Abstract—Software controls in technical systems are becoming more and more important and complex. Model based safety analysis can give provably correct and complete results, often in a fully automatic way.

These methods can answer both logical and probabilistic questions. In common practice, the needed models must be specified in different input languages of different tools depending on the chosen verification tool for the desired aspect. This is time consuming and error-prone. To cope with this problem we developed the safety analysis modeling language (SAML).

In this paper, we present a new tool to intuitively create probabilistic, non-deterministic and deterministic specifications for formal analysis. The goal is to give tool-support during modeling and thus make building a formal model less error-prone. The model is then automatically transformed into the input language of state of the art verification engines. We illustrate the approach on a case-study from nuclear power plant domain.

I. INTRODUCTION

Analyzing safety related aspects of a system is an important focus in safety critical engineering domains. This becomes even more complicated because of the rising number and complexity of embedded systems. Because more and more functionality is transferred from hardware to software components, it becomes continuously harder to verify safety aspects. Model-based analysis can help solving this problem by finding causal connections between component malfunctioning and overall system hazards based on mathematical deduction techniques [1], [2].

Today, there exist elaborate verification tools (like PRISM\(^1\) or NuSMV\(^2\)) and corresponding formalization logics. In general safety analysis may be qualitative, with two-valued logic or quantitative where hazard probabilities are computed.

Although these verification methods exist, the models which are used for qualitative analysis and those for quantitative analysis have to be built separately. This is potentially very time-consuming and error-prone. To solve the gap between the different modeling languages we proposed the safety analysis modeling language (SAML) in former work [3], [4]. It is a formal tool-independent language which allows the expression of both, qualitative and quantitative aspects. Thus SAML is powerful enough to model software components, hardware components, environment and failure modes. For safety critical systems, all those aspects have to be considered together.

Nevertheless, SAML is a formal language and creating models with only a text editor is a difficult task. Especially for engineers who are familiar with their domain but not with formal safety analysis modeling languages. In the software development domain, special tools (i.e. integrated development environments) greatly assist the developer to produce error-free, well structured and readable program code. The same applies for the creation of formal models.

We propose a combination of the advantages of SAML with the benefits of a user-friendly design environment for model-based analysis. In this paper we present an eclipse based tool called Software-intensive Systems Specification Environment (S\(^3\)E). This tool was created to support developers and engineers in model creation as well as formal analysis of this model. The main features are manifold. First there is an elaborated editor with syntax highlighting, auto completion and error detection for easy model creation. Next there is a simulator to enhance model understanding and debugging. Finally the tool provides a seamless integration of several model checkers and analysis tools.

The rest of the paper is structured as follows: the language SAML is quickly outlined in section II. The case-study of a nuclear reactor control unit for illustration purposes is presented in Section III. Section IV gives a detailed view of working with SAML and the S\(^3\)E tool is presented in section V, followed by a brief summary of further work in section VI and related work in section VII. Finally section VIII gives a summary and conclusion.

II. SAFETY ANALYSIS MODELING LANGUAGE (SAML)

To verify safety aspects in complex systems, especially in software intensive systems, safety analysis can be used. Several safety analysis tools are already developed. Each of them focuses on a specific class of problems and provides a fitting specification language.

Tools can be separated into two classes by the type of question that should be answered. This might be a logical

\[^1\]http://www.prismmodelchecker.org/

\[^2\]http://nusmv.fbk.eu/
question like: “what must happen to cause a critical system failure?”. This is mostly solved by formal verification and model checking techniques [5], [6], [7], [8]. Analysis based on this are called qualitative. On the other hand it is possible to check quantitative aspect like: “what is the probability of a critical system failure?” This is done with stochastic models and quantitative approximations [9], [10], [11], [12].

In practice, both qualitative and quantitative aspects must be considered. Therefore separate system models need to be built in every specification language/tool which has to be used. This is time-consuming and even more important error-prone. To cope with this issue, we developed the tool-independent specification language SAML (Safety Analysis Modeling Language) with an automatic model transformation framework [13] to tool-specific languages of the verification engines.

Entire syntax and semantics of the SAML framework can be found here [3], [13]. A SAML model is basically a set of finite state automata. These are executed with discrete time steps using synchronous parallel transitions. A SAML model contains one or more modules with a set of state variables that are updated according to update rules. State variables are discrete integer values which describe the state of a component.

An update rule consists of an activation condition, followed by a set of discrete probability distributions over the next states. Multiple probability distributions denote non-deterministic specifications. A probability distribution specifies the probabilistic behaviour.

