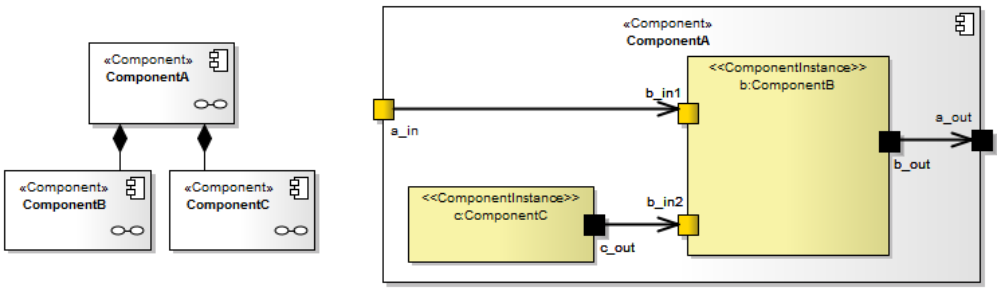


Figure 17 External Structure (left) and Internal Structure (right) of ComponentA

The two possible views of the system's component model are shown in Figure 17. On the left side, the *component external structure* view of *ComponentA* is shown. As explained in section 4.2.1.2, this view is supposed to be used, when a component's whole refinement structure over all existing hierarchy levels is of importance. In contrast, the *component internal structure* view shown on the right side of Figure 17 depicts the realization of only one refinement level. Note that the synchronization of model changes between *component external structure* and *component internal structure* views, e.g. the creation or deletion of component instances, has not been implemented yet.

When ports are created or changed for a component instance in *component internal structure* view, this is automatically synchronized with its classifier component as well as with all of the classifier component's instances. During the synchronization, consistency checks for duplicate port names are performed. In order to support the feature that allows component instances to expose only a port subset of its classifier, the port deletion with component instances hasn't any effect on the classifier. However, the deletion in the other direction is synchronized. When a completely-modeled component is instantiated in another component, all of its ports are automatically instantiated as well. The user's task is then only to delete those ports, which are not needed in the specific context.

Additionally, the prototype facilitates navigation by introducing a custom menu for this. It's possible to navigate from a component or one of its instances directly to its associated failure model and vice versa. Depending on the system's structure in the EA Project Browser and its size this can be a huge time saver. Furthermore, the prototype allows navigating from a specific component directly to all other components where the specific component has been instantiated in.

5.1.2 C²FT Models

This section describes the C²FT models, which have been created for the example system's components to show the integrated safety modeling capabilities of the prototype. The models are depicted in Figure 18. *ComponentB's* associated failure model is a CFT, while *ComponentC* is associated to a FT. According to the system's component model, these two failure models are instantiated in the CFT, which is associated with *ComponentA*.

The presented failure models show all possible modeling elements for FTs and CFTs. These include basic events, input events, output events, component instances, fault tree connectors, port mapping connectors and the gate types AND, OR, XOR, M/N and NOT. Although not shown in the models, it is also possible to assign descriptive names for gates. Since the stereotype *CFTInstance* didn't deliver enough context information in its default caption,

this was customized to show the *CFTInstance's* name as well as its classifier's name, failure model type and associated component. This can be exemplarily seen for the instance of *B_FailureModel* in *A_FailureModel*, where

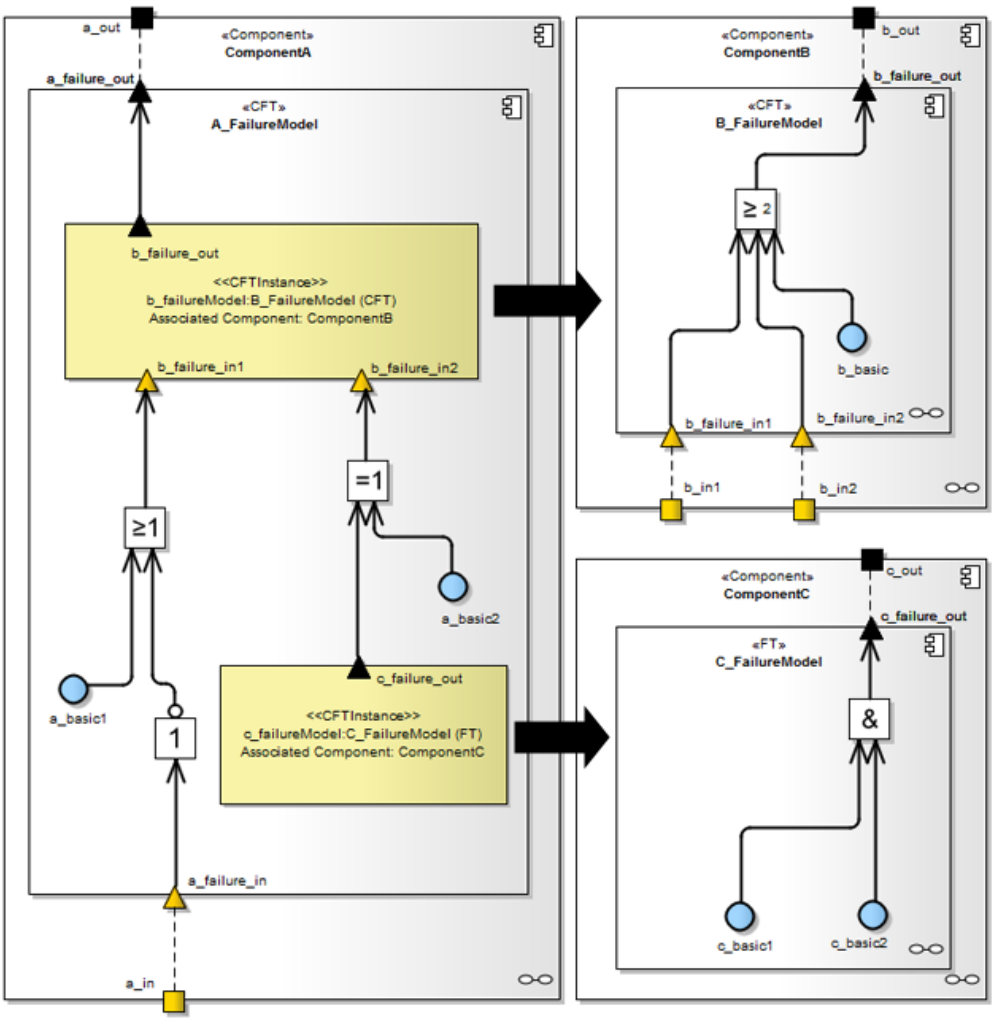


Figure 18 C²FT Models of Example System's Components

"b_failureModel" is the instance's name, "B_FailureModel" is its classifier's name, "CFT" is the failure model type and "ComponentB" is the associated component.

