Proceedings of the International Conference on Computer and Information Science and Technology
Ottawa, Ontario, Canada, May 11 – 12, 2015
Paper No. 127

UML Model to Fault Tree Model Transformation for Dependability Analysis

Zhao Zhao, Dorina C. Petriu
Carleton University, Department of System and Computer Engineering
1125 Colonel by Dr, Ottawa, Ontario, Canada K1S5B6
zhaozhao@sce.carleton.ca; petriu@sce.carleton.ca

Abstract- This paper proposes a model transformation to automatically generate Fault Tree models from UML software models annotated with dependability annotations. The goal is to extend the model-driven software development process with the capability of verifying some important dependability properties (such as reliability, safety) starting early in the software lifecycle, by solving the generated Fault Tree models with existing fault tree analysis tools. Feedback from the analysis will help developers in selecting suitable design alternatives in order to build systems that meet their non-functional requirements. The model transformation language used in this study is ATL (ATL Transformation Language). The transformation takes as input UML Composite Structure Diagrams, Sequence Diagrams and Use Case Diagrams, extended with two UML profiles: MARTE (a standard profile adopted by OMG) and DAM (a profile specializing MARTE). The focus of this paper is on the ATL transformation.

***Keywords*:** UML, fault tree, model transformation, software dependability

1. Introduction

By changing the focus of software development from code to models, Model-Driven Development (MDD) enable developers to verify the non-functional properties (NFP) of software (such as performance, availability, reliability, safety, etc.) by transforming UML design specifications annotated with extra information to appropriate analysis models. Many formal modeling techniques and tools have been developed for the analysis of different non-functional properties (e.g., Markov chains, queuing networks, Petri nets, fault trees, etc.) The research challenge is to bridge the gap between model-driven development tools and different existing analysis tools by using model transformation techniques.

Many approaches to model transformations have been based on general purpose programming languages, such as Java; but more recently specialized model transformation languages have also been developed. ATL for instance, offers semantics that allow for the specification of transformation rules, which are much more easily maintained than those expressed in general-purpose programming languages. It opens up the possibility of using formal proofs to verify transformation correctness through ATL's rule traceability capability, which are not feasible with transformations written in general-purpose programming languages.

This paper addresses the problem of automatic derivation of Fault Tree models from a UML software model with dependability annotations. Before deriving fault tree models, we annotate UML design specifications with quantitative dependability attributes. We are using two existing UML profiles for this purpose: The standard UML Profile for Modeling and Analysis of Real-Time Systems (MARTE) (Web-1) and the Dependability Analysis Model (DAM) profile, which extends MARTE (Bernardi et al., 2011) (Bernardi et al., 2013). Figure 1 shows the transformation process of UML+MARTE/DAM models to Fault Tree models. The modeling tool we used for producing UML Diagrams is Papyrus (Web-4), which is an open-source tool based on EMF (Web-2) and supports the MARTE profile. Additionally, we used a DAM profile application plugin (Bernardi et al., 2011) and (Bernardi et al., 2013) that enabled the use of DAM stereotypes in the Papyrus modeling environment. The annotated UML model conforms to the standard UML 2.4 Metamodel (Web3). The Fault Tree Metamodel is created by using Ecore tools. The

ATL transformation we developed, UML2FT, performs the mapping from the annotated UML source model to the Fault Tree target model. The generated Fault Tree model is then processed by a simple JAVA program to make it readable by the FaultCAT open-source fault tree analysis tool (Web-5). We selected FaultCAT because it is the only open-source tool we found that directly support XML input/output.

