A UML Profile for Automatic Code Generation of Optimistic Graceful Degradation Features at the Application Level

Lars Huning, Padma Iyenghar and Elke Pulvermueller
Institute of Computer Science, University of Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany

Keywords: Adaptive Systems, Code Generation, Embedded Software Engineering, Embedded Systems, Functional Safety, Graceful Degradation, Model-Driven Development.

Abstract: Safety standards such as ISO26262 or IEC61508 recommend a variety of safety mechanisms for the development of safety-critical systems. One of these mechanisms is graceful degradation, which aims to provide a degraded service of an application after an error has occurred. While several safety standards recommend graceful degradation, they do not provide any concrete development or implementation assistance. This paper employs model-driven development to realize such an automated approach for optimistic graceful degradation, which is a specific variant of the graceful degradation safety mechanism. We introduce a UML profile that may be used to model optimistic graceful degradation at the application level within a UML class diagram. We leverage this model representation to automatically generate productive source code that is capable of optimistic graceful degradation. This source code is generated without requiring any additional developer actions.

1 INTRODUCTION

The size and complexity of software in safety-critical embedded systems is growing increasingly (Trindade et al., 2014; Penha et al., 2015). Safety standards, such as IEC61508 (IEC61508, 2010) or ISO26262 (ISO26262, 2018), recommend development practices and safety mechanisms to realize such systems. They propose *model-driven development* (MDD) as one alternative to deal with the rising complexity, an idea that is increasingly adopted by the industry (Laplante and DeFranco, 2017; Trindade et al., 2014; Penha et al., 2015). While safety standards offer guidelines on the use of safety mechanisms, they do not provide any concrete development assistance for their realization. This paper aims to fill this gap, by providing a model representation and automatic code generation for one specific safety mechanism: *graceful degradation*.

In order to apply graceful degradation to a system, it has to be composed of multiple states. In case an error occurs, a graceful degradation mechanism ensures that the system switches to a safe state. In contrast to approaches that rely on redundancy, the newly assumed state is often degraded, i.e., it provides a service of lower quality than the original state (Saridakis, 2005). On one hand, this may result in stopping a service in a system that provides several services, e.g., dropping error logging functionality in order to use the underlying hardware to assume system control functionality in case the original system control hardware failed (IEC61508, 2010). On the other hand, the erroneous service may be replaced by another with lower quality. As an example, we consider the position sensor of an elevator, which measures the distance to the next floor. If this sensor fails, another module may continuously inform the elevator control software to always use the slowest speed (Shelton and Koopman, 2004).

Previous research on graceful degradation (cf. section 5) has not only investigated how to select the most useful degraded state given some optimization criteria, but also provided design patterns (Saridakis, 2009) and a meta-model (Penha et al., 2015). However, there remains an open research challenge. It concerns the automatic generation of productive code for graceful degradation, that includes automatic state transitions in case of an error (Penha et al., 2015). In this paper, we propose a solution for this research challenge based on the *Unified Modeling Language* (UML) (OMG UML, 2017) by introducing the following novel ideas:

a) a model representation based on UML stereotypes for specifying which classes inside a UML class

Huning, L., Iyenghar, P. and Pulvermueller, E.
A UML Profile for Automatic Code Generation of Optimistic Graceful Degradation Features at the Application Level.
DOI: 10.5220/0008949803360343
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 336-343
ISBN: 978-989-758-400-8; ISSN: 2184-4348

diagram are capable of graceful degradation.

b) a model representation based on UML stereotypes to specify which objects may be used to provide degraded services in case of an error.

c) a proof of concept code generation based on the novel model representations of contribution a) and b).

The remainder of this paper is organized as follows: Section 2 presents our system model and an existing design pattern for graceful degradation that serves as the basis for our solution. Section 3 introduces our model representation to specify graceful degradation at the UML class level, while section 4 provides a proof of concept code generation for the developed profile. In Section 5 we present related work, before we conclude the paper in section 6.

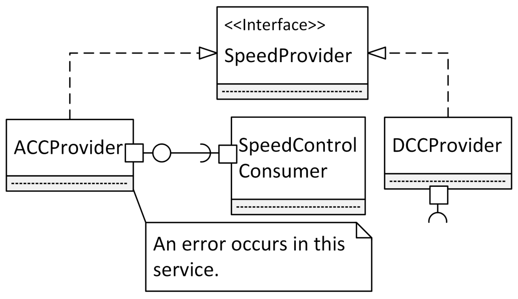
2 BACKGROUND

This section presents background knowledge on graceful degradation and how this concept may be applied to our system model (cf. section 2.1). Furthermore, we summarize a specific design pattern for graceful degradation (cf. section 2.2). This design pattern is the basis for our approach at the code-level.