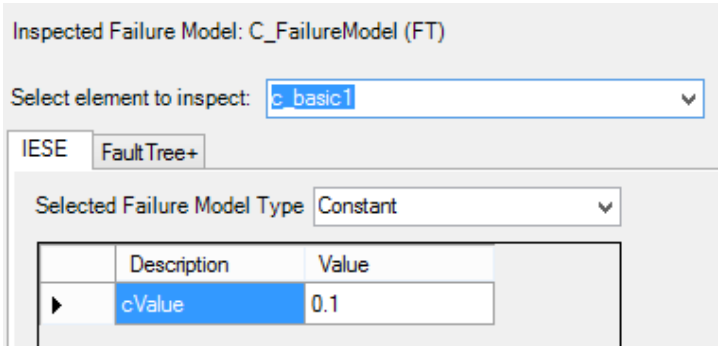
The described features for port synchronization in component models are analogously available for failure events with CFT instances and their classifiers. This holds also true for navigation features. Note that port creations and changes in components are also synchronized with the C²FT models.

Apart from modeling C²FTs, the prototype also allows to model FTs and CFTs that are not associated to a component at all. This can be done by selecting an EA package instead of an existing component, when pressing the menu command to create a new CFT or FT.

5.1.3 CFT Analysis

This section describes the prototype's provided mechanism to parameterize failure models and subsequently compares the safety analysis results that have been computed for the example system with the IESE and FaultTree+ analysis back ends.

The developed prototype is capable of performing qualitative, semi-quantitative and quantitative analysis for C²FTs. Note that with semi-quantitative analysis, a quantitative analysis is performed a defined number of times, where the failure probability for basic failure modes is randomly determined from a normal distribution between defined boundaries for each iteration. The computation results are visualized by a histogram chart. Since randomization makes result validation difficult, semi-quantitative analysis is not considered in this section.



Basic Failure Mode	Failure Model Type	Parameter Description	Probability
c_basic1	Constant	cValue	0.1
c_basic2	Constant	cValue	0.3

Figure 19 CFT Parameterization Dialog and Parameterization for C_FailureModel

All parameters in the EA back end analysis profiles explained in section 4.2.1.3 are represented by tagged values. Because of the fact that EA doesn't provide a comfortable way for editing tagged values, a custom UI dialog, which can be seen in Figure 19, has been developed to enable the parameterization of failure models. The scope of this dialog is always only one failure model, in the example it's *C_FailureModel*. The upper combo box control provides all elements that are contained by C_*FailureModel*, grouped

by their element types “Input Events”, “Output Events”, “Basic Events” and “Gates” for a better clarity. When an element is selected, the tab control is updated with the currently set parameters for the selected element, separated in different tabs for each supported analysis back end.

Note that many parameter descriptions have a prefix character that looks superfluous at the first glance, however it’s mandatory, because there are some parameters in different failure model types having the same name and EA requires an element to have different names for different tagged values.

In order to demonstrate quantitative analysis in an understandable way, the analysis is performed only for the example system’s *ComponentC*, whose associated FT has two conjugated basic events *c_basic1* and *c_basic2*. The parameterization for the *C_FailureModel’s* basic events is shown in Figure 19. The IESE back end delivered a top event probability of 0.03 for the C*_FailureModel’s* output failure mode *c_failure_out* with the given parameterization, which can be easily verified manually.

a_failure_out — FaultTree+ Minimal Cut Set Calculation Results

Unavailability: 0.17235941746797848 Frequency: 0.6070507502520702 Analysis Date: Sun Oct 27 15:58:46 CET 2013

Order	Unavailability	Minimal Cut Sets
2	0.03317456389511551	b_failureModel.b_basic, a_failure_in
2	0.005687068096305516	b_failureModel.b_basic, a_basic1
3	0.105	a_basic2, c_failureModel.c_basic1, a_failure_in
3	0.06999999999999999	a_basic2, c_failureModel.c_basic2, a_failure_in
3	0.018000000000000002	a_basic2, c_failureModel.c_basic1, a_basic1
3	0.012	a_basic2, c_failureModel.c_basic2, a_basic1
3	0.010236722573349928	b_failureModel.b_basic, a_basic2, c_failureModel.c_basic1
3	0.006824481715566618	b_failureModel.b_basic, a_basic2, c_failureModel.c_basic2
4	0.018666666666666668	a_basic2, c_failureModel.c_basic2, c_failureModel.c_basic1, a_failure_in
4	0.003200000000000001	a_basic2, c_failureModel.c_basic2, c_failureModel.c_basic1, a_basic1
4	0.0018198617908177653	b_failureModel.b_basic, a_basic2, c_failureModel.c_basic2, c_failureModel.c_basic1

a_failure_out — IESE Prime Implicant Calculation Results

Order	Prime Implicants
2	b_failureModel.b_basic, - a_failure_in
2	b_failureModel.b_basic, a_basic1
3	a_basic2, b_failureModel.b_basic, - c_failureModel.c_basic2
3	a_basic2, b_failureModel.b_basic, - c_failureModel.c_basic1
3	a_basic2, - c_failureModel.c_basic2, - a_failure_in
3	a_basic2, - c_failureModel.c_basic1, - a_failure_in
3	a_basic2, - c_failureModel.c_basic2, a_basic1
3	a_basic2, - c_failureModel.c_basic1, a_basic1
4	- a_basic2, b_failureModel.b_basic, c_failureModel.c_basic2, c_failureModel.c_basic1
4	- a_basic2, c_failureModel.c_basic2, c_failureModel.c_basic1, - a_failure_in
4	- a_basic2, c_failureModel.c_basic2, c_failureModel.c_basic1, a_basic1

Figure 20 Example System Analysis Results

The qualitative analysis of the example system's C²FT models is shown by the computation of prime implicants in the IESE back end and minimal cut sets in FaultTree+. As the result names already suggest, FaultTree+ eliminates NOT gates from FTs to gain coherent FTs, which have minimal cut sets as result.