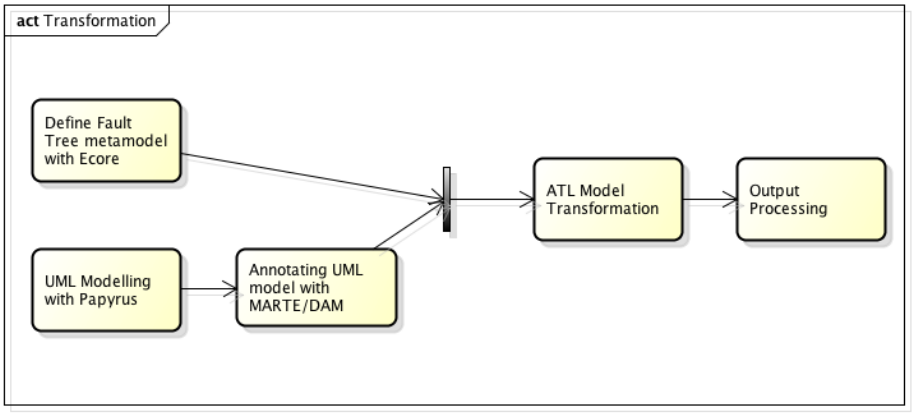


Fig. 1. Activity of UML2FT transformation.

The goal of this paper is to define, implement and test an ATL transformation accepting as input a source model composed of UML composite structure, use case and sequence diagrams (all with MARTE/DAM applied stereotypes) and generating as output a regular Fault Tree model. The steps for achieving this goal have resulted in the following contributions:

a) Transformation mapping, i.e. the design of mapping rules from source to target model elements through the analysis of the problem domain, and
b) Implementation and verification of the ATL transformation.

The paper is organized as follows: section 2 briefly talks about related works; section 3 presents the source and target models and their mapping; section 4 discusses the ATL implementation; section 5 presents verification issues; and section 6 presents the conclusions.

2. Related Work

Existing work on software dependability analysis by transforming UML models to different analysis models is surveyed in (Bernardi et al., 2012). Some of the previous papers, such as (Lauer et al., 2011), (Hu et al., 2011), (Grunske et al., 2005), (Domis et al., 2008) and (Hassan et al., 2005) use fault tree as their target analysis model, but only a few of them applied model transformation languages to achieve automatic generation. Some of the existing approaches (e.g., those considering different failure modes and those using component fault trees) do contain manual steps throughout the derivation of the fault trees, so it is difficult to completely automate the entire generation of the fault trees. Since our focus was on complete automation of the transformation from UML to fault tree models, we have considered the generation of traditional fault trees, which do not require manual intervention.

(Web 6) proposed to consider every possible failure mode for each different element of the UML sequence diagrams, but this may bring too many unnecessary redundant events. Therefore, we choose not to take different kinds of failure modes into account. The works in (D'ambrogio et al., 2002) and (Pai et al., 2002) have presented an approach for automating the model transformation by providing high-level algorithms (implemented in general purpose languages rather than model transformation languages). The former took the system behavior into account by considering a path-based approach and proposed an algorithm for mapping the UML-based specification onto a fault tree model to predict the system failure rate. The latter describes a framework for modeling computer-based systems, based on UML, that

facilitates automated dependability analysis during design, using Dynamic Fault Trees as target formalism to evaluate the system unreliability of fault-tolerant software systems at design stage.

One of the objectives of our work was to take into account the interaction between software components like in (D'ambrogio et al., 2002) and the redundancies in system architecture like in (Pai et al., 2002). Another main objective was to implement the transformation in a specialized Model Transformation Language (in our case ATL) instead of a general purpose languages, compared to those existing works.