For a SAML model to be valid it is important that activation conditions are formulated in a way, that only one update rule in every module can be active at a time i.e. that all other activation conditions of the same module evaluate to false.

In most cases, the functional behavior is deterministic. Probabilistic modeling is very often well-suited to specify hardware failure occurrence, while the malfunctioning for software components is usually non-deterministic. The same applies to a controlled system, where the control input is specified non-deterministic.

One major aspect that has to be considered when creating a model is the impact of the decision towards nondeterminism. Non-deterministic behavior has a big impact on the calculated probability of quantitative verification. Instead of a distinct probability, only a best case or worst case analysis can be performed.

The verification cannot be done on a SAML model directly as it is not intended to provide its own verification tools. In order to verify a SAML model, it has to be transformed into the input language of the desired analysis tool. This transformation is done automatically on a syntactical level. At the current state there exists converters to NuSMV (for qualitative analysis) and to prism (for quantitative analysis). The output of this transformation is a semantically equivalent model expressed in the input language of the chosen verification tool. A formal proof of the semantic equivalence of the transformed models may be found in [13].

Using automatic model transformations offers two big benefits. Firstly the separation allows switching between different verification engines without touching the model. This allows even the integration of new (bigger/better/faster) verification engines. Secondly it is guaranteed that all analysis are done on the same model. Therefore, in the sense that the same model was used, all results are consistent with each other.

### III. Power Supply Case-Study

In the following a case-study is used to illustrate the usage of SAML. The case study is taken from a work of Pierre-Yves Chaux et. al. [14]. It is a power control unit that is responsible for providing energy for cooling and controlling a nuclear reactor core. Redundancies are used to make the system safe. This is a common solution in safety critical engineering. The main question for safety analysis is the benefit of these redundancies and more important if there is still a single point of failure, that is not caught by these redundancies.

![Component Structure and Power Flow](image.png)

Figure 1 illustrates the different system component. The main power supply is an electric transformer (TR1) and a redundant transformer (TR2). If the main transformer fails, the control unit (DB2) is going to switch to the second one. If this transformer fails too, the second control unit (DB1) will switch from the power supply of the transformers to a diesel engine (diesel) by using switches CB1 and CB2.

Each component in this scenario can fail and also be repaired. While most components can only fail if they are working, the diesel generator can also break down if not in use. The hazard in this scenario is the loss of power, which occurs when DB1 has no energy output. Because the original
case-study did not contain probabilities, we assume a failure rate for each of the seven failures of $10^{-6}$ 1/h.

IV. Model Creation and Analysis of the Case-Study with SAML

A model based safety analysis is done in two steps. First of all, a formal system model has to be created. This includes the functional model as well as failure modes and environmental models. Afterwards, the model can be transformed into the input language of several model checkers to analyze the model with respect to different specifications. For better understanding, each step will be exemplified with the power control case study.

In common for all modeling aspects as well as analyzing the results of the model checker, it is important to be aware of the used time-unit. As SAML relies on Markov decision processes, a step is a discrete time unit, which must be defined prior to the creation of the model and consistent over all modules. Given probabilities must be transformed into the used time-unit of the system using Equation 1.

$$p_{step} = \lambda \cdot \delta t \quad (1)$$

In this example we choose the step size as one minute. A more elaborated discussion about discretization of continuous time probabilities may be found in [13], [15].

A. System Modeling

A formal system model of the power supply system consists of three main parts: the functional model itself, a model of the physical environment and a model of the considered failure modes that may occur. The reason for this is that the safety of a system cannot be decided without considering the working environment and component faults.

1) Functional Model: The functional model describes the desired behavior of the system. This is done by creating modules for each used component with one or more state variables and update rules.

As an example, we are going to create the diesel generator (Diesel) as a SAML module. Because the generator can be in active or dormant mode, we define two states (0 for standby, 1 for running). The requirement for the diesel generator is, that it turns on, when the switch CB2 is closed, and turns off, when the switch is opened. We assume that the switch module contains a state variable CB2_state with also two states (0 for opened, 1 for closed). These requirements lead to the update rules shown in figure 2.

As the functional model is supposed to work appropriately, it does not contain any non-deterministic choices or probabilistic distributions. This also holds for this example: the update rules contain only one choice and only a trivial probability distribution (i.e. probability one).

module diesel
diesel_state : [0,1] init 0;
CB2_state=0 -> choice:(1:(diesel_state'=0));
CB2_state=1 -> choice:(1:(diesel_state'=1));
endmodule

Figure 2. functional model of diesel generator

2) Environmental Model: As most systems are somehow connected with the environment, it is important to represent the influence of this connection inside the system model. As shown in figure 1, the transformers get power from the environment. The failure of this power supply will heavily influence the overall hazard probability as both transformers cannot supply energy. In this case the diesel generator will become the only power supply left. For that purpose it is necessary to also include the power-supply of the environment into the system model.