Figure 20 compares the computation results of both back end tools, that of FaultTree+ on the top and that of IESE on the bottom. They are presented in tab controls, where a tab is available for each output failure mode or gate that was analyzed qualitatively. The prime implicants as well as the minimal cut sets are ordered in a manner that those results with the lowest order appear at the top of the list, because these include those implicants having the greatest impact on the system's safety. One can verify manually that both results are identical except for the fact that in IESE result view, those implicants having a "-" in front of the name are negative implicants. These are positive in the FaultTree+ result view because of the above mentioned coherence issue.

5.2 Conclusion

This section summarizes the requirements of this thesis and concludes to which degree they have been fulfilled.

The Safe Component Model (SCM) should be evaluated with respect to the SPES_XT requirements.

The SCM's main concern was to implement the SPES modeling approach only for fault tree analysis. This thesis contributed significantly to the evolution of the SCM to the SDM by creating and implementing the necessary model abstractions, which were the first step to achieve an integrated framework that allows performing safety analysis with hetereogeneous analysis techniques.

FTA should be included according to the SDM for the modeling front end EA in connection with analysis back ends from IESE and FaultTree+.

The FT, CFT and C²FT models that resulted from the SPES_XT requirements have successfully been integrated both into the SDM and into EA by means of profiles and an add-in. As a consequence, their modeling in EA is fully supported. By means of the add-in and the implementation of the model transformation layer, qualitative, quantitative and semi-quantitative analysis operations can be performed for FT, CFT and C²FT models in the required back ends. However, the actual goal was to show only *in principle* that analysis tasks for C²FTs can be performed for the back ends within the SPES_XT layered architecture, so the goal's fulfillment has been exceeded by providing analysis capabilities for all commonly used analysis operations.

The developer documentation should be as comprehensive that the prototype implementation can be maintained and extended.

As it is usual in software development, information is presented beginning from an abstract view on the overall system down to the detailed depiction of different aspects of the system's parts. This approach was followed in this thesis' structure, too. The architecture's description in section 3 first puts the notion of an abstract safety exchange layer into context. Then, it describes the functional contents of the front end layer and model transformation layer and their interaction in principle. Finally, their actual implementation for the front end EA and the analysis technique FTA is explained in section 4.

Developers not familiar with the system can draw on sections 3 and 4 until they reach the subsections including concrete development hints that help them to maintain and extend the system. In addition, general information about the used technologies, e.g. the EA tool architecture or the process of creating MDG technologies, is given including the solutions for pitfalls that have been encountered during the development of the prototype. Their documentation avoids the search for the same problem solutions again and improves the development knowledge for EA developers of Fraunhofer IESE in general.

The model transformation of failure models from front end representation to SDM representation should be correct.

The validation strategy for the model transformation has been detailed in section 4.4. Although the created test cases have succeeded, one has to be aware of the fact that a huge number of different CFT models exist that cannot be tested manually, so an automatic mechanism would be desirable that is able to create and validate random models.

Because of the fact that on the one hand only a minimal set of test cases has been created and on the other hand, unintended usage of model elements has not been tested, statements about the model transformation's robustness can not be made at present development stage. However, this is negligible considering the actual intent of the system, namely its prototype character. At this stage, high test coverage is not needed unless the system is supposed to be used productively.

5.3 Possible Directions for Future Work

This thesis had two different goals: On the one hand, it performed the integration of Enterprise Architect as a modeling front end tool into the SPES tool architecture and on the other hand, it replaced the former used safety exchange format Safe Component Model with the Safety Development Model. In this respect, one could imagine of two possible areas for future work.

Firstly, the EA extension for C²FTs could be developed further with respect to usability and for more comfortable modeling capabilities, because there is much potential for the automation of tasks like it has already been done in the MagicDraw implementation. In addition, the current performance of the extension is not sufficient for the use with systems of realistic size, so a performance optimization could also be taken into consideration.

Secondly, in addition to *fault tree* analysis, SPES_XT also requires the integration of *markov analysis* and *failure mode and effects analysis* (FMEA) into the Safety Development Model. Thus, these techniques are also candidates for being integrated into EA in the future.