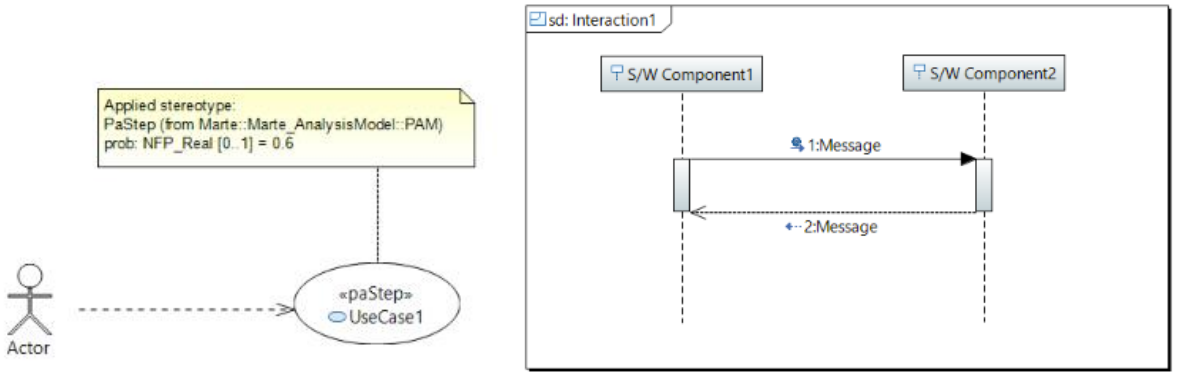
3. Fault Tree Model Derivation

3. 1. Source Model

The approach for obtaining the fault tree is path-based, similar with (D'Ambrogio et al., 2002), where the software architecture is described by sequence and deployment diagrams. The source model in our work includes a UML2 structural view (composite structure diagrams) and a behavioral view (use case and sequence diagrams), extended with MARTE/DAM performance annotations. A sequence diagram is a scenario notation that describes a finite sequence of operations between different objects. A use case diagram is a representation of a user's interaction with the system. In our work, each use case is associated with an interaction. A composite structure diagram shows the internal structure of a classifier, including its interaction points to other parts of the system (Web-3). It shows the configuration and relationship of parts, which together perform the behavior of the containing classifier. We chose to use composite structures in order to model the represented components, allocated hardware devices and connectors. Also, instead of using deployment diagrams to show the actual software to hardware allocation of the system, we chose to use the stereotype "allocate" from the MARTE profile in the composite structure diagram. An annotated UML source model example is shown in Figure 2.

In the Composite Structure diagram, S/WComponent 1 and S/WComponent2 and H/WComponent are stereotyped with <<DaComponent>> from DAM_Profile, which has an attribute "failure" of type "DaFailure" allowing users to enter an "occurrenceProb" for each component.

The hardware spare is stereotyped with <<DaSpare>> and shows its substituted component (in this example is "H/W Component") it also has the complex property type of "failure", as the software components. The stereotype <<Allocate>> from MARTE shows both the client and supplier of allocation, for instance in the example model, both of the software components are being allocated to the same H/W Component. In the sequence diagram, each lifeline represents a component from the composite structure diagram.



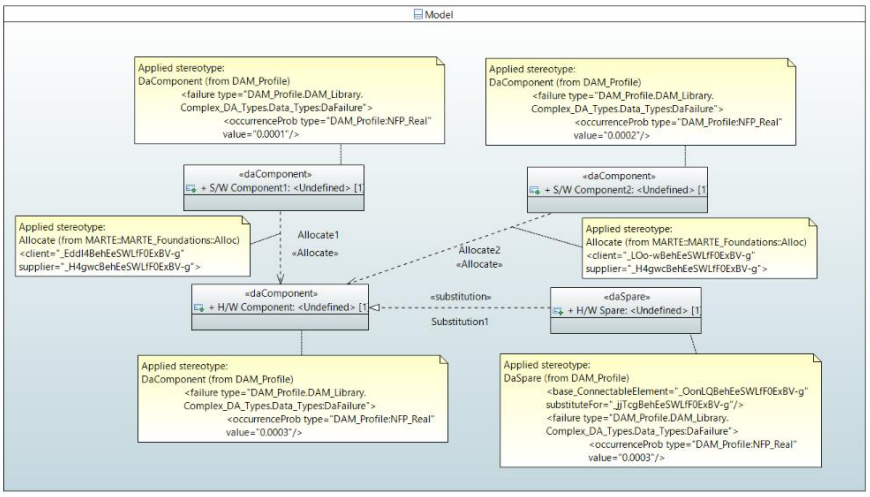


Fig. 2. Annotated UML source model example.

3. 2. Target Model

As there is no standard metamodel for Fault Trees defined in advance, we built a fault tree metamodel for our transformation (see Figure 3) according to the FaultCat analysis tool. The metamodel from Figure 3 describes regular fault trees, which are fully supported by the FaultCAT tool used in this work. In the literature there are extended fault trees models, such as dynamic fault trees, which contain other types of elements (such as spare gate, functional dependency gate or conditional events). Such elements are not being considered in our work yet, but we may refine our transformation in future work.