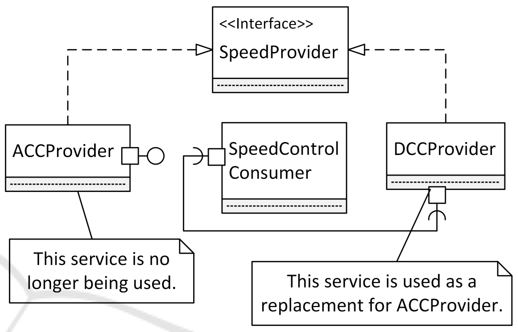
2.1 Graceful Degradation and System Model

This section describes the concept of graceful degradation and shows how it may be applied to the system model used in this paper. We start with the following definition of graceful degradation: “a smooth change of some distinct system feature to a lower state as a response to errors” (Saridakis, 2009, p. 69). Lowering the state may be achieved by either removing an erroneous component from the system or replacing it with a pre-existing spare component that provides lower quality (Saridakis, 2005).

Figure 1 shows an example of graceful degradation by employing a speed control system of an automobile as an application example (Penha et al., 2015). In this example, an automobile has two different driving modes: *Dynamic Cruise Control* (DCC) and *Adaptive Cruise Control* (ACC). In DCC, the system automatically controls the throttle and brake of the vehicle in order to maintain a steady speed set by the driver. In ACC (cf. figure 1(a)), the system controls the throttle and brake of the vehicle in order to maintain a steady speed while also maintaining a safe distance to the vehicle in front. The ACC system depends on a radar to measure the distance to the vehicle



(a) Original system state, where the speed control consumer gets its data from the ACC provider. (UML 2.5 class diagram).



<<Interface>>
SpeedProvider
ACCProvider
SpeedControl Consumer
DCCProvider
This service is no longer being used.
This service is used as a replacement for ACCProvider.

(b) Degraded system state, in which the ACC provider has been replaced by the DCC provider (UML 2.5 class diagram).

Figure 1: Basic concept of graceful degradation exemplified by a speed control system example.

in front. In case there is an error in the radar, the ACC functionality needs to be disabled and the system has to switch gracefully to DCC (cf. figure 1(b)).

The concept of graceful degradation may be realized at different levels of the system. For example, the job of an erroneous, safety-critical hardware module may be taken up by another suitable hardware module that performs non safety-critical operations (IEC61508, 2010). Another example may consider rescheduling the tasks to be performed on the CPU in case computing resources are exhausted (Gonzalez et al., 1997). In this scenario, safety-critical tasks gain priority during scheduling at the cost of other, non safety-critical tasks.

The two previous examples either concern the hardware of the application or may be applied without specific application knowledge. In contrast to this, our approach focuses on graceful degradation at the software application level. It assumes that the application consists of several software components that interact with each other to fulfill the application’s specification.

In the context of this paper, we use the following system model: the (software) application runs on a single machine, i.e., we do not consider distributed systems. Additionally, in the context of this paper, a component consists of one or more object-oriented classes. A single class in each component is responsible for communication with other components. This class is often referred to as an "interface" in the literature (Avizienis et al., 2004). However, the approach presented in this paper heavily uses the object-oriented programming concept of an interface. In order to differentiate between these two terms, we refer to the class that is responsible for communication with other components as the *boundary class* of each component, while the term *interface* will be used to refer to the object-oriented programming concept.

In this paper, we use two categories of components. *Providers* are components that provide some sort of functionality or service that may be used by other components. At the implementation level, providers implement one or more interfaces. *Consumers*, on the other hand, require and utilize the services made available by the providers. At the implementation level, consumers make use of the interfaces implemented by the providers. Consumers and providers communicate through virtual channels called *bindings* (Saridakis, 2005). In this paper, we represent a binding at the model level via UML ports (cf. section 3) and at the code-level via reference variables (cf. section 4). Then, in the context of this system model, graceful degradation may be achieved as follows: if a (runtime) error in a provider has been detected, the consumers may either interchange this provider with another provider that implements the same interface (albeit at a lower quality) (Shelton and Koopman, 2004) or stop using any services offered by this provider (Saridakis, 2005).

