Dependability Modeling of Cyber-Physical Systems in the Gamma Framework

Richárd Szabó, András Vörös
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: richard.szabo@edu.bme.hu, voros.andras@vik.bme.hu

Abstract—Cyber-physical systems (CPS) can be found everywhere: smart homes, autonomous vehicles, aircrafts, healthcare, agriculture, and industrial production lines. CPSs are often critical, as system failure can cause serious damage to property and human lives.

Today’s cyber-physical systems are extremely complex, heterogeneous systems, so rigorous engineering approaches are needed both at design and runtime. On one hand, model-based techniques support the efficient system design, and on the other hand, fault-tolerant middleware and communication technologies support the reliable operation of critical CPS. However, modeling dependability-related system aspects is far from trivial. In this paper, our goal is to show a methodology that introduces design patterns for dependability modeling in the Gamma modeling framework to take a step towards the efficient design of dependable CPSs.

Index Terms—model-based development, cyber-physical system, CPS, formal analysis, dependability

I. INTRODUCTION AND BACKGROUND

Cyber-physical systems (CPS) are complex, heterogeneous systems that contain a wide variety of devices, from sensor networks through edge devices to cloud-based services. CPSs often provide critical services, where the correct and reliable operation is crucial. Model-based techniques support not only the design of critical CPSs, but also formal verification techniques can be applied to system models to find design flaws and stochastic analysis techniques ensure that the system meets the extra-functional requirements. Ensuring dependable operation necessitates the application of fault-tolerant design/architecture patterns. However, model-based tools do not provide support to efficiently design complex dependability-related design solutions, where the services and also allocation is properly modeled.

In this paper, our goal is to make a step towards the design of dependable CPSs by proposing an approach that provides design support for engineers. Our proposed approach is model-driven and relies on an open-source design tool, the Gamma framework. Our long-term goal is to transform Gamma functional architecture models to system models enriched with the chosen design patterns of the services. In addition, our goal is to operate the services by using technologies with built-in dependability mechanisms. In the following, the background of our work is summarized.

A. Dependability

Dependability is the ability of a system to avoid service failures that are more frequent and more severe than is acceptable [10].

Our approach focuses on the following aspects of dependability:

- **availability**: readiness for correct service.
- **reliability**: continuity of correct service.
- **safety**: absence of catastrophic consequences on the user(s) and the environment.

One of the means to ensure the dependability of the system is fault tolerance, which is the ability of the system to avoid service failures in the presence of faults. Our approach applies the following fault tolerance techniques to ensure the dependability of the system:

- **error detection**: identification of the presence of an error.
- **fault masking**: systematic usage of compensation to conceal a possibly progressive failure.

B. Model-driven Development

Model-driven development is a system design approach where the main product of development is the models that describe the system. These models contain valuable information about our system. Using the designed models, we can derive additional models and generate software source code/configurations. By using properly certified code generators, we can ensure the quality of the generated code, thus avoiding errors from human coding, and ensuring the quality of critical systems. We can also derive analysis models from the engineering models.

C. Formal Verification and Dependability Analysis

In our approach, we rely on models as the main artifact of the development. From the developed models we can derive formal/analysis models and run qualitative analysis to find causes of potential hazards and the effects of faults, and quantitative analysis to calculate reliability and availability of the system. Formal verification is a technique to exhaustively examine the behaviour of the systems and prove the error-free behaviour. Dependability analysis is based on stochastic
models of the system: traditional engineering models have to be enriched with fault-related probabilities and environmental assumptions and stochastic analysis is used to ensure dependability measures of the design of the system under the given environmental conditions.

D. Gamma Statechart Composition Framework

Gamma is a statechart composition framework [1] designed to model and verify component-based reactive systems and generate code from the models. The tool supports the hierarchical composition of the statecharts, which enables engineers to focus on subcomponents of the complex systems. The framework raises a new modeling layer over engineering state machines to describe communication between components, making it suitable for modeling complex, hierarchical systems. The tool allows the user to formally verify the models using UPPAAL [2] and Theta [11]. Finally, source code can be generated from the models.