As we can see from Figure 3, a Fault Tree model contains elements named FTelements, which can be either Events or Gates. Such an element can have children elements, for example an event can have a child gate, and a gate can have several children events. There is a constraint that a gate cannot have another child gate, and an event cannot have more than one child gate. Every event has two attributes: Title and Info (which can be used to store extra useful information). Some specialized events types (BasicEvent, Undeveloped Event and ExternalEvent) have also a Probability attribute, which is the occurrence probability of the event. The note shows that a gate element cannot have another gate as a child element. Figure 4 shows a fault tree example. Note that UndevelopedEvent and ExternalEvent are not used in our model transformation, and the content of Info is not shown in the figure.

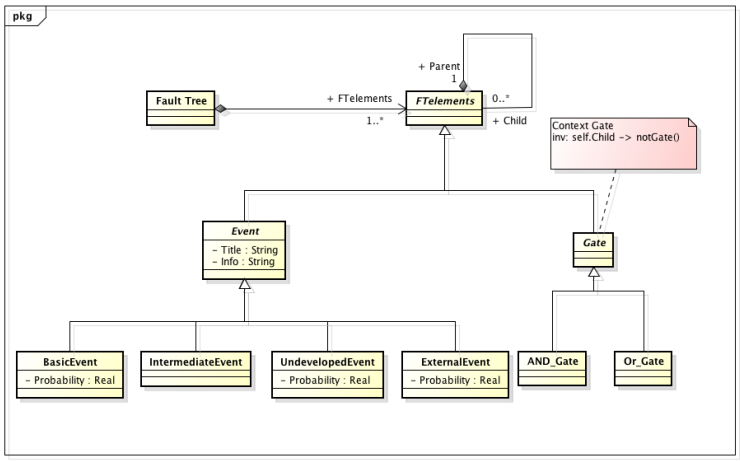


Fig. 3. Fault Tree Metamodel

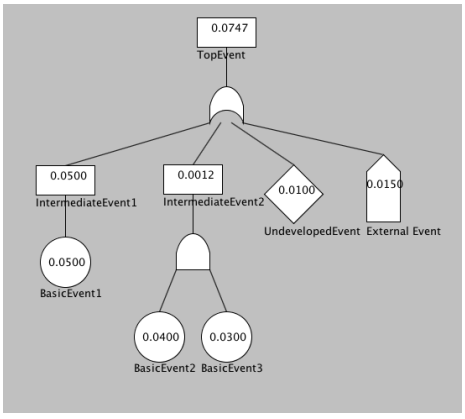


Fig. 4. Fault Tree Example

3. 3. Source to Target Mapping

Based on the metamodels for both UML and Fault Trees, the mapping from source to target can be briefly described as in the table 1 below. A fault tree is generated from the top (which represents the undesired event that the system fails) to the bottom (each leaf is a basic events). In other words, the tree is constructed backwards: for every gate, first is generated the output event and then the input events.

The top level of the fault tree (top event) corresponds to the root of the source model (a Model element), which is the output of an OR gate; the inputs of this OR gate are intermediate events corresponding to the failure of each use case. Each use case has an execution probability. However, for an intermediate event, we are unable to set the probability value, because the intermediate event probability is a result calculated from input events; moreover, the FaultCAT tool does not support conditional events (which could have a probability occurrence set by the user). To overcome this problem, we assign the occurrence probability value to the attribute Info of an intermediate event. Then, when assigning the probability value to any of the children (i.e., basic event) of this intermediate event, we multiply the original failure occurrence probability value by the intermediate event's Info attribute.