6 Appendix

6.1 EA Development Additional Materials

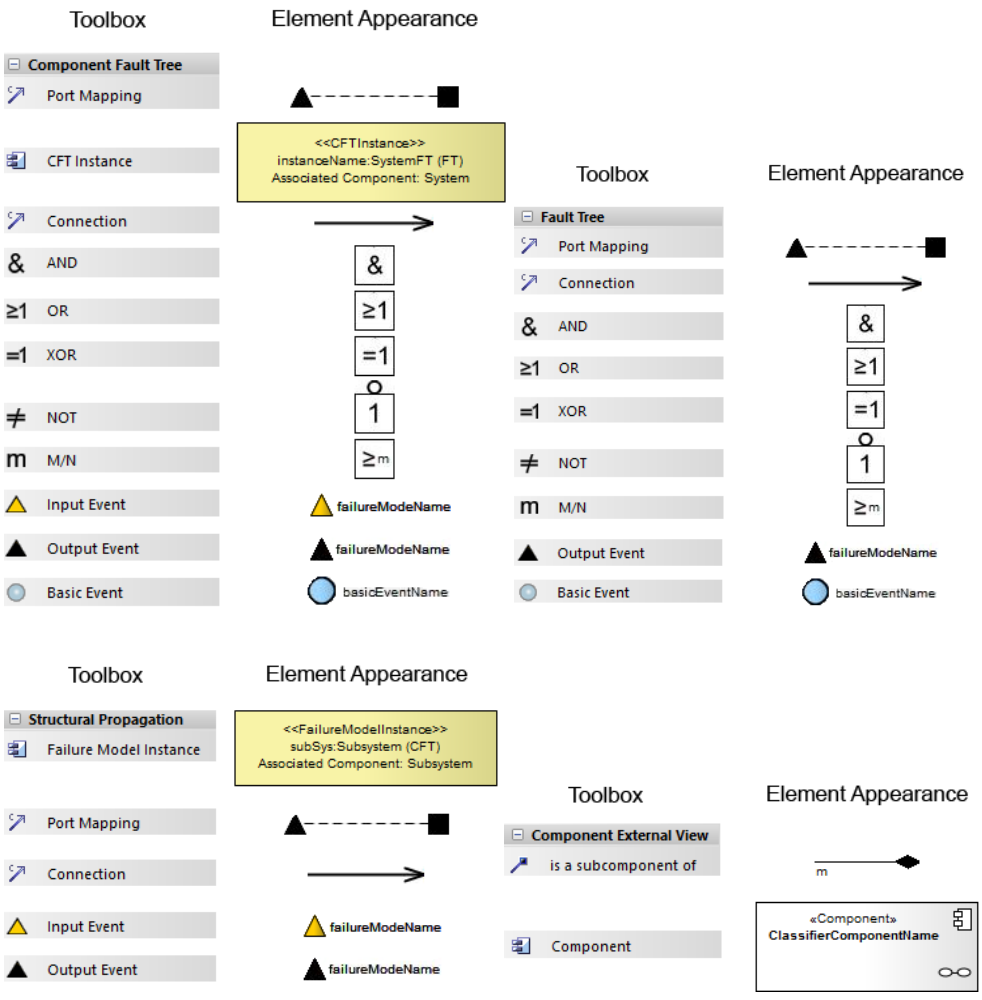


Figure 21 Toolboxes created within the front end prototype

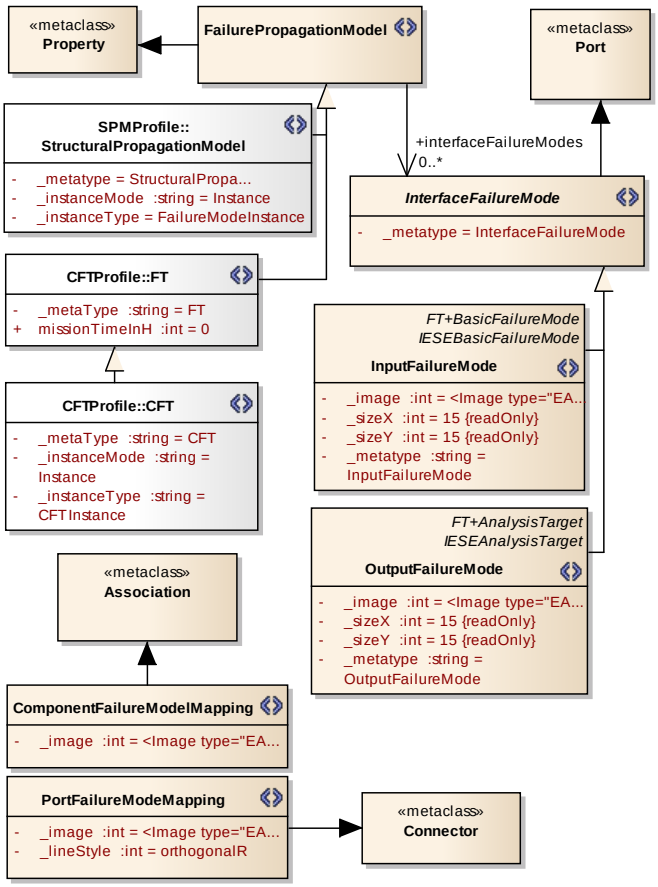


Figure 22 EA Failure Propagation Model Profile