E. Related Work

Various approaches exist in the literature to design dependable systems. The most commonly used architecture design techniques in practice are manual construction of models and using automatic techniques to synthesize full systems from components. During the manual construction, the engineer has to build everything from the ground up using well-defined modeling languages like SysML [5], but has the freedom to use any components, technologies and algorithms, at the cost of having to implement them. Automated techniques on the other hand like ArcheOpterix [7], can provide optimized architecture, but at the cost of only building it from a set of predefined set of elements.

In [6] a UML profile is defined to aid the dependability analysis of real-time systems. The DAMRTS (Dependability Analysis Models for Real-Time Systems) profile supports the modelling of the probabilistic aspects of systems defines a transformation of UML models to probabilistic timed automata.

A more comprehensive summary of the tools and algorithms can be found in [12].

II. OVERVIEW OF THE APPROACH

This section details our approach for the dependable design of component-based systems. We identified the following steps that should be supported by the Gamma tool and the planned transformations.

1) Functional architecture: The engineer designs the functions/services of the system using the Gamma Statechart Composition Framework. In this step, the focus is on the intended functions/functionalities. The output of this step is the function definitions as statecharts and the functional architecture.

2) Environment modeling: In the second step, the engineer enriches the model with the environment information. The environment is described by stochastic events or in the case of a complex environment, stochastic behaviour models provide the inputs (and faults) for the functional model. The environment can include various inputs from the world outside of the system, possible failure causes for both hardware and software components. E.g. hardware failures can be caused by overheating from a heatsource outside of the system.

3) FMEDA: In the third step, using the functional model and the environment model the engineer manually performs the Failure modes, effects, and diagnostic analysis (FMEDA) to determine the error propagation and compute the effects. The output of the analysis shows which components of the system require fault-tolerant patterns to meet the extra-functional requirements.

4) Applying dependability design patterns: In the fourth step, the engineer can apply design patterns based on the output of the FMEDA. Our goal is to support this step: we plan to provide a library of fault-tolerant patterns and corresponding transformations. At first, according to the FMEDA, the engineers have to choose the design pattern to be applied on the (critical) functions. The tool will transform the model and apply the design pattern. In addition, when the design pattern has to be fine-tuned for the domain, the engineer can modify the result accordingly by modifying/redefining the logic itself. Our idea is not only to transform the model, but also to define the constraints for deployment as annotations on the model. Engineers can also configure the deployment by changing the annotations.

5) Verification and extra-functional analysis: In the fifth step, the verification and analysis can be performed. Formal verification can be applied in two phases: at first, the functional architecture model has to be verified for providing the intended functionalities. Then, the dependable functionalities and error propagation in the system can also be verified by the advanced techniques provided by Gamma. The dependability model of the system containing the environmental and fault information is analyzed by the stochastic analysis algorithms in the stochastic Gamma [9]. When deployment is also provided, the whole system can be evaluated from the dependability point of view.

After the final step if the results of the analysis is not acceptable the process can be repeated from Step 4. or a total redesign of the system is needed.

III. DESIGN PATTERN LIBRARY FOR DEPENDABILITY

During the design of the system, architectural design patterns can be used to ensure the dependability of the system. The following design patterns were chosen to apply the fault tolerant techniques and aspects presented in Section I-A.

Our plan is to provide a library supporting some of the well-known design patterns for error detection and fault-tolerant behaviour [4]. In order to reach the target dependability measures, the applied design patterns must also follow deployment constraints such as that variants must be deployed on separate devices with similar capabilities. In addition, different
technologies can be used to increase the fault tolerance of the system, but those can add additional constraints during the system design (e.g. Kubernetes [3] works with distributed systems, thus only asynchronous components can be used in the system).

In the following, each pattern is presented on an example seen in Fig. 1. The engineer designs a part of the system which contains three components connected to each other. The FMEA analysis shows that the component in the middle (Component_2) is a single-point-of-failure, so the engineer decides to use a pattern to increase the fault tolerance of the system.