Modeling environmental behavior is done using the same SAML language aspects as also the functional or the failure models. Especially for environmental modeling, nondeterminism is commonly used. It is hard to determine a probabilistic distribution for this kind of influence. Because there is no special SAML syntax for environmental modeling, there is no need to present another example here.

3) Failure Model: In order to verify safety issues based on failure occurrences, failures have to be included in the model. The specification of failure modes does not necessarily need special language aspects, but can be integrated analogously to other system components.

In addition to that, current work is on providing special predefined failure modes. This is possible because modeling failures in SAML follows the methodology of [16], [3]. The basic idea is to divide the failure model into the failure occurrence pattern and a specific local failure behavior.

The occurrence pattern specifies the type of occurrence. This can be a transient or a persistent failure. Persistent failures only occur once and afterwards stay in the failure state, whereas transient failures can occur at every step.

Also it is possible to describe the failures based on their mode. Per-time failures describe the occurrence of the failure in a given time interval. The per-demand failures describe failures which can only occur if there is a request/demand.

Based on this classification four main patterns can be identified. These are combinations of per-demand/per-time and transient/persistent and can be defined using special keywords in SAML.

Unfortunately the failure behavior itself cannot be described in such a pattern, as it is mostly specific to the actual model. For describing the usage of failures, we will extend the diesel generator example with a failure mode. As this failure can occur in every situation, the pattern is obvious a
per-time failure. Also it has to be a transient failure because of the fact that it should be repairable.

We assume a failure rate of $10^{-6}$/hour. Because our model has a step size of one minute, it is necessary to recalculate the units in order to get meaningful results. Also we are not interested in failure rates but per-step failure probabilities, which can be calculated by $p_{\text{step}} = \lambda \cdot \Delta t$, where $\lambda$ is a failure rate and $\Delta t$ the step size of the model. As a result, the per-step failure probabilities of the case study is $6 \cdot 10^{-5}$.

On the other side, repairing the system is a human task and depends on several issues like knowledge, spare parts and repairing queue. In this case we decide to use a non-deterministic repair time.

constant double $p_{\text{fail}} := 6 \cdot 10^{-5}$;

failuremodule diesel_rf
  diesel_f: [0..1] init 0;
  diesel_f=0 -> choice:( p_{\text{fail}}:( \text{diesel_f}'=1 )
                       + (1-p_{\text{fail}}):( \text{diesel_f}'=0 ) )
  diesel_f=1 -> choice:( 1: ( \text{diesel_f}'=0 )
                        + choice:( 1: ( \text{diesel_f}'=1 ) )

endmodule

Figure 3. manual failure pattern for diesel generator failure

With this information it is possible to create the failure occurrence pattern shown in figure 3. A constant $p_{\text{fail}}$ is used to define the probability. The failure module contains a state variable $\text{diesel_f}$ with the states 0 and 1, where 0 represents no failure and 1 indicates a failure. This is sufficient for most failure patterns but not necessary limited to only two states. The first update rule in this example defines how the failure occurs. With the probability of $p_{\text{fail}}$, the next state of this module will become 1, which indicates a failure, and with the complementary probability $(1-p_{\text{fail}})$ stays at 0.

If the state of $\text{diesel_f}$ is 1 i.e. the failure occurred, the second update rule describes, when the failure is repaired. In this case two choices are used, which represents non-determinism. In this case the model checker will always choose one of the possibilities based on a worst case assumption.

After the creation of the failure occurrence pattern, the specific failure behavior must be implemented by extending the diesel module as shown in figure 4. The already existing update rules are extended by the failure state variable $\text{diesel_f}$. Because the prior model only contained the functional model, this behavior has to be preserved by adding the non-failure state (0) of the failure state variable to the activation conditions. In addition to that, the behavior of the error occurs is modeled by adding a new update rule with the failure state (1) is added to the module. In this example, the diesel generator is always forced to be deactivated (0) while the failure pattern is activated.

module diesel
  diesel_state : [0..1] init 0;
  diesel_f=0 & CB2_state=0
    -> choice:( 1: (diesel_state'=0 ));
  diesel_f=0 & CB2_state=1
    -> choice:( 1: (diesel_state'=1 ));
  diesel_f=1 -> choice:( 1: (diesel_state'=0 ));
endmodule

Figure 4. extended model of diesel generator

Details about formal failure modeling and failure patterns can be found in [16], [17].