Table. 1. Mapping from source to target model

UML+MATE/DAM diagrams	FaultTree Model
Model	IntermediateEvent (TopEvent)
	OR_Gate
<<PaStep>> UseCase	IntermediateEvent
Interaction	OR_Gate
CombinedFragment	IntermediateEvent
	OR_Gate
InteractionOperand	IntermediateEvent
	OR_Gate
BehaviorExecutionSpecification	IntermediateEvent
	OR_Gate
<<DaComponent>> S/W Component	BasicEvent
<<DaComponent>> H/W Component (without spare)	BasicEvent
<<DaComponent>> H/W Component (with spare)	IntermediateEvent
	AND_Gate
	BasicEvent
<<DaSpare>> Component	BasicEvent
<<DaConnector>> Connector	BasicEvent

We consider that a message may fail due to the failure of the hardware link conveying the message. A message can then be mapped to the connector between the hardware components on which the two software components exchanging the message are allocated. For a hardware component, we have two different mapping methods. For those that do not have spare components, we directly map the hardware component to a basic event representing the failure of the component; the failure occurrence probability is an attribute of the applied stereotype <<DaComponent>>. For the hardware components with spares, we generate an intermediate event representing the failure of all hardware components, along with an And Gate (only if all hardware components fail, then the system would fail). The input events to the AND gate are basic events representing the failure of the original hardware component along with its spares as basic events.

4. Implementation

The ATL implementation of the transformation in this study follows the regular structure of ATL transformations. The source, transformation, and target models each has their own separate metamodels, which are each based on a common meta-metamodel (ECORE).

Within the ATL module UML2FT that contains all the transformation rules, there are 7 matched rules for all conditional mappings, and 4 lazy rules for all unconditional mappings. We also used 32 helpers that may be invoked by the transformation rules, 8 of which were particularly important. Examples of each type of rule/helper are shown in the table 2 below.

Table. 2. Examples of main rules and helpers in UML2FT transformation

Type	Name	Description
Matched Rules	CombinedFragment	Transformed CombinedFragment to IntermediateEvent and Or_Gate
Lazy Rules	getHardwareComponent	For those BehaviorExecutionSpecification running on HardwareComponent without spares, find its covered lifeline, then find its represented SoftwareComponent, then find the allocated HardwareComponent, transformed it into BasicEvent
Helpers	getSoftwareProb()	For BehaviorExecutionSpecification, get its covered lifeline's represented SoftwareComponent's failure occurrence probability, returns the value of it

In this implementation, matched rules are used for source elements such as model, use case, interactions, fragments, and certain applied MARTE/DAM stereotypes, whereas lazy rules are used for source elements that satisfy specific conditions generating target elements such as represented software, allocated hardware or spare components. The logic linking of each target elements with one another, based on the ordering of the source model's elements, is embedded within each rule, where the relationship is determined by the containment 'Child' association. This ensures that target elements are properly linked with each other when generated.

Below shows the ATL code of each type of rule/helper example we described above. In Code Fragment 1, each combined fragment is transformed to an intermediate event followed by an Or_Gate. The name attribute of the intermediate event contains the name of the combined fragment itself and its operator kind. Code Fragment 2 shows the helper getSoftwareProb() in which the getAppliedStereotype() call retrieves the stereotype with the specified qualified name that is applied to this element, or null if no such stereotype is applied. The getValue() call retrieves the value of the property with the specified name in the specified stereotype for this element. So that this helper returns the occurrenceProb value of the complex data type 'DaFailure' from stereotype 'DaComponent'.

Code Fragment. 1. Matched rule example – CombinedFragment

```
rule CombinedFragment{
from
    uml: MMa!CombinedFragment
to
        Event: MMb!IntermediateEvent(
                Title <- uml.name + uml.interactionOperator + 'fail',
                Child <- Gate
                ),
        Gate: MMb!Or_Gate(
                Child <- uml.operand
           )
        }
```

Code Fragment. 2. Helper – getSoftwareProb()

```
Helper context MMa!BehaviorExecutionSpecification
        def: getSoftwareProb(stereotype: String): OclAny =
        if self.covered.represent.getAppliedStereotype('DAM::')+ stereotype).oclIsUndefined()
        then OclUndefined
        else self.covered.represent.getValue('DaComponent','Dafailure.occurrenceProb')
        endif
;
```

5. Verification and Case Study

We have used seven test cases and two case studies to verify our transformation. Each test case was designed to cover one or more specific conditions. These tests will cover all the matched rules and lazy rules introduced in section 4. During the execution, every helper has been called at least once.