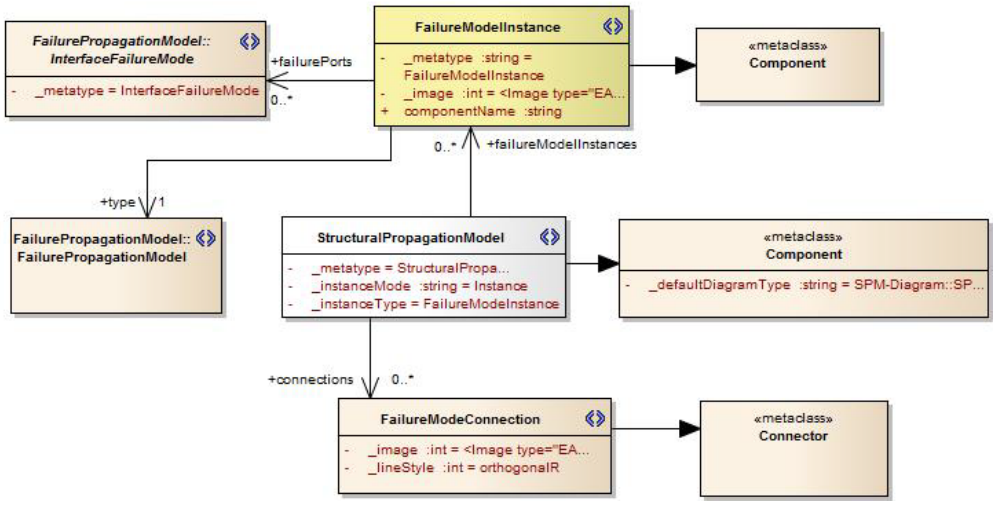


Figure 23 EA Structural Propagation Model Profile

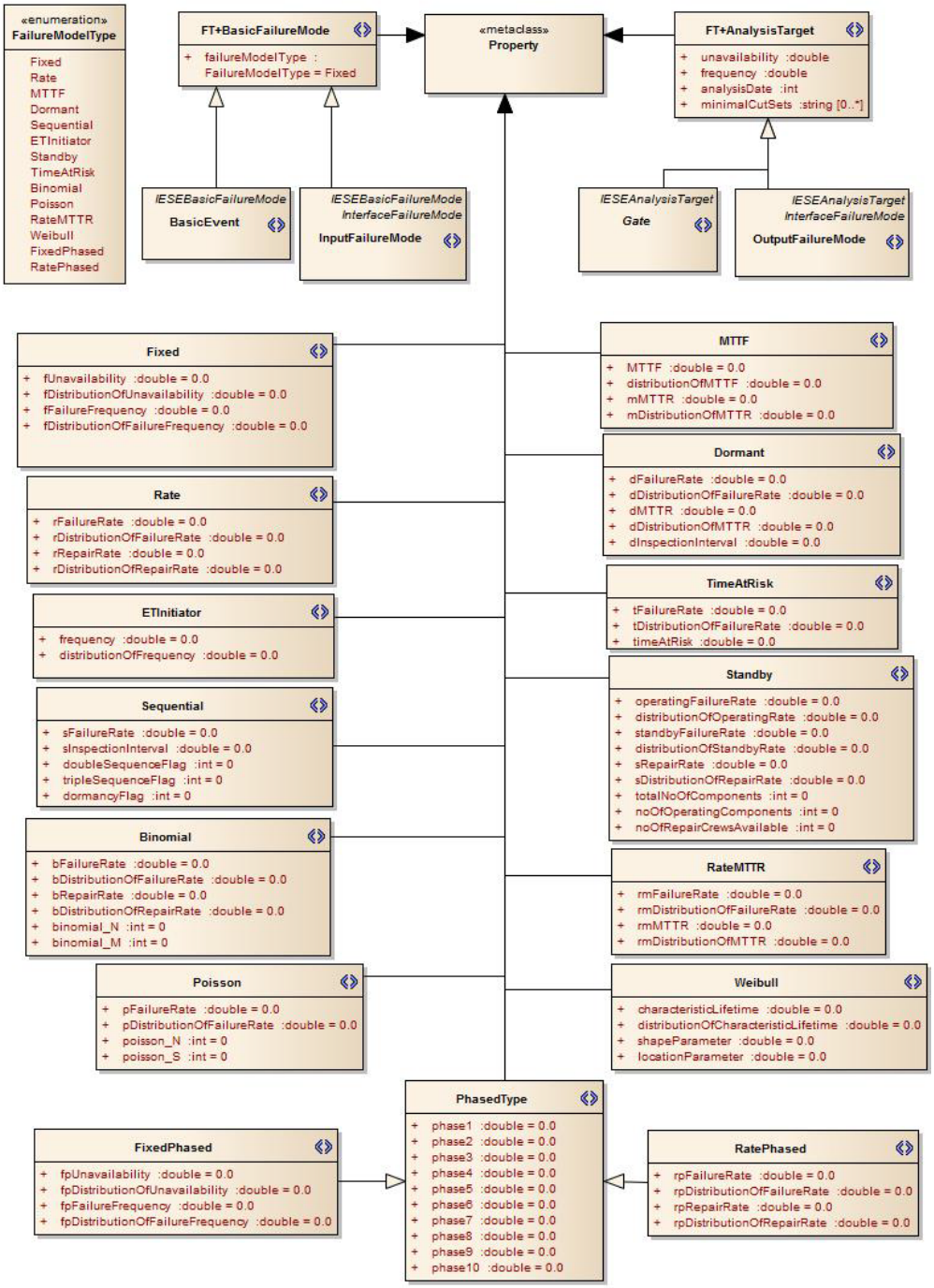


Figure 24 FaultTree+ Analysis Profile

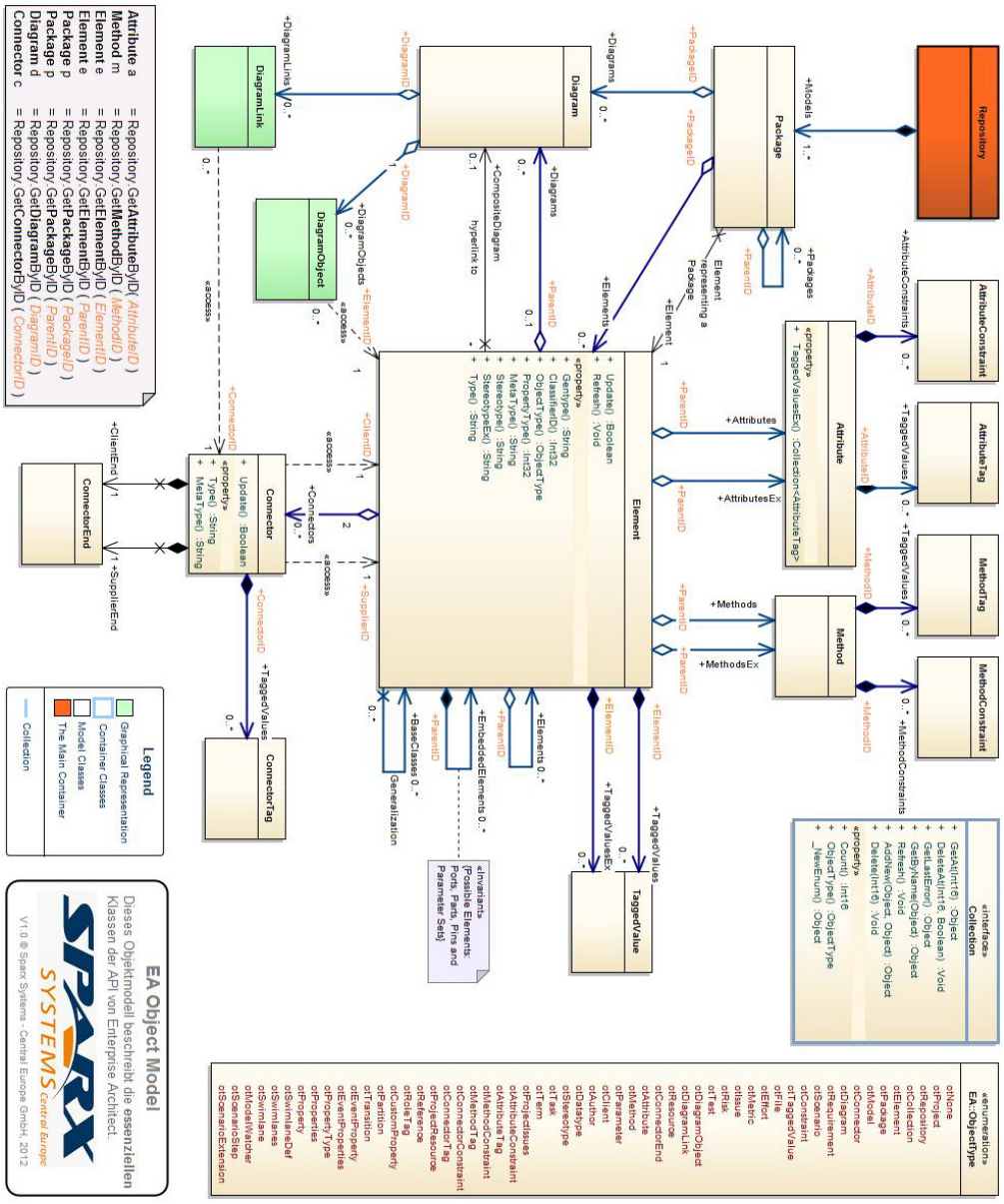


Figure 25 EA Object Model [27]

