Towards a Model-Driven Method for Reliable Applications: From Ideal To Realistic Transmission Semantics

Vidar Slåtten, Frank Alexander Kraemer and Peter Herrmann
Department of Telematics
Norwegian University of Science and Technology (NTNU), N-7491 Trondheim, Norway
{vidarsl, kraemer, herrmann}@item.ntnu.no

ABSTRACT

We present a model-driven method to incrementally introduce fault-tolerance mechanisms into application models that are initially developed with assumptions of ideal transmission semantics. As main structuring units, our models use collaborative building blocks in UML that can encapsulate the behaviour of several participants in order to perform a certain task. Since these building blocks can be designed and analysed separately, fault-tolerance mechanisms can be introduced block by block, which reduces the size and complexity of specifications that have to be understood at a time. Applying fault tolerance at the application layer also brings the benefits of easily porting applications to other platforms and applying model-level analysis tools to the fault-tolerance mechanisms themselves. We illustrate our method through the development of an access control system.

1. INTRODUCTION

A major challenge when developing distributed, reactive applications is that each component an application consists of typically has to maintain interactions with several other components. This implies a considerable amount of coordinating logic. Such logic gets even more complex once applications should handle situations in which communication is disturbed by flaws like message loss. Luckily, in many cases, already simple fault-tolerance mechanisms can lead to more reliable applications. For example, timers started when waiting for response signals can protect a component from waiting forever for an answer that may have been lost in a channel.

Though the introduction of even simple fault-tolerant behaviour itself is a task that increases development time, it has an additional effect that makes it problematic during the development: Fault-tolerance mechanisms obscure the essence of the actual application logic, making it harder to understand in the beginning and maintain in the long run. For instance, in a finished, fault-tolerant specification it may not be immediately clear if a certain timer is an important application feature or only has the task to monitor a communication channel, since fault-tolerance mechanisms and application logic are mixed together.

Another reason demanding a clear strategy for the introduction of fault tolerance is rather practical. The expertise needed for securing an application with respect to fault tolerance is different from the expertise needed for application development in specific domains. Only few engineers cover both. Our method therefore explicitly addresses two separate groups of experts; one for the specific application domain in which the system should be applied and one for fault tolerance in general. This is analogous to our method presented in [14] to handle security aspects by a separate team of experts.

For these reasons, we propose a two-phase method, depicted in Fig. 1, based on our engineering method for reactive system, SPACE [22]. In a first phase, a system specification is developed by experts for the specific application domain in step **D1**. As major specification units, we use special self-contained UML building blocks addressing a single task. Several case studies have shown reuse proportions of 71 % on average, see [20]. Therefore, domain-specific libraries can offer existing solutions that can be composed to more comprehensive units, until a complete system is obtained. We will detail this development phase in Sect. 2. Once this specification passes the analysis based on model checking **A1**, the majority of domain-specific design is complete and fault-tolerance experts become involved. We refer to a specification at this stage as *idealized*, meaning that it is fault-intolerant, but free from the design faults checked for by the analysis step.

Within the second phase, the initially idealized specification is incrementally improved to match more realistic transmission semantics in which messages in transit can be lost (step **D2**). This task is performed by an expert on fault tolerance, assisted by an expert on the application domain when necessary. Similar to the initial development in **D1**, this step is supported by a library of fault-tolerance mechanisms that store solutions to reoccurring problems. This development phase is the main contribution of this paper and is further detailed in Sect. 4.

Once the system passes analysis **A2** it is considered reliable (fault-tolerant and free from design faults) and can be implemented by our automated process consisting of a model transformation [19] with a subsequent code generation step. This step supports different execution platforms such as standard Java, embedded Java for Sun SPOTs [21],