2.2 A Design Pattern for Optimistic Graceful Degradation

This section briefly summarizes a specific design pattern for graceful degradation (Saridakis, 2009). This pattern is the basis of our software architecture used for automatic code generation of graceful degradation presented in section 4. Besides the components that make up the application, the pattern consists of three notable software entities:

- A *notifier* is responsible for signaling that an error has occurred inside a component. There may exist many notifiers inside the system. For example, each component may contain its own notifier in case it detects an error. Another approach may use concurrent notifiers that observe a component independently.
- An *assessor* is responsible for determining which components, other than the erroneous component, may be affected by the error. Notifiers signal the occurrence of an error to the assessor.
- One or more *loaders* are employed to actually degrade the affected components to a lower state. The loaders gain their information which components are affected from the assessor. In this paper, the loaders are located inside the boundary class of each component.

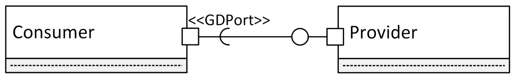
In (Saridakis, 2009), this pattern is further refined into an optimistic, pessimistic and causal variant. They describe different strategies for the assessor to decide which components should be removed. The purpose of the pessimistic and causal variant is to reduce the runtime overhead of executing the loaders multiple times in case an error propagates through the system. In our approach, the runtime overhead for the loaders amounts to changing the value of a reference variable (cf. section 4) and is comparably small to other alternatives. Thus, the runtime overhead of running the loaders several times in a row is also small. Therefore, we focus only on optimistic graceful degradation, in which only the component directly affected by the error is replaced or removed. Future work may also include the other two variants, which may be achieved by including an appropriate tagged value in the stereotypes presented in section 3 and subsequently modifying the code generation for the assessor.

3 MODELING OPTIMISTIC GRACEFUL DEGRADATION FOR CODE GENERATION

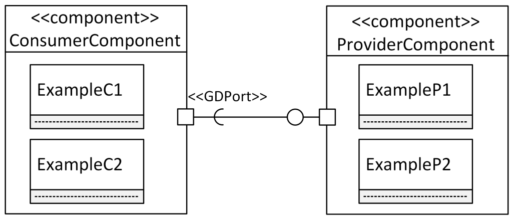
In order to model optimistic graceful degradation for automatic code generation, two key challenges need to be solved. The first challenge concerns how a component may be marked as intended for graceful degradation. The second challenge concerns how the state decrements may be modeled within the application model.

3.1 Marking a Component as Capable of Graceful Degradation

The first part of a model representation for optimistic graceful degradation is to specify how selected components may be marked as capable of graceful degradation. In this paper, we examine two alternatives based on UML how this may be achieved: via UML class diagrams or via UML component diagrams.



(a) Model representation via UML class diagram.



(b) Model representation via UML component diagram.

Figure 2: Alternative model representations for specifying that graceful degradation should be added to a component during automatic code generation.

3.1.1 Specifying Graceful Degradation via UML Class Diagrams

This section describes an alternative to specify that a component is capable of graceful degradation via UML class diagrams. It marks the boundary class of the component, in order to specify that the whole component is capable of graceful degradation. We choose the boundary class to represent the whole component, because it is responsible for interacting with other components, and graceful degradation in this paper entails modifying these interactions (cf. section 4).

An instance of a boundary class may have multiple bindings to other components. In order to minimize the effect of errors on the system, each binding should be treated separately by the degradation mechanism. This prevents the removal or replacement of bindings that are unaffected by the error (Saridakis, 2005). In UML, the metaclass "Port" is one viable alternative for specifying the bindings between objects. A port which is applied to a class may specify an interface that this port either requires or fulfills. This is analogous to the concept of consumer and producer components (cf. section 2.1). In general, a single UML port may be used to represent more than one binding and more than one interface. In our solution, we assume that only a single binding and interface is specified per port. This does not influence the generality of our solution, as additional bindings may be specified by adding additional ports to the boundary class.

In order to mark a port as being capable of graceful degradation, we employ UML stereotypes. Stereotypes are an inbuilt feature of UML and may be used to extend any metaclass in the UML metamodel (OMG UML, 2017). We introduce the novel stereotype «GDPort», which extends the UML metaclass "Port". Applying this stereotype to a port represents that the class to which the port is applied is capable of graceful degradation. Additionally, the «GDPort» stereotype also contains specific tagged values related to graceful degradation that are further described in section 3.3.