The fault-tolerant patterns provided by the library are the following:

<table>
<thead>
<tr>
<th>Category</th>
<th>Fault-tolerant pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error detection</td>
<td>Two-channel architecture with comparison</td>
</tr>
<tr>
<td>Error detection</td>
<td>Two-channel architecture with safety checking</td>
</tr>
<tr>
<td>Fault tolerance (HW)</td>
<td>N-modular redundancy</td>
</tr>
<tr>
<td>Fault tolerance (SW)</td>
<td>N-version programming</td>
</tr>
<tr>
<td>Fault tolerance (SW)</td>
<td>Recovery blocks</td>
</tr>
</tbody>
</table>

A. Error Detection Patterns

1) Two-channel Architecture with Comparison: The two channels work on shared inputs, with comparison of the outputs. This pattern provides high error detection coverage, but can have increased detection latency.

- **Parameters**: Tolerance for accepting outputs as equal.
- **Constraints**: The channels must be deployed separately to avoid common mode faults of the hardware components, which could cause the faulty behaviour of both channels.

In this example the engineer annotates Component_2 with Two-channel Architecture with Comparison (2CC) from the library as seen in Fig. 2. After the annotation, model transformation generates the components defined in the pattern. The 2CC pattern defines a second channel of the annotated software component, a DataMultiplier to share the data between the channels and a Comparator to compare the data from the channels and raise an error signal if the output is outside the acceptance range.

2) Two-channel Architecture with Safety Checking: This pattern provides an independent channel for safety checking.

- **Parameters**: Explicit safety rules.
- **Constraints**: -

In this example the engineer annotates Component_2 with Two-channel Architecture with Safety Checking (2CS) from the library as seen in Fig. 3. After the annotation, model transformation generates the components defined in the pattern. The 2CS pattern defines a safety channel to the annotated software component, a DataMultiplier to share the data between the channels and a Safety Checker to check the data from the channels and raise an error signal if the output is outside the acceptance rules. The generated safety channel is only a skeleton, the safety engineer must implement the logic of the channel according to the parameters.

B. Fault tolerance Patterns for Permanent Hardware Failure

1) N-modular Redundancy: This pattern provides fault tolerance by masking the failure with majority voting.

- **Parameters**: Number of modules.
- **Constraints**: The modules must be deployed separately to avoid common mode faults.

In this example the engineer annotates Component_2 with N-Modular Redundancy (NMR) from the library similarly as seen in Fig. 4. After the annotation, model transformation generates the components defined in the pattern. The NMR pattern defines the replication of the annotated component N times (where N is the parameter given by the engineer), a DataMultiplier to share the data between the replicas and a Voter to collect the data from the replicas and decide the output with majority vote. The logic of the generated Voter can be modified to achieve more complex fault tolerance.

C. Fault tolerance Patterns for Software Failure

1) N-version Programming: This pattern provides active redundancy by using multiple software modules with diverse implementations, algorithms and programming languages.

- **Parameters**: Number of variants, explicit acceptance rules.
- **Constraints**: The modules must be deployed separately to avoid common mode faults.
In this example, the engineer annotates SW Component 2 with N-Version Programming (NVP) from the library as seen in Fig. 5. After the annotation, model transformation generates the components defined in the pattern. The NVP pattern defines the replication of the annotated software component N times (where N is the parameter given by the engineer), a DataMultiplier to share the data between the replicas and a Voter to collect the data from the replicas and decide the output with majority vote and to provide error signal if the output is outside the acceptance range. The logic of the generated Voter can be modified to achieve more complex fault tolerance.

Fig. 5. Model transformation for N-Version Programming.

2) Recovery Blocks: This pattern provides passive redundancy with multiple software modules and acceptance checking. The next module is executed only if the previous fails on the acceptance check.

- **Parameters**: Number of modules, explicit acceptance rules.
- **Constraints**: -

In this example, the engineer annotates SW Component 2 with Recovery Blocks (RB) from the library as seen in Fig. 6. After the annotation, model transformation generates the components defined in the pattern. The RB pattern defines the replication of the annotated software component N times and Checker N times (where N is the parameter given by the engineer) in a chain, a DataMultiplier to share the data between the replicas. If the output of a replica is outside the acceptance range the next replica is executed, if there are no more replicas an error signal is sent.

Fig. 6. Model transformation for Recovery Blocks.

IV. CONCLUSION

In this paper, we presented a design approach to design dependable CPSs and we defined a library of dependability-related design patterns to aid the design process. In the future, we plan to implement the approach and evaluate on industrial case studies. In addition, the formerly introduced deployment modeling approach [8] will also be integrated.

REFERENCES