Model-Driven Engineering of Dependable Systems

Vidar Slåtten
Department of Telematics, NTNU, Trondheim, Norway
vidarsl@item.ntnu.no

***Abstract*—Improving the dependability of a computer system increases the acquisition cost so much that many systems are built without a cost-effective level of dependability. This motivates our decision to work on *reducing the development effort and competence required to create dependable, distributed, reactive systems*. The scope is narrowed to extending the SPACE method with software-implemented fault-tolerance mechanisms and providing tool-supported fault removal in the form of model checking. The results so far mainly cover fault removal, but we also have some early results on providing fault-tolerance mechanisms at the application layer. We discuss future work as well.**

I. Introduction

As many of us already depend on computer systems to lead our lives to a standard we find acceptable, we find it natural to put a correspondingly strong emphasis on ensuring that these systems are indeed dependable. In [1], one of the definitions of the dependability of a system is "the ability to avoid service failures that are more frequent and more severe than is acceptable.", a service failure being a deviation in the visible system behaviour from that of its specification.

Depending on the likelihood of faults and the consequences of service failures, some share of the development resources should go towards ensuring the dependability of any system being developed. However, increasing the dependability of a system, always comes at a cost, in terms of the effort and competence required of the developers.

Helvik [2] argues that "the additional investment in extra hardware and in development of fault handling, maintenance support etc., may be larger than the cost of the development of a system with "ordinary" dependability", but that for a balanced application of the means for dependability (see next section) this acquisition cost may be more than offset by lower operation costs over the system life cycle.

One example of how to increase the dependability is adding fault-tolerance mechanisms, in order to avoid service failures even in the presence of active faults. Knowledge in both the functional and the dependability domain is required of the developers to correctly implement this. And in so doing, the system is likely to become more complex, increasing the chance of software faults being introduced during development or later maintenance [3].

All in all, we find that it is so hard to create dependable systems, that developers tend to create systems that have lower acquisition costs, but unnecessarily high life cycle costs. The question at hand is then: How can we enable developers to create dependable systems without paying too high a price in terms of the effort and competence required?

For the remainder of this paper, to denote a low cost in terms of both effort and competence, we say that something can be done *easily*.

In this work, we concern ourselves with distributed reactive systems. These are systems that are (intended to be) always running, maintaining an ongoing interaction with their environment. Such a system is usually not easily reset to some global initial state, and if so, important state information regarding its current sessions with its environment may be lost. Hence, "restarting" is usually not a good option; we need to avoid failures instead.

To sum up, we would say that the *goal* of this work is *to reduce the development effort and competence required to create dependable, distributed, reactive systems*.

A. Scope and Terminology

Avizienis et al. provide a reference work for the dependability field [1]. We will adhere to the definitions therein.

The authors of [1] define the threats to dependability to be faults, errors and failures. Hereby, a failure is when the service of a system deviates from the correct service. A service is defined to be the behaviour of the system that is visible to its users. Further, an error is the part of the total state of a system that may lead to a failure, i.e. a deviation in the state of a system from the correct state. Finally, a fault is the (hypothesized) cause of an error, e.g. a software bug.

The concept of dependability is further defined to encompass the following attributes: availability, reliability, safety, integrity and maintainability. While all of these attributes are worthy of attention when making a distributed reactive system, we will focus primarily on reliability and availability while also giving some thought to maintainability. The attribute of safety could prove an interesting extension, but this is not in focus at the moment.

We include maintainability as a secondary focus because we think that creating highly reliable and available systems also implies creating easily maintainable systems. I.e. if a system is built using a method that ensures easily understandable specifications, we are less likely to introduce faults both during initial development and later maintenance.

In addition to defining dependability, its attributes and its threats, the authors of [1] lay out the means for achieving

dependability:

- **Fault prevention** means to prevent the occurrence or introduction of faults.
- **Fault tolerance** means to avoid service failures in the presence of faults.
- **Fault removal** means to reduce the number and severity of faults.
- **Fault forecasting** means to estimate the present number, the future incidence, and the likely consequences of faults. [1]

Fault prevention is closely related to the maintainability attribute, as the methods for fault prevention result in easily maintainable specifications, and easily maintainable specifications decrease the chance of introducing new faults.