Figure 2(a) shows how the model representation based on the «GDPort» stereotype looks like in a UML class diagram. The boundary class `Consumer` requires an interface that is provided by the class `Provider`. The «GDPort» stereotype is applied to the port of the boundary class of the consumer. We mark consumers as capable of graceful degradation instead of provider components, because a provider may already be in an erroneous state when degradation is required. Thus, from a safety perspective, degradation should be performed by consumers instead of providers.

3.1.2 Specifying Graceful Degradation via UML Component Diagrams

This section describes an approach for marking a component as capable of graceful degradation based on UML component diagrams. Similar to class diagrams, components in a component diagram are shown as rectangles and may contain ports that connect a component to another component. Classes that a component contains are shown within the rectangle that represents the component. Besides classes, ports may also be applied to components, thus the «GDPort» stereotype introduced in section 3.1.1 may be reused in this scenario. This is shown in figure 2(b). Two components, (ConsumerComponent and ProviderComponent), are connected via their respective ports. The «GDPort» stereotype is applied to the port of ConsumerComponent. As a component encompasses all its classes, there is no need to specifically mark the boundary class.

3.2 Specifying Degradation Fallbacks for State Decrement

Besides specifying that a component is used for graceful degradation (cf. section 3.1), it is also necessary to specify which providers may be used as potential fallbacks for a consumer component in case of an error. These fallbacks may not be specified within the «GDPort» stereotype, as the potential providers of a consumer may be different for each instance of the consumer. Thus, the fallbacks have to be specified at the instance level, where the bindings between indi-

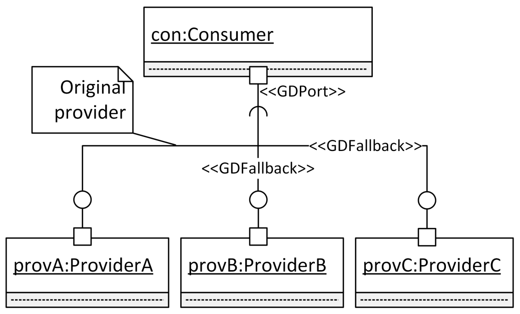


Figure 3: Model representation for specifying degradation fallbacks (UML 2.5 object diagram). The «GDPort» stereotype in this object diagram only serves an illustrative purpose, signifying that the class `Consumer` has been marked with the «GDPort» stereotype. All instances of `Consumer` share the same values for the tagged values of «GDPort».

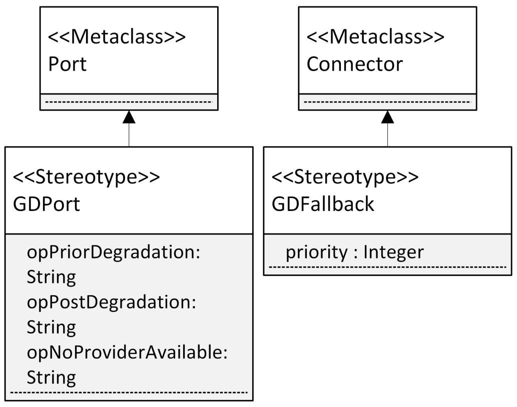

<<Metaclass>> Port
<<Metaclass>> Connector
<<Stereotype>> GDPort
opPriorDegradation: String
opPostDegradation: String
opNoProviderAvailable: String
<<Stereotype>> GDFallback
priority : Integer

Figure 4: UML 2.5. profile diagram for representing graceful degradation in UML models.

vidual instances may be visualized within a UML object diagram. Such binding between ports are called UML *connectors*. They link one port to another, specifying which provider a specific consumer should use.

In order to specify degradation fallbacks, we introduce the novel «GDFallback» stereotype. It extends the UML metaclass "Connector" and thus offers a visual representation of the fallbacks that is directly visible in the model. This approach is shown in figure 3. The instances provB and provC are connected to the port of the instance con via their respective port. The connection link between these ports is marked with the «GDFallback» stereotype.

In order to determine the order in which the fallbacks are used, a tagged value may be included in the «GDFallback» stereotype that indicates the priority of the fallback (cf. section 3.3). Ties may either be broken arbitrarily, or detected and removed by a simple model validation check prior to code generation.