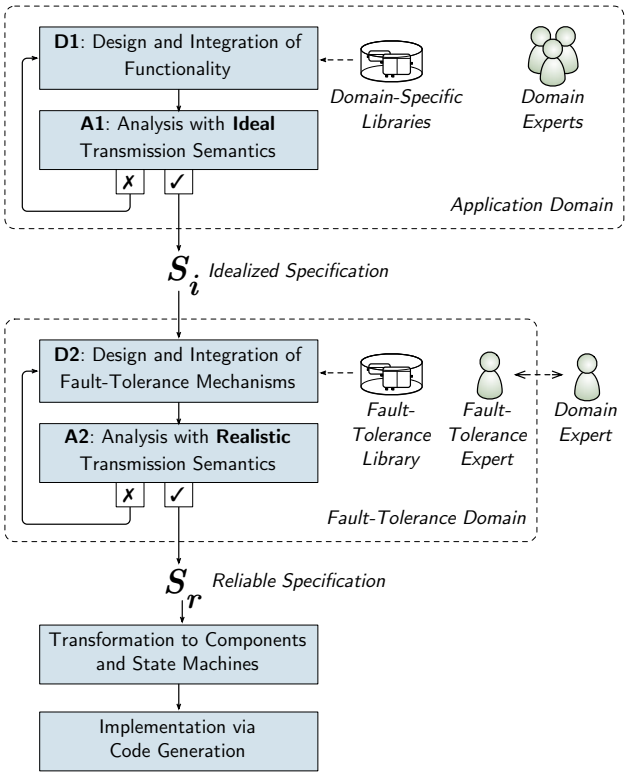

Figure 1: Development Method

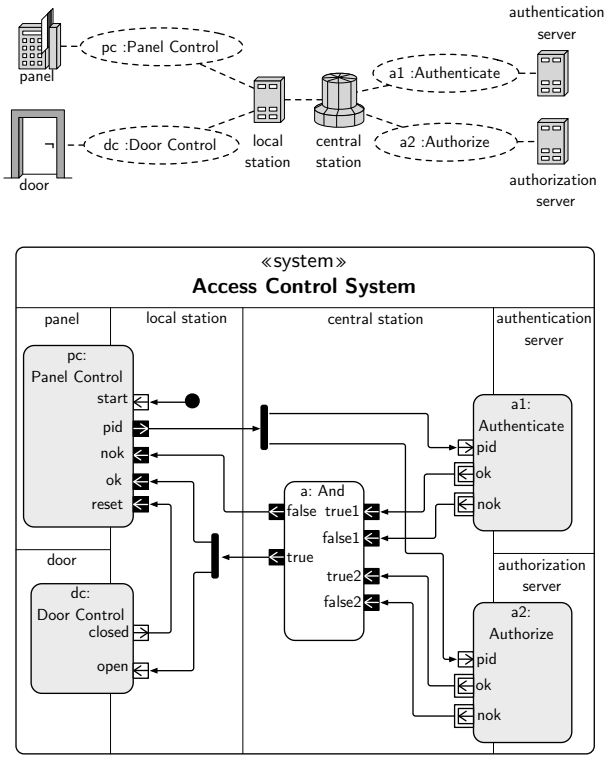

Figure 2: Access Control System

Android [15], as well as Telenor's Connected Objects Operating System (*COOS*, [16]).

In the following, we will detail the development steps of phase 1 for the application domain in Sect. 2 by the example of an access control system. The models in Sect. 2 assume ideal transmission semantics, i.e. perfect channels. In Sect. 3, we introduce realistic transmission semantics. This is the starting point for development phase 2, discussed in Sect. 4, that incrementally evolves the idealized system to a reliable one by introducing fault-tolerance mechanisms. We present related work in Sect. 5 and discuss our method in Sect. 6. We end with some concluding remarks in Sect. 7.

While our tools also handle activity diagrams with object nodes, operations and flows, we disregard data within this paper for clarity. For system specifications that include data, see [21, 22].

2. IDEALIZED APPLICATION MODELS

To illustrate our approach, we consider an access control system. At the top of Fig. 2 the system's structure is specified by a UML collaboration, with the icons representing its participants (UML collaboration *roles*). They show that the system consists of the actual door to control, an input panel, a local station located in the vicinity of the door and the panel, a central station and two servers for authentication and authorization. The ellipses (UML collaboration *uses*) in between refer to collaborations that describe functions between their participants. The collaboration use *pc: Panel Control*, for instance, covers the behaviour between the local station and the panel, in which users provide their access code.

Since UML collaborations are a structural description without any behaviour, we use UML activities to complement the specification, as shown in the lower part of Fig. 2. Here we see that each collaboration role is represented by its own activity partitions for the panel, the door, the stations and the servers. The collaboration uses are represented by call behaviour actions that in turn refer to activities. At their frames, they have pins attached which are used to control their behaviour.

The system is started via the initial node within the local station, which starts the panel. Once users enter their personal identification (pid), the corresponding pin from block *pc* emits a token containing it onto the flow. The token is forwarded to the central station, where it is forwarded, after the fork node, to both the authenticate and authorize collaborations, more generally referred to as *building blocks*.

The internal behaviour of the authenticate block is shown by the UML activity diagram to the left in Fig. 3. It is a simple inquiry pattern, where a server validates the pid and the client interprets its answer as either ok or not. The authorization works in a similar way.

To utilize these blocks within the access control system, however, the internal details are not important. It is sufficient to look at the external behaviour of these blocks, expressed by the so-called external state machine (ESM) to the right in Fig. 3. It shows that after a token is provided via input node *pid*, the authentication terminates either via *ok* or *nok*.

The central station of Fig. 2 collects the answers of both the authenticate and the authorize collaborations. Note that they were started simultaneously, but the servers may respond in any order. The results are fed into the block

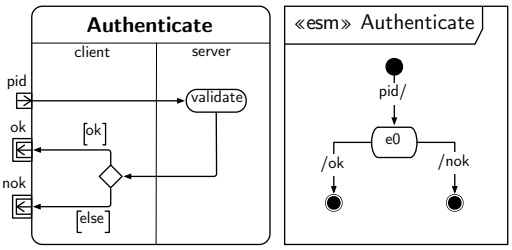

Figure 3: Behaviour of the Authenticate building block

a:And, which realizes a logical function that corresponds to a boolean and gate: Only when both the authentication and the authorization are ok, *true* is sent back to the local station upon which the door is opened. In all other cases, the door remains closed. The ESMs of *Panel Control* and *Door Control* are depicted in Fig. 4.