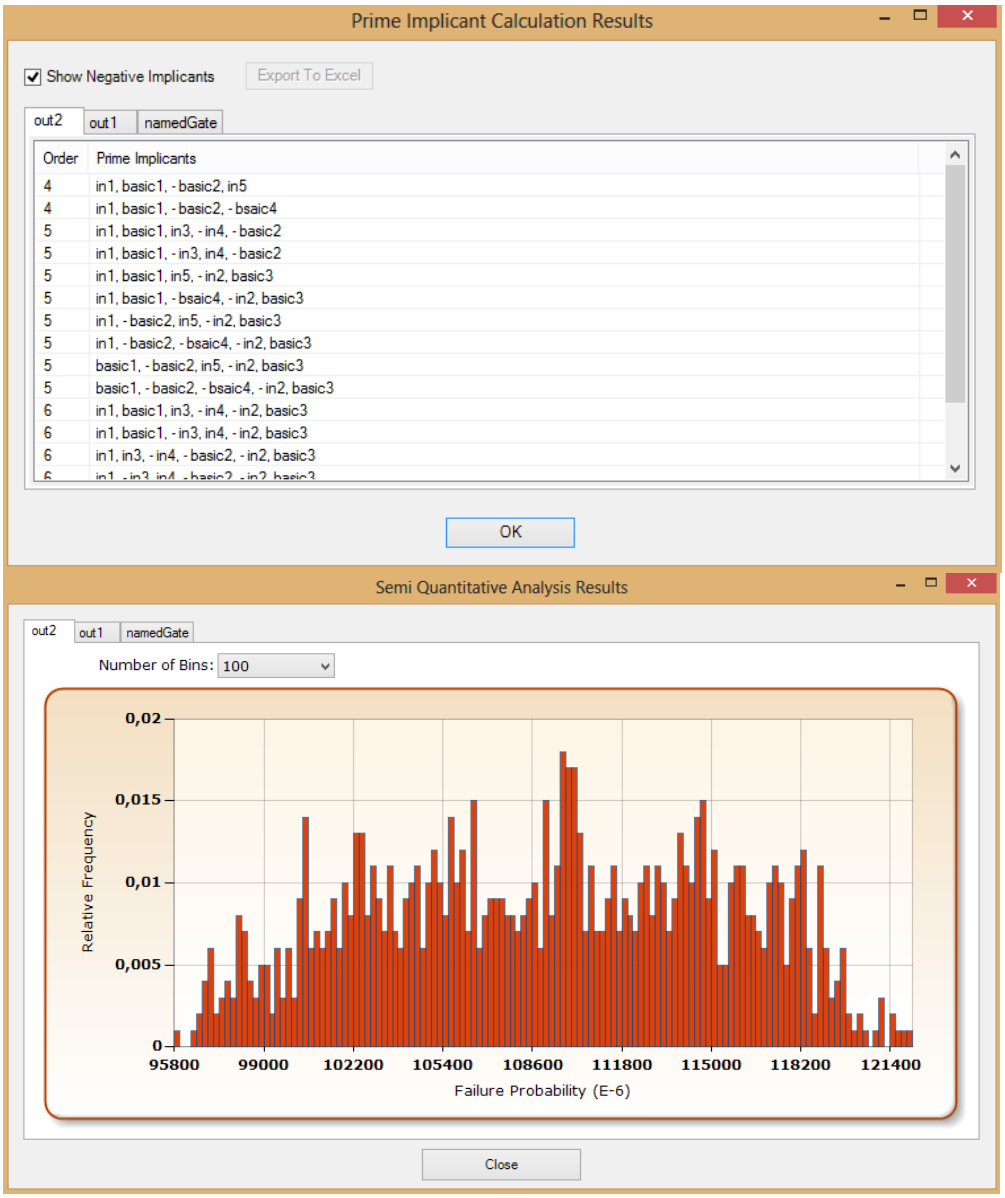


Figure 26 EA Analysis Result Presentation

6.2 XML Serialization Example

6.2.1 CFT Test Model

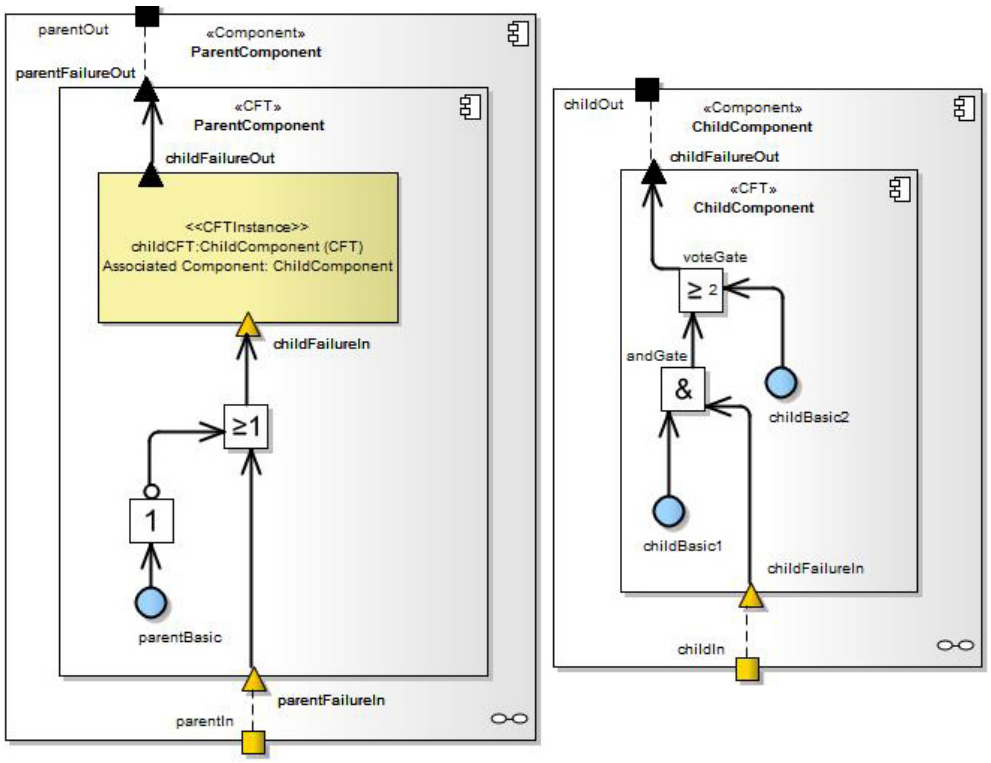


Figure 27 CFT XML Serialization Test Model

6.2.2 Serialized CFT

```
<FailureModel Name="ParentComponent" GUID="{FBB51D05-83B4-40b6-B711-CEF18CA4D542}" FailureModelType
        ="CFT" MissionTime="0">
  <outputFMs>
    <outputFM Name="parentFailureOut" Stereotype="OutputFailureMode" GUID="{D3AF4F2A-B63E-41e6-8356-
        20FDD67B376B}" />
  </outputFMs>
  <inputFMs>
    <inputFM Name="parentFailureIn" Stereotype="InputFailureMode" GUID="{7E5A826F-7417-419e-A32E-
        1AAB075BD684}" />
  </inputFMs>
  <basicEvents>
    <basicEvent Name="parentBasic" Stereotype="BasicEvent" GUID="{64E21F24-9AF6-4339-A794-
        973D867E5B52}"/>
  </basicEvents>
  <gates>
    <gate Name="" Stereotype="NOT" GUID="{A5E2C564-7A48-4f59-B3AA-B105A60AE15C}" />
    <gate Name="" Stereotype="OR" GUID="{AD10B5E7-0FD6-460b-AFCD-F4D6E0E349DF}" />
  </gates>
  <FailureModelInstances>
    <FailureModelInstance Name="childCFT" GUID="{BC013602-1E8D-499a-BA7A-3C67E593CD65}">
```

```
      <failureInports>
        <failureInport Name="childFailureIn" Classifier="{2465A975-81A0-42b8-A85D-757C3D1FF797}"
        GUID="{26413AB4-4B93-4a36-9220-D57F48AD15D0}" />
      </failureInports>
      <failureOutports>
        <failureOutport Name="childFailureOut" Classifier="{D99B3865-B404-4eed-A19E-2A0234A89864}"
        GUID="{A5A02307-4AC0-4e33-8A40-EED6787319C5}" />
      </failureOutports>
      <FailureModel Name="ChildComponent" GUID="{ABEAE856-E1CA-4667-813C-CE65FC3A7F46}" FailureModel
        Type="CFT" MissionTime="0">
        <outputFMs>
          <outputFM Name="childFailureOut" Stereotype="OutputFailureMode" GUID="{D99B3865-B404-4eed-
                A19E-2A0234A89864}" />
        </outputFMs>
        <inputFMs>
          <inputFM Name="childFailureIn" Stereotype="InputFailureMode" GUID="{2465A975-81A0-42b8-A85D-
                757C3D1FF797}" />
        </inputFMs>
        <basicEvents>
          <basicEvent Name="childBasic1" Stereotype="BasicEvent" GUID="{D4CBA227-348D-4abc-A8E2-
                9BA2CC5F3D4B}" />
          <basicEvent Name="childBasic2" Stereotype="BasicEvent" GUID="{AA409D42-72A5-43ed-9B41-
                E6AB67995E0D}" />
        </basicEvents>