A drawback of this approach is its potential ambiguity during automatic code generation. In general, a port should contain only a single connector to another port. Multiple connectors provide ambiguity as to which connector should actually be used at the start of the program. This drawback is easily solved by parsing the information from any connectors with the «GDFallback» stereotype and removing them prior to code generation.

3.3 UML Profile for Automatic Code Generation of Optimistic Graceful Degradation

This section formalizes the results for modeling application-level graceful degradation described in section 3.1 and section 3.2 inside a UML profile representation. A UML profile is used to group stereotypes which are defined for a specific purpose. The profile is shown in figure 4 and consists of two stereotypes: «GDPort» and «GDFallback».

The «GDPort» stereotype may be applied to the metaclass "Port" and is intended to be applied to the boundary class of a consumer component. As ports may be applied to classes, as well as components, this enables the use of both alternatives presented in section 3.1. The tagged values of the «GDPort» stereotype allow to execute custom actions during certain points of time within a state decrement, as proposed by (Penha et al., 2015). The actions may be executed at the following points in time: 1) prior to degradation, 2) after degradation and 3) in case an operation from an interface is called when no functioning provider is available any more. Case three may occur when the original provider, as well as every fallback provider, have been identified as erroneous. In this case, a default operation may be specified, e.g., to stop the service. Note that the tagged values only specify the names of operations within the boundary class to which the port is applied. The actual source code for these methods has to be added manually by the developer within the boundary class.

The stereotype «GDFallback» may be applied to the metaclass "Connector", which is the metaclass of the link connecting two ports. It specifies the fallbacks to be used by the port marked with the «GDPort» stereotype. As it is only intended for this specific purpose, it should only be applied to connectors of which a single end connects to a port that is marked with the «GDPort stereotype». This may be enforced by a simple model validation check prior to code generation. The tagged value of «GDFallback» indicates in which order the fallbacks are used in case of degradation (cf. section 3.2).

4 PROOF OF CONCEPT: CODE GENERATION FROM THE PROFILE

This section briefly examines how actual, productive source code may be generated from the profile shown in section 3.3. While code generation for self-adaptive systems has already been covered by several approaches (e.g., Morin et al. (2009); Fleurey et al. (2009)), these approaches largely rely on the use of virtual machines to achieve adaption via reflection mechanisms. Embedded systems, in contrast, are typically written in C or C++ which do not offer these features. An alternative employs v-table mappings to reconfigure method call bindings during runtime (Saridakis, 2004). However, as noted by the authors, this approach may result in runtime exceptions in case there remain no non-erroneous fallback providers. Catching these exceptions requires manual implementation changes in the application model. Our approach, in contrast, enables developers to specify a specific operation that is executed in case there exists no more non-erroneous fallback provider (via the tagged value `opNoMoreProvidersAvailable` shown in figure 4).

Thus, we choose to present yet another alternative, which is based on the reassignment of reference variables. For this, we employ the following mappings of model elements to source code:

- A port tagged with the «GDPort» stereotype in the application model is realized as its own class X. This class is instantiated by the consumer of the port.
- Within the class X, there is a reference variable y pointing to the corresponding interface that is offered by the port.
- The reference variable y may be set to any other provider via a specific setter, as long as the provider fulfills the interface of the corresponding port.
- The class x also contains an array of references to the alternative providers that have been marked with the «GDFallback» stereotype in the application model.

In the pattern described in section 2.2, the assessor is responsible for triggering the degradation step. It may be realized as a global singleton class that contains references to every boundary class in order to trigger the degradation of each respective boundary class. For this, the assessor needs a reference to each boundary class. This is realized by introducing the interface `GDConsumer` that each boundary class of a consumer has to implement. The interface contains an error(uintptr_t) method, that triggers the boundary class to check whether it has been affected by an error and to degrade gracefully in case it has been affected. As we only consider graceful degradation on a single machine in this paper, this check may be realized by comparing the machine address of an erroneous provider with the current provider of a consumer. In distributed systems, the providers would have to be distinguishable by some other kind of unique identifier.

Notifiers, which are responsible for informing the assessor of an error within the application, are able to access the assessor due to its global nature. The notifiers itself may be generated by approaches such as described in (Trindade et al., 2014; Huning et al., 2019).