Fault removal can be further divided into verification, diagnosis and correction [1]. To make fault removal easy for the developers, these processes should be as automated as possible. This work focuses on fault removal during development time only, not during system use.

We will focus on tolerating both hardware and software faults, but we narrow it down to fault-tolerance mechanisms that can be implemented in software, using only off-the-shelf hardware. In [5], the authors define the term *software fault tolerance* to mean just this. However, that term is also often used for mechanisms that aim only at tolerating software faults. We therefore stress our intended meaning by using the term *software-implemented fault tolerance* to denote this part of our scope.

To efficiently achieve our goal, we choose to extend an existing tool-supported method already under development at the Department of Telematics: The SPACE method [4], accompanied by the Arctis tool suite, provides a novel way of developing reactive systems. Here, we use UML collaborations between two or more roles as the basic building blocks for composing systems. These collaborations are further refined by UML activities to describe their behaviour. Each collaboration is encapsulated in a building block with an external state machine that describes its interface behaviour. Hence, each block can can be reused without looking inside at its detailed behaviour. For deployment, the collaborative specifications are transformed into component state machines before a code generator generates the (Java) code.

These models have a formal semantics defined in temporal logic, making model checking possible. Model checking can be highly automated and is hence a suitable choice for our verification method. Many fault-tolerance mechanisms can also be thought of as collaborations between two or more roles, possibly physically distributed. We therefore see enhancing Arctis with a *library of reusable fault-tolerance mechanisms* as a natural step towards achieving our goal.

We argue that the SPACE method already scores high in fault prevention through its graphical, layered UML models. We therefore do not emphasize improving on fault prevention, or maintainability, in this work. However, it will be a constraint not to worsen maintainability when attempting to improve on the other targeted dependability attributes.

Faults can be classified by eight viewpoints, one of them being the objective, malicious or non-malicious [1]. Removing or tolerating malicious faults (human attacker) is outside our scope.

B. Related Work

In [5], the authors introduce what is later coined the SwiFT library of fault-tolerance mechanisms and argue that, as supported by the end-to-end argument, these fault-tolerance mechanisms are at least as well included at the application layer, as in underlying system layers. While they work at the level of programming languages, we attempt to bring this idea into the domain of model-driven engineering.

A survey of application-layer fault-tolerance mechanisms can be found in [3], where the authors argue for tolerating software faults, not just hardware faults. They also state that "A number of important choices pertaining to the adopted fault-tolerance provisions, such as the parameters of a temporal redundancy strategy, are a consequence of an analysis of the environment in which the application is to be deployed and run." This implies that our approach should include a description of the deployment environment of the system.

Bucchiarone et al. [6] present plans for an approach similar to our own, in that they consider both fault tolerance and fault removal important means to increase the dependability of systems. They specify functional and fault-tolerance requirements by UML use cases. Then, a system architecture is created in the form of UML component diagrams for structure and state machines for their abstract behaviour. They plan to use model checking to verify that the system architecture adheres to the fault-tolerance requirements. Once this is verified, they intend to use the system architecture model to generate test cases for a manually implemented executable system. They already have a tool-supported method, CHARMY, that does this, but it does not take fault tolerance into account yet. While combining fault-tolerance mechanisms with model checking to uncover design faults is similar to what we are planning, our approach enables us to generate code directly, instead of manually implementing the system and then testing it.

Domokos and Majzik [7] look at how to incorporate dependability via *aspect-oriented* modelling. This aims to separate the concerns of the functional designer and the dependability expert. The dependability expert adds reusable fault-tolerance patterns to the functional design, and a model weaver then generates an integrated model, as well as an analysis model that captures the failure and repair processes of the system components and how errors propagate between them. This approach is very high-level in that it does not deal

with the actual behaviour of the system, just the structure. It is hence not possible to generate an executable system from these models, nor to verify if the system has been composed correctly with regards to behaviour. It does, however, cover fault forecasting and could provide useful inspiration should we go in that direction later.

Guelfi et al. [8] present the DRIP Catalyst approach where coordinated atomic actions (CAAs) are used to specify all system behaviour. A CAA is represented by an activity diagram with each role in its own partition, somewhat similar to the way we use activity diagrams to describe the collaboration of roles. They intend to follow the MDA approach of refining a platform-independent model to a platform-specific model (PSM) and then generating code, but currently they write the PSM directly. Verification of the system behaviour was not implemented at the time of writing, but was planned as future work.