        <gates>
          <gate Name="" Stereotype="M/N" GUID="{BFE27FAE-0333-457d-B866-A4506C13EB26}" m="2" />
          <gate Name="" Stereotype="AND" GUID="{5958DE2E-613E-44dd-AB87-18C86847F1E6}" />
        </gates>
        <FailureModelInstances />
        <connections>
          <connection SourceElement="{AA409D42-72A5-43ed-9B41-E6AB67995E0D}" TargetElement="{BFE27FAE-
                0333-457d-B866-A4506C13EB26}" GUID="{FC7611F8-EA39-4e7b-BF03-D3D7C39ABD8C}" />
          <connection SourceElement="{BFE27FAE-0333-457d-B866-A4506C13EB26}" TargetElement="{D99B3865-
                B404-4eed-A19E-2A0234A89864}" GUID="{61142AC7-725E-40db-853D-A273930EC409}" />
          <connection SourceElement="{5958DE2E-613E-44dd-AB87-18C86847F1E6}" TargetElement="{BFE27FAE-
                0333-457d-B866-A4506C13EB26}" GUID="{7DF051F8-396C-4d1d-B70E-964BA90FEF82}" />
          <connection SourceElement="{D4CBA227-348D-4abc-A8E2-9BA2CC5F3D4B}" TargetElement="{5958DE2E-
                613E-44dd-AB87-18C86847F1E6}" GUID="{2D666991-F3D2-4a7a-8222-D7CF035CAB4C}" />
          <connection SourceElement="{2465A975-81A0-42b8-A85D-757C3D1FF797}" TargetElement="{5958DE2E-
                613E-44dd-AB87-18C86847F1E6}" GUID="{06CDE132-6E78-4502-9202-F2A053D14AD5}" />
        </connections>
      </FailureModel>
    </FailureModelInstance>
  </FailureModelInstances>
  <connections>
    <connection SourceElement="{64E21F24-9AF6-4339-A794-973D867E5B52}" TargetElement="{A5E2C564-7A48-
        4f59-B3AA-B105A60AE15C}" GUID="{984EE77B-CBE0-4313-9405-B6909916E8C0}" />
    <connection SourceElement="{A5E2C564-7A48-4f59-B3AA-B105A60AE15C}" TargetElement="{AD10B5E7-0FD6-
        460b-AFCD-F4D6E0E349DF}" GUID="{E4B47117-71A1-4a41-AEC0-D8269427F68D}" />
    <connection SourceElement="{AD10B5E7-0FD6-460b-AFCD-F4D6E0E349DF}" TargetElement="{26413AB4-4B93-
        4a36-9220-D57F48AD15D0}" GUID="{821554EF-5287-423d-80CD-D9A8AE70F3B4}" />
    <connection SourceElement="{A5A02307-4AC0-4e33-8A40-EED6787319C5}" TargetElement="{D3AF4F2A-B63E-
        41e6-8356-20FDD67B376B}" GUID="{771E44B7-1E0C-445b-8212-DAB38D2D271A}" />
    <connection SourceElement="{7E5A826F-7417-419e-A32E-1AAB075BD684}" TargetElement="{AD10B5E7-0FD6-
        460b-AFCD-F4D6E0E349DF}" GUID="{F93613F8-0FC0-4be1-AF2E-AADB06C77C2A}" />
  </connections>
</FailureModel>
```