The last entity of the pattern presented in section 2.2 are the loaders, which are responsible for performing the actual degradation step. If the port classes are generated as described above, they take the role of a loader by invoking the setter method to assign one of the alternative providers as the current provider. In case the boundary class contains any methods whose names match the value of one of the tagged values in the «GDPort» stereotype, these methods are executed at the relevant point in time of the degradation step.

Listing 1 shows the generated code for the loaders. Lines 4-15 contain the code for the generated port, along with the current provider (line 9) and the fallbacks (line 10), as well as the respective setters (line 12-14). The port is instantiated by the surrounding boundary class (line 17).

```
 1  class X : GDConsumer{
 2    class PortY : InterfaceY{
 3      void error(uintptr_t adr){
 4        //check port instance
 5        //if affected by failure
 6      }
 7      InterfaceY* current;
 8      InterfaceY* fallbacks[1];
 9      setCurrent(InterfaceY val);
10      setFallback(InterfaceY val,
11        int pos);
12    };
13    PortY portY;
14    //User-defined variables
15    //and methods
16  };
```

Listing 1: Excerpt of a consumer's boundary class, showing the code level realization of a port.

5 RELATED WORK

Related work on graceful degradation includes approaches that study the optimal distribution of programs on available hardware platforms (Becker and Voss, 2015; Nace and Koopman, 2001), as well as approaches that aim at optimizing the use of computing resources in resource-limited situations (Glass et al., 2009; Gonzalez et al., 1997). Furthermore, graceful degradation at the system level of an individual hardware platform has been studied (Schirmeier et al., 2011). None of these approaches consider graceful degradation at the application-level which is targeted by our approach. Approaches that consider graceful degradation at the application level include (Saridakis, 2009, 2005, 2004; Shelton and Koopman, 2004). However, they do not consider MDD or automatic code generation for their graceful degradation features. Nonetheless, we build upon the design pattern presented in (Saridakis, 2009) in this paper (cf. section 2.2).

A group of graceful degradation approaches for fail-operational systems in automobiles has been presented in (Penha et al., 2015; Hussein et al., 2017). In contrast to the previous approaches they consider model-driven development and provide partial code generation. However, they exclude the code generation of the actual degradation step and classify this as a further research challenge. Our paper provides a solution to this research challenge.

The approach presented in this paper only considers error handling and omits the automatic generation of error detection mechanisms via MDD. This research topic has been partially covered by (Trindade et al., 2014; Huning et al., 2019) and these approaches may be used in conjunction with the approach presented in this paper.

The automatic generation of graceful degradation has also been studied from a theoretical point of view (Lin et al., 2019). However, they note that their approach is limited to small and medium scale applications.

Graceful degradation itself belongs to the field of self-adaption. Using MDD to enable self-adaption has been studied by several authors, e.g., (Morin et al., 2009; Fleurey et al., 2009). However, their approaches assume that the hardware is capable of running a Java Virtual Machine in order to use reflection mechanisms for self-adaption (Fouquet et al., 2012). Embedded systems typically do not provide the required computing resources for this. Additionally, safety standards only allow static memory allocation in safety-critical applications (IEC61508, 2010), which is also not considered by these approaches.

The general idea of extending application models with additional features by invoking MDD has also been explored in other domains. For example, a model-based framework for the validation of timing requirements in embedded systems has been proposed in (Noyer et al., 2016). Other approaches provide model-based tool support for energy-aware scheduling (Iyenghar et al., 2016; Iyenghar and Pulvermueller, 2018).

6 CONCLUSION

This paper proposes an MDD-based approach to automatically add the safety mechanism graceful degradation to software applications. Graceful degradation aims at providing continued service of the application in the presence of errors. It is highly recommended for several safety integrity levels by safety standards such as ISO26262 and IEC61508. Our approach introduces new UML stereotypes that may be applied to a UML application model. By parsing the information of these stereotypes, our approach may automatically transform the input model to include (optimistic) graceful degradation features. Then, automatic code generation of common MDD tools may be used to obtain productive source code from this model. This source code is capable of automatically performing optimistic graceful degradation once an error has been detected. Our approach does not require any manual developer interactions besides applying the newly introduced stereotypes and configuring their tagged values to the application's requirements.