B. Safety Analysis

The choice towards a certain set of model checkers that are used depends on the wanted results i.e. logical cause consequence relations and/or probabilistic calculation.

1) Qualitative Analysis: Main goal of qualitative analysis is the cause consequence relations between failure modes and the system hazard (in this case the loss of power). One concrete question for the case study would be: “Can a single defect lead to a breakdown of the full system?”

One method which can be used for this is deductive cause-consequence analysis (DCCA) [5]. A DCCA consists of several questions asked to a qualitative model checker. Indeed it is made by collecting all possible failure modes and combining the failures until this set of failures can lead to the hazard. This is done in several steps. In the first step the model is analyzed under the assumption that no failures occur. Obviously the result of the question, if the hazard can occur, should be false. Otherwise the functional design of the system is incorrect.

In the next step, the system is analyzed under the assumption that only one failure occur at a time. This identifies all single point of failure. Afterwards all combination of two failures are tested, and so on. In the subsequent steps all found critical failure combinations are excluded. This can be done because a set of failures containing a subset that leads to a hazard obviously also lead to the hazard. This procedure is repeated until all possible combinations are tested. Although exponential in theory, monotone arguments bring the effort into polynomial costs in most practical scenarios [18], [2]. All minimal sets of failures that can lead to a system hazard are called minimal cut set.

For the example case-study of the power control unit, DCCA pointed out seven minimal cut sets. In detail, one single point of failure (if DBA1 fails) was detected,

3Note that non-deterministic modeling assumes no knowledge about repair/diagnoses intervals.
four sets with two failures (\{Diesel,DB2\}, \{Diesel,CB1\}, \{DB2,CB2\}, \{CB1,CB2\}) and two combinations with three failures (\{Diesel,TR1,TR2\}, \{TR1,TR2,CB2\}). These results are equal to the ones from the original case-study [14].

2) Quantitative Analysis: After the determination of critical failure mode combinations, a quantitative safety analysis is conducted which computes the overall occurrence probability of the hazard. Obviously this requires known failure rates and failure probabilities of the failures.

One method is probabilistic DCCA (pDCCA) [10]. In the case study the quantitative analysis would answer the question: “What is the overall probability that the power supply fails?”.

V. SOFTWARE-INTENSIVE SYSTEMS SPECIFICATION ENVIRONMENT

A challenge for SAML, as well as for every other developed language, is the acceptance of the user. In case of model-based safety analysis, an user has a high knowledge about the modeled system and safety analysis aspects. He is not necessarily an expert in creating safety analysis models. In order to reach this target group, an elaborate, easy-to-use user interface is needed.

We decided to use eclipse as a well-known development framework. Based on this framework we created a plugin named Software-intensive Systems Specification Environment (S\(^3\)E). This includes a graphical user interface (GUI) with high-level development tools, elaborate editor and integration of project management features (like bug tracker, import and export functionality, outline view and versioning tools).

Additionally model transformation and analyzing is implemented as well as the result visualization of the verification process. The different components of S\(^3\)E will be explained in detail in the following sections.

A. Editor

The main feature for better model development is the editor. We developed an elaborate editor with syntax highlighting as well as syntactical checks. Most errors, which are recognized, were extended with helpful error messages and quick fixes for providing reasonable solutions. A screenshot of the current version of this tool is shown in fig. 5.

B. Analyzer

In order to support the engineer to analyze the model, the S\(^3\)E tool must translate the model automatically into a state-of-the-art model checker input language. Referring to the usage of the eclipse “run” button, we are using a “run analysis” button, which allows selecting the model file as well as the needed specification.

Additionally the model checker verification is also integrated into S\(^3\)E so it is not necessary to use the model checker by hand as it is started automatically in the background. The results of the model checker then is parsed and displayed in an easy to view and understandable way inside the eclipse framework.

C. Model Simulation

In order to provide the user as much tool-support as possible, we also thought about some type of debugger. Because a classic debug functionality as known from programming languages is not possible in the context of formal model analysis. Therefore we decided to create a user controlled simulator.

This simulation enables the user to understand the behavior of the model, by giving him the opportunity to make all non-deterministic and probabilistic decisions over time.

The history of all decisions and the dependent reactions of the systems based on user decisions can be viewed and analyzed, which gives a big benefit if the model seems to work wrong or the results of the model checkers seems to be unreasonable.