We have not yet come across any approaches that express fault-tolerance mechanisms as reusable collaborations. Nor have we found any that formally verify the system behaviour after introducing fault-tolerance mechanisms, while still modelling the system detailed enough that executable code can be generated automatically.

C. Structure

The rest of this paper is structured as follows: The next section will outline our research design, including how we intend to validate our approach. Sect. III summarizes our results so far, while Sect. IV outlines our future work.

II. Research Design

We have chosen to focus on using model checking and software-implemented fault tolerance to increase the dependability of distributed reactive systems built using the SPACE method. Within this scope, we have come up with the following research questions and corresponding working hypotheses:

RQ1: How can we enable developers to easily remove faults in the functional design of systems?

- H1: The temporal logic semantics of SPACE specifications enable us to automatically create a behaviour model of the system so that model checking can be used to verify a set of functional properties.
- H2: A useful subset of the properties to be verified can be derived automatically from the model.
- H3: In case of property violations, the error trace from the model checker can be expressed in terms of the UML model so that the developer does not need to be competent in the temporal logic domain.
- H4: It is possible to automate the diagnosis of a useful subset of design faults.
- H5: It is possible to provide automatic corrections for a useful subset of diagnosed design faults.

RQ2: How can we enable developers to easily augment their systems with fault-tolerance mechanisms?

- H6: We can and should provide most fault-tolerance mechanisms at the application layer.
- H7: Fault-tolerance mechanisms are suitable to be expressed and encapsulated in reusable collaborative building blocks.

Many fault-tolerance building blocks will be platform specific. We must have a way to easily configure them to the current platform. An example is how the expected round-trip delay of messages should be taken as input when configuring some timers.

- H8: A deployment model is sufficient to automatically configure fault-tolerance mechanisms.

RQ3: How can we answer RQ2 without compromising the maintainability of the system specifications?

- H9: Encapsulating fault-tolerance mechanisms in building blocks is sufficient to separate the concerns of functionality and fault tolerance.
- H9b: If H9 is not true, then separating fault tolerance into its own aspect of the model will achieve sufficient separation of concerns.

RQ4: How can we enable developers to easily remove design faults in a platform-specific, fault-tolerance augmented system?

- H10: Our current work to answer RQ1 can be extended to verify a useful subset of functional properties in the presence of faults, by using a deployment model to automatically generate platform-specific environment behaviour in the temporal logic specification.

To fully analyse e.g. a system using replicated instances to tolerate node crashes, we must model check a model with several active instances of each type. This can potentially cause a state space explosion, as the state space can grow exponentially with every instance. However, the interesting result is to see how few working nodes a system can manage with, not how many.

- H11: We can verify sufficiently large models, to be useful in practise, without the state space exploding.

A. Methodology and Validation of Results

The work done so far has taken the form of extending Arctis with working implementations of our methods for both fault removal and fault tolerance. This enables us to experiment with our methods in practise, and let others (mainly students, so far) use the same tools and give feedback on their suitability. We intend to continue creating working implementations to extend the Arctis tool suite when possible.

Validation is a matter of using our tools and building blocks to create working (small scale) systems. Through the ISIS project [9], industry partners are testing the tool suite for further use. This will hopefully lead to a much

larger base of users to collect feedback from. Publishing peer-reviewed papers in related fields is, of course, also a form of validation; one which we aim to pursue as much as possible, as the thesis is going to be a paper collection.

III. Results (so far)

In [10], we show how we can automatically transform collaborative UML specifications of the systems created in Arctis, to the Temporal Logic of Actions, TLA [11]. We also automatically create theorems for a set of safety properties that we wish to be informed of any violations of. These theorems, along with the corresponding system specifications, are then given as input to a model checker, which gives a textual error trace in terms of the TLA specification if any theorem violations are found.

In [4], we outline the whole SPACE approach. Relevant to this work, is how we have integrated the model checking into Arctis, so that any error traces are given graphically, in terms of the original specifications. We further show how a theorem violation is taken as a symptom, to be used as input when attempting an automatic diagnosis. In some cases, we also provide fixes that can be automatically applied to the system specification.