Future work may extend our approach to distributed systems. Additionally, we only cover graceful degradation at the software level. (Semi-) automating hardware graceful degradation may be another area for future work, as well as the inclusion of other non-functional requirements, such as real-time requirements.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Economics and Technology (Bundesministeriums fuer Wirtschaft und Technologie-BMWi) within the project "Holistic model-driven development for embedded systems in consideration of diverse hardware architectures" (HolMES).

REFERENCES

Avizienis, A., Laprie, J. C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Becker, K. and Voss, S. (2015). Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 110–118.

Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., and Jézéquel, J.-M. (2009). Models in software engineering. chapter Modeling and Validating Dynamic Adaptation, pages 97–108. Springer-Verlag, Berlin, Heidelberg.

Fouquet, F., Morin, B., Fleurey, F., Barais, O., Plouzeau, N., and Jézéquel, J.-M. (2012). A dynamic component model for cyber physical systems.

Glass, M., Lukasiewycz, M., Haubelt, C., and Teich, J. (2009). Incorporating graceful degradation into embedded system design. In *Design, Automation and Test in Europe Conference Exhibition*, pages 320–323.

Gonzalez, O., Shrikumar, H., Stankovic, J. A., and Ramamritham, K. (1997). Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *IEEE 32nd Real-Time Systems Symposium*, pages 79–89.

Huning, L., Iyenghar, P., and Pulvermueller, E. (2019). UML specification and transformation of safety features for memory protection. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, Heraklion, Crete, Greece. INSTICC, SciTePress.

Hussein, M., Nouacer, R., and Radermacher, A. (2017). Safe adaptation of vehicle software systems. *Microprocessors and Microsystems*, 52.

IEC61508 (2010). IEC 61508 Edition 2.0. Functional safety for electrical/electronic/programmable electronic safety-related systems.

ISO26262 (2018). ISO 26262 Road vehicles – Functional safety. Second Edition.

Iyenghar, P. and Pulvermueller, E. (2018). A model-driven workflow for energy-aware scheduling analysis of IoT-enabled use cases. *IEEE Internet of Things Journal*.

Iyenghar, P., Wessels, S., Noyer, A., and Pulvermueller, E. (2016). Model-based tool support for energy-aware scheduling. In *Forum on Specification and Design Languages*, Bremen, Germany.

Laplante, P. A. and DeFranco, J. F. (2017). Software engineering of safety-critical systems: Themes from practitioners. *IEEE Transactions on Reliability*, 66(3):825–836.

Lin, Y., Kulkarni, S., and Jhumka, A. (2019). Automation of fault-tolerant graceful degradation. *Distributed Computing*, 32(1):1–25.

Morin, B., Barais, O., Nain, G., and Jezequel, J.-M. (2009). Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA. IEEE Computer Society.

Nace, W. and Koopman, P. (2001). *A Product Family Approach to Graceful Degradation*, pages 131–140. Springer US, Boston, MA.

Noyer, A., Iyenghar, P., Engelhardt, J., Pulvermueller, E., and Bikker, G. (2016). A model-based framework encompassing a complete workflow from specification until validation of timing requirements in embedded software systems. *Software Quality Journal*.

OMG UML (2017). OMG Unified Modeling Language Version 2.5.1. Technical report, Object Management Group.

Penha, D., Weiss, G., and Stante, A. (2015). Pattern-based approach for designing fail-operational safety-critical embedded systems. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 52–59.

Saridakis, T. (2004). Towards the integration of fault, resource, and power management. In *23rd International Conference on Computer Safety, Reliability and Security*, pages 72–86, Potsdam, Germany.

Saridakis, T. (2005). Surviving errors in component-based software. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 114–123.

Saridakis, T. (2009). *Design Patterns for Graceful Degradation*, pages 67–93. Springer Berlin Heidelberg, Berlin, Heidelberg.

Schirmeier, H., Neuhalfen, J., Korb, I., Spinczyk, O., and Engel, M. (2011). RAMpage: graceful degradation management for memory errors in commodity linux servers. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 89–98.

Shelton, C. P. and Koopman, P. (2004). Improving system dependability with functional alternatives. In *International Conference on Dependable Systems and Networks*, pages 295–304.

Trindade, R., Bulwahn, L., and Ainhauser, C. (2014). Automatically generated safety mechanisms from semi-formal software safety requirements. In Bondavalli, A. and Di Giandomenico, F., editors, *Computer Safety, Reliability, and Security*, pages 278–293, Cham. Springer International Publishing.