VI. NEXT STEPS AND OUTLOOK

The current editor already includes syntactical as well as simple semantically checks. Also most of the detected errors are extended with further comments and helpful suggestions to fix the error. Also the transformations as well as the display of results are already implemented. Current work is on increasing the user-experience by simplifying the interfaces.

Additionally we are working to generate DCCA [18] proof obligations automatically. The algorithm for creating is pretty simple but needs a list of the failure modes and even more important the capability to run the model checker several times with a new set of specification, dependent on the set before. For example the DCCA for the small case-study with seven error modules consists of thirty one specifications that were created in four iterations. The identification of failure modules therefore is quiet simple as the failure modes are already classified in SAML by a special keyword. Also qualitative model checkers are already supported to run automatically. With these preconditions, it is possible to create a DCCA fully automatically by pressing a single button. The results of the DCCA then will be filtered and display in an easy to view way.

Also the SAML language is extended for special failure mode modules to extend the basic failure modeling abilities [17].

Furthermore SAML is developed further to be able to cope with sets of constants and modules. This leads to the specification of model families. The requirements for a system are often antagonistic (for example safety, costs and performance). Usually an engineer tries to make design decisions in such a way that all objectives are considered adequate. Our approach is to leave design decisions open and use multi-objective optimization to identify best compromises. One of the main challenges for the editor towards
optimization is the question of debugging or simulating of sets of models. As a long-term goal we are planning a graphical editor. Using a graphical editor will not only increase the acceptance of the tool but also give a better understanding of the model when displayed as an image. We are currently working on defining the needed graphical notation, which has to be as specific as SAML and for the same time easy to understand by simply looking at it. Here, we are thinking about using UML as a graphical interface. Main benefit is the already well known structure and widely acceptance of this notation for system model creation. Currently we are evaluating the feasibility of this approach.

VII. RELATED WORK

The most known example of a formal language used for the development of safety critical applications is the SCADE suite4, developed by Esterel Technologies. SCADE is based on the synchronous data-flow language LUSTRE. However the included model-checker is not well suited for more complex safety analysis as shown in [2]. It also does not allow probabilistic analysis.

Another example is the language FIACRE [19]. It is included as intermediate language in the TopCased toolkit [20] to verify properties of SysML models that are created in TopCased. Unfortunately the formal models grow very large even for small case-studies [19] because of the large number of supported SysML artifacts.

A recent approach was created for the COMPASS project [21] to combine the already existing FSAP-NuSMV/SA [7] framework with the MRMC probabilistic model checker [22]. This combination allows analyzing models specified in the SLIM language, which is inspired by the Architecture Analysis and Design Language (AADL) and its error extension. The SLIM language describes the architecture and behavior of the system components, it allows for a combination of discrete and continuous (hybrid) variables. From the AADL error annex, exponentially distributed failure modes are supported and the effects can automatically be injected into the system specification. The hybrid behavior of the SLIM models and all internal transitions are removed by lumping and the resulting interactive Markov chain is analyzed with MRMC.

COMPASS is one of the few existing approaches which also combine qualitative and quantitative modeling and analysis. Nevertheless, the design is very much tool-dependent and exchanging the model-checking tools is not possible.

In addition, none of the existing approaches allows for the integration of both per-time and per-demand failure modes as it is possible with SAML and our analysis approach. These two different probabilistic occurrence pattern are described...
in IEC 61508 [23] as high or continuous demand and low demand failure modes.

VIII. CONCLUSION

In this paper we presented a Eclipse-based tool for the Safety Analysis Modeling Language (SAML). This tool, called Software-intensive Systems Specification Environment (S³E), is able to cope with several model checkers by providing an easy to use interface for running the verification and displaying the results inside the tool. Also a model simulator is successfully integrated.

Furthermore we created an elaborated editor with syntactical as well as simple semantically checks. We customized the detected errors with meaningful messages and helpful suggestions.

By integrating the language into a common modeling framework, we were able to benefit from the already existing acceptance of Eclipse. Also the development framework provides tool-independent features like versioning support and project management integration. With this tool we are going to make the SAML method accessible for broader acceptance in industrial contexts.

As a result of the last months we proudly present the first version of the specification environment S³E. After the basic functionality is now available, the next steps will be the improvement of the user experience by adding comfortable tools. The current version of the S³E tool can be downloaded from http://euromover.cs.uni-magdeburg.de/cse/

ACKNOWLEDGMENTS

Michael Lipaczewski is sponsored by the Deutschen Ministerium für Bildung und Forschung in the ViERforES project (BMF, project-Nr.: 01IM08003C).

Simon Struck is sponsored by the German Research Foundation (DFG) within the ProMoSA project.

REFERENCES