The work so far, has focused on RQ1. We consider H1-H3 confirmed, while H4 and H5 can be considered plausible; some examples have been demonstrated, but the set of faults to be automatically diagnosed and fixed is not yet very large.

In [12], we introduce symmetric building blocks as a way to model multiple instances of the same type collaborating with each other. We further introduce a reusable building block for a leader election protocol, which itself contains a reusable failure detector to detect node crashes. This paper gives an early indication to support hypotheses H6, H7 and H9.

IV. Future Work

RQ2 calls for a library of both platform-independent and platform-specific building blocks that a developer can use to compose a suitable set of fault tolerance provisions for the fault model the system is being developed for. We envision a library that contains building blocks for dealing with the most typical (failure) semantics of the system's underlying platform, as well as software faults.

Some fault-tolerance mechanisms can, however, be more suitable to place in the platform or the code generator. E.g., we may create a code generator for a group communication system (GCS), like the ones in [13], to take advantage of the atomic broadcast primitive, which ensures that all group members receive the same messages in the same order. This will ease the application development, at the expense of all nodes having to run a middleware that may be doing more than the application requires.

To separate the concerns of functionality from fault tolerance (see RQ3), we will experiment to see if the current encapsulation in building blocks will be enough. If not, we have to add new decomposition techniques so as to avoid cluttering the functional view of the specifications. Aspect orientation promises a solution to this, but it remains to be seen if it can be adopted in this case.

RQ4 leads us to another main focus, which is on further developing our verification by model checking, introduced in [4], [10], to also verify the functional properties in the face of channel and node failures. This can be done by altering the temporal logic specification to take into account that e.g. channels may drop messages and nodes may crash. We then add fault-tolerance mechanisms and run the verification to track down any new design faults introduced in this process.

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. on Dep. and Sec. Comp.*, vol. 1, no. 1, pp. 11–33, 2004.

[2] B. E. Helvik, *Dependable Computing Systems and Communication Networks; Design and Evaluation*. Trondheim, Norway: Tapir Academic Press, January 2009.

[3] V. D. Florio and C. Blondia, "A survey of linguistic structures for application-level fault tolerance," *ACM Comput. Surv.*, vol. 40, no. 2, pp. 1–37, 2008.

[4] F. A. Kraemer, V. Slåtten, and P. Herrmann, "Tool support for the rapid composition, analysis and implementation of reactive services," *Journal of Syst. and Soft.*, 2009, in press.

[5] Y. Huang and C. M. R. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," in *Proc. of 23rd International Symp. on FT Comp.*, June 1993, pp. 2–9.

[6] A. Bucchiarone, H. Muccini, and P. Pelliccione, "Architecting Fault-Tolerant Component-Based Systems: From Requirements to Testing," *Electron. Notes Theor. Comput. Sci.*, vol. 168, pp. 77–90, 2007.

[7] P. Domokos and I. Majzik, "Design and analysis of fault tolerant architectures by model weaving," in *Proc. of the Ninth IEEE Int. Symp. on High-Assurance Syst. Eng.*, 2005.

[8] N. Guelfi, R. Razavi, A. Romanovsky, and S. Vandenbergh, "DRIP Catalyst: an MDE/MDA Method for Fault-tolerant Distributed Software Families Development," in *OOPSLA & GPCE workshop on best practices for Model Driven Development*, Vancouver, Canada, 2004.

[9] "ISIS Project," http://www.isisproject.org/, Oct. 2009.

[10] F. A. Kraemer, V. Slåtten, and P. Herrmann, "Engineering Support for UML Activities by Automated Model-Checking — An Example," in *Proc. of the 4th Int. Works. on Rapid Integration of Software Engineering Tech.*, November 2007.

[11] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[12] F. A. Kraemer, V. Slåtten, and P. Herrmann, "Model-Driven Construction of Embedded Applications based on Reusable Building Blocks - An Example," in *Proc. of the 14th Int. SDL Forum*, September 2009, pp. 1–18.

[13] N. Carvalho, J. Pereira, and L. Rodrigues, "Towards a generic group communication service," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 2006, pp. 1485–1502.