6.2.3 Serialized Analysis Result

```
<PICalculationResult>
        <ElementOfInterest GUID="{D3AF4F2A-B63E-41e6-8356-20FDD67B376B}" AnalysisDate="Wed Oct 23
                        20:07:42 CEST 2013">
                <PrimeImplicant Order="3">
                        <implicant Name="parentFailureIn" Positive="true"/>
                        <implicant Name="childCFT.childBasic2" Positive="true"/>
                        <implicant Name="childCFT.childBasic1" Positive="true"/>
                </PrimeImplicant>
                <PrimeImplicant Order="3">
                        <implicant Name="parentBasic" Positive="false"/>
                        <implicant Name="childCFT.childBasic2" Positive="true"/>
                        <implicant Name="childCFT.childBasic1" Positive="true"/>
                </PrimeImplicant>
        </ElementOfInterest>
</PICalculationResult>
```

Literature

[1] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V. (BITKOM), "Eingebettete Systeme - Ein strategisches Wachstumsfeld für Deutschland" 2010. [Online]. Available: http://www.bitkom.org/de/themen/54926_62539.aspx. [Accessed Octobre 9th, 2013].

[2] International Electrotechnical Commission, *IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems,* 2010.

[3] International organization for Standardization (ISO), Draft International Standard (DIS), *ISO/DIS 26262: Road vehicles - Functional Safety,* 2009.

[4] Pohl K., Hönninger H., Achatz R. and Broy, M. (Eds.), Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology, Berlin, Heidelberg: Springer, 2012.

[5] Institute of Electrical and Electronics Engineering (IEEE), *IEEE Standard Glossary of Software Engineering Terminology,* 1990.

[6] Heineman, G.T. and Councill, W.T., Component-Based Software Engineering: Putting the Pieces Together, Amsterdam: Addision-Wesley Longman, 2001.

[7] Stahl, T. and Völter, M., Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management, 2. Ed., Heidelberg: Dpunkt Verlag, 2007.

[8] Ericson, C., „Fault Tree analysis - A History“ in: *Proceedings of The 17th International System Safety Conference*, Seattle, WA, 1999.

[9] Leveson, N.G. and Harvey P.R., „Analyzing Software Safety“ in: *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 5, September 1983.

[10] Kaiser B., Liggesmeyer P. and Mäckel O., „A new component concept for fault trees“ in: *Proceedings of the 8th Australian workshop on safety critical systems and software*, Darlinghurst, Australia, 2003.

[11] Domis D., Integrating Fault Tree Analysis and Component-Oriented Model-Based Design of Embedded Systems, PhD Kaiserslautern, 2012.

[12] Electric Power Research Institute, „Computer Aided Fault Tree Analysis System (CAFTA) - Product Abstract“ [Online]. Available: http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?ProductId=000000003002000020. [Accessed October 15th, 2013].

[13] Isograph Reliability Software, „FaultTree+ - Product Description“ [Online]. Available: http://www.isograph-software.com/2011/software/reliability-workbench/fault-tree-analysis/. [Accessed October 15th, 2013].

[14] Siemens AG, „RAM Development for Gasification Plants, Slide 8“ May 5th, 2010. [Online]. Available: http://www.gasification-freiberg.org/PortalData/1/Resources/documents/paper/IFC_2010/02-3-Sutor.pdf. [Accessed October 15th, 2013].

[15] Total Quality Management, „IQ-FMEA - Product Description“ [Online]. Available: http://www.tqm.com/software/fmea-software/iq-fmea. [Accessed October 15th, 2013].

[16] Object Management Group (OMG), „Object Constraint Language (OCL) Specification“ [Online]. Available: http://www.omg.org/spec/OCL/. [Accessed October 26th, 2013].

[17] „dom4j 2.0 open source java library for working with XML, XPath and XSLT“ [Online]. Available: http://dom4j.sourceforge.net/.

[18] „XPath Language Reference“ [Online]. Available: http://www.w3schools.com/xpath/default.asp.

[19] Microsoft Developer Network, „Mithilfe von GUIDs eindeutige IDs erzeugen“ [Online]. Available: http://msdn.microsoft.com/de-de/library/bb979128.aspx. [Accessed October 23rd, 2013].

[20] Sparx Systems, „Official Website“ [Online]. Available: http://www.sparxsystems.com/. [Accessed October 22nd, 2013].

[21] Sparx Systems, *Enterprise Architect v10 Help Contents.*

[22] Killian T., Scripting Enterprise Architect - A Guided tour to Enterprise Architect's scripting capabilities, Leanpub, 2012-2013, Available: https://leanpub.com/ScriptingEA.

[23] Killian T., Inside Enterprise Architect - Querying EA's Database, Leanpub, 2012-2013, Available:https://leanpub.com/InsideEA.

[24] Gamma E., Helm R., Johnson R.E. and Vlissides J., Design Patterns - Elements of Reusable Object-Oriented Software, Amsterdam: Addison-Wesley Longman, 1994.

[25] „JUnit - A programmer-oriented testing unit testing framework for Java“ [Online]. Available: http://junit.org/.

[26] Isograph Reliablity Software, *FaultTree+ V11.2 Technical Specification*.

[27] Sparx Systems Official Blog, „EA API Description“ March 19th, 2003. [Online]. Available: http://blog.sparxsystems.de/2012/03/enterprise-architect-datenmodell-ea-api/. [Accessed October 17th, 2013].

List of Figures