We also included two more complex case studies in which all the main aspects of our model transformation was covered and tested. For instance, the second case study is an example of a Message Redundancy Service (MRS) from (Bernardi et al., 2013), as shown in Figure 5 and 6. Figure 7 shows the composite structure diagram for MRS. As we can see from the diagrams, the Use Case Diagram shows the main use case realized by the scenario given in the Sequence Diagram in Figure 6. MessageReplicator receives messages from Clients, and then specifies target receivers and the file to deliver. The Redundancy Manager, which is in charge of the actual delivery, creates replicas called Payloads. Each Payload sends back to RM a result that can be either of approval or of rejection. The fault tree generated by our transformation is shown in Figure 8.

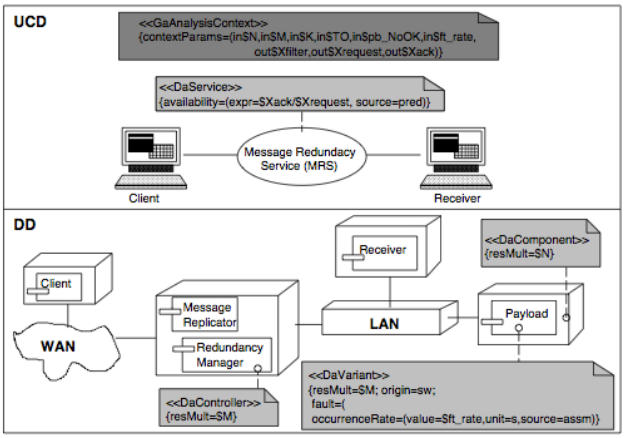


Fig. 5. Message Redundancy Service overview (UCD) and architecture (DD)

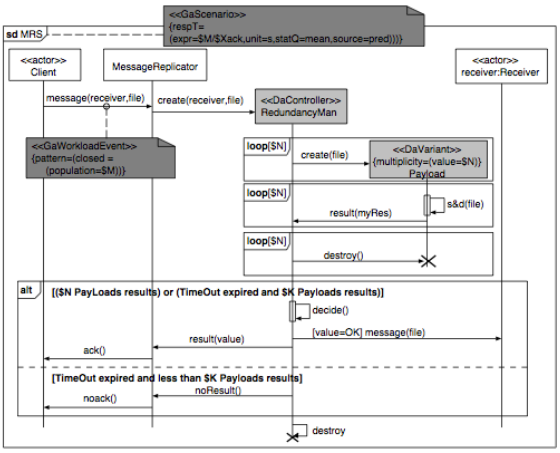


Fig. 6. MRS scenario (SD diagram)

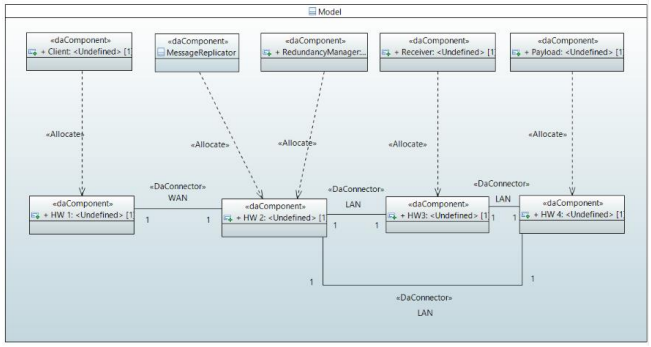


Fig. 7. Rebuilt Composite Structure diagram for MRS

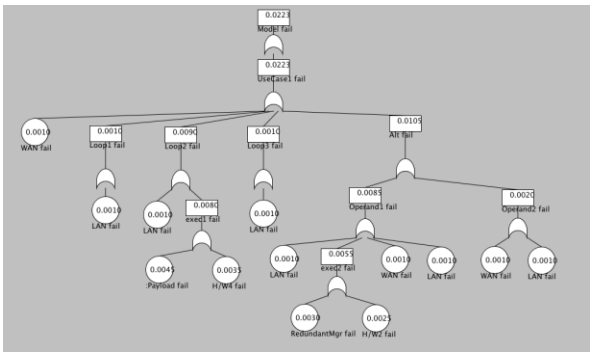


Fig. 8. Generated fault tree of MRS

6. Conclusion

This paper presents a model transformation by using the ATL transformation language, which transforms UML 2.4 software models composed of sequence diagrams, composite structure diagrams, and use case diagrams, along with applied MARTE/DAM stereotypes into Fault Tree Models.

The UML modeling tool used in this work is Papyrus, which is an open-source tool based on the Eclipse environment. One of the reasons we selected Papyrus was that the ATL engine was able to directly accept the UML model extended with profiles produced by Papyrus as the source model. Our ATL transformation model, UML2FT, also accepted the custom target Fault Tree metamodel, which we created with the ECORE tool. Using the Fault Tree metamodel, ATL was able to produce an XMI file of the target model, which could then be processed by a simple JAVA program and then be read and analyzed by an existing fault tree analysis tool. We selected the open-source tool FaultCAT as our Fault Tree analysis tool because it accepts input in an XML format, very similar with the output generated by our transformation. There are more sophisticated fault tree analysis tools available, but the problem is that they have their own input language different from XML, requiring a more complex translation of the generated fault tree. The ATL UML2FT module we have designed, implemented and tested consists of 7 matched rules to handle conditional mappings, and 4 lazy rules for all unconditional mappings. We also used 32 helpers that may be called by transformation rules to achieve the transformation.

The model transformation we produced has some limitations. For example, we generate only regular fault trees as target models, which are supported by the analysis tool we used (FaultCAT). We have not considered extended versions, such as dynamic fault tree introduced by Pai et al., (2002) which extend regular fault trees with dynamic gates (spare gates, priority AND gates, etc.) for the modeling of fault tolerant hardware systems. Extensions to regular fault tree are left as future work.

References

A. Hassan, K. Goseva-Popstojanova, and H. Ammar. (2005). “UML Based Severity Analysis Methodology”, Reliability and Main- tainability Symposium.

Bernardi, Simona, Merseguer, José, Petriu, Dorina C. (2011) : A dependability profile within MARTE. Software and System Modeling. 10(3), pp. 313–336.

Bernardi, Simona, Merseguer, José, Petriu, Dorina C. (2012) : Dependability modeling and analysis of UML-based software systems", ACM Computing Surveys, Vol. 45, Issue 1.

Bernardi, Simona, Merseguer, José, Petriu, Dorina C. (2013). Model- Driven Dependability Assessment of Software Systems, ISBN 978-3-642-39511-6, Springer.

D’Ambrogio, A, G. Iazeolla, and R. Mirandola. (2002) "A method for the prediction of software reliability." Proceedings of the 6-th IASTED software engineering and applications conference (SEA2002), Cambridge, MA.

Domis, Dominik, and Mario Trapp. (2008). "Integrating safety analyses and component-based design." Computer Safety, Reliability, and Security. Springer Berlin Heidelberg, 58-71.

Ganesh J Pai and Joanne Bechta Dugan. (2002). “Automatic Synthesis of Dynamic Fault Trees from UML System Models”, The 13 th International Symposium on Software Reliability Engineering.

Lars Grunske, Bernhard Kaiser. (2005). “Automatic generation of ana- lyzable failure propagation models from component-level failure annotations”, Fifth International Conference on Quality Soft- ware, (QSIC 2005), pp.117-123.

Lauer, Christoph, Reinhard German, and Jens Pollmer. (2011). "Fault tree synthesis from UML models for reliability analysis at early design stages." ACM SIGSOFT Software Engineering Notes 36.1

Wensheng Hu and Zhouhui Deng. (2011). “A Method of FTA Base On UML Use Case Diagram”, Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference.

Web sites:

Web-1: http://www.omgmarte.org. Consulted February 15, 2015

Web-2: http://help.eclipse.org/kepler/index.jsp. Consulted February 15, 2015

Web-3: http://www.omg.org/spec/UML/2.4.1. Consulted February 15, 2015

Web-4: http://wiki.eclipse.org/Papyrus_User_Guide. Consulted February 15, 2015

Web-5: http://www.iu.hio.no/FaultCat. Consulted February 15, 2015

Web-6: http://www.omg.org/news/meetings/workshops/SWA_2007_Presentations/04-2_Harper-Parkinson.pdf. Consulted February 15, 2015