Since all building blocks are encapsulated by ESMs, the access control system of Fig. 2 can be simulated and analysed even if the panel or door control blocks are not yet specified internally. Once all building blocks are complete, the system (although assuming ideal channels) can also be implemented and executed as an early prototype. This can be used to uncover situations not yet considered and to get early feedback from the customers.

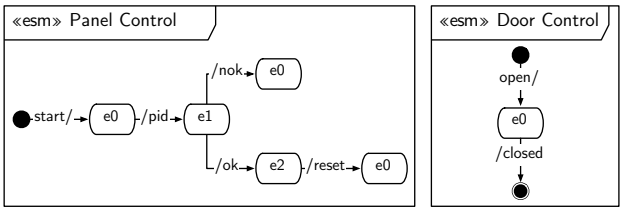

Figure 4: External descriptions of the Panel Control and Door Control building blocks

The tool suite that accompanies our method has analytic capabilities to aid the developer in creating a well-formed specification. Syntactic inspectors check that the model is syntactically correct, for example that all output pins have a connected flow. Semantic analysis is achieved through the use of a model checker [30]. The semantics of our models are precisely defined so that an automated transformation to a model checking language can be done [22]. Properties like freedom from deadlocks and that the composition respects all ESMs are then automatically formulated and verified.

The access control system is analysed and does not violate any properties as it is now.

3. TRANSMISSION SEMANTICS

The activities in Fig. 2 and 3 use control flows that cross partition borders. Since partitions denote physically distributed components, this implies some form of communication, in which data is transmitted from a sender to a receiver. For all platforms we generate code for, this communication is provided by an asynchronous message bus, in which the sending operation is decoupled from the receiving of a message, so that a sender does not get blocked. This also means that there is no upper bound on the time it may take a transmitted message to reach its receiver. To mirror this in the execution semantics for activities based on token flows, we therefore assume that tokens are buffered between partitions in an implicit waiting place, as illustrated in Fig. 5. We assume in the following that there is at most one message corresponding to a certain activity flow under transmission, that means that the place accepts at most one token.

Figure 5: Implicit waiting place for transmissions

Ideal Transmission Semantics. The models in Fig. 2 and 3 assume an ideal form of communication, in which messages are never lost. This means that the transmission between partitions has the semantics described by the building block in Fig. 6, with a send and receive operation. Every token sent will eventually be received, as expressed by the ESM to its right.

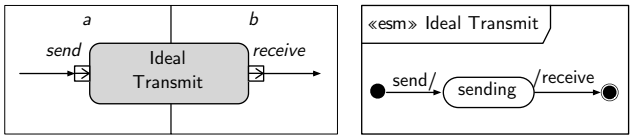

Figure 6: Ideal transmission semantics

Realistic Transmission Semantics. To represent message loss, we define realistic transmission semantics by the building block in Fig. 7. It has the same send and receive operations as the idealized transmit in Fig. 6, but also models, by pin and ESM transition *lost*, that tokens can be lost and hence never be received. (Since *lost* and *receive* are mutually exclusive, they technically belong to different UML parameter sets denoted by the additional box around them.)

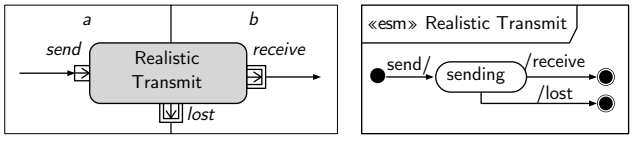

Figure 7: Realistic transmission semantics

4. RELIABLE APPLICATION MODELS

If we assume realistic transmission semantics, i.e. that channels may lose messages, a single lost message anywhere in the access control system will leave it deadlocked. The analysis step A2 therefore reveals the following error scenarios:

- The initial token from the local station to the central station containing the *pid* is lost, so the local station waits forever for a reply.
- Any one of the responses from the central station to the local station are lost, also leaving the local station waiting forever.

- Any token lost between the central station and one of the servers causes both the local station and the central station to wait forever.

In the event of a message being lost by a channel, the application should be notified so that it can handle the event in the best way possible for its specific domain. An application could, for example, attempt to compensate for a broken channel by using a redundant information source. Otherwise, it could degrade its service, simply informing its users that requests currently cannot be handled. To achieve that the application is notified of message loss, it can either register for such events through a platform API, or use a timeout mechanisms explicitly represented within the model. The rest of this section presents such a mechanism, how it can be integrated with the access control system and how it can be encapsulated for reuse.

4.1 Reliable Notify

Unreliable channels do not explicitly notify any party in the event that a message is lost. Hence, detection must be done at the endpoints. The sender of a message can start a timer when sending a message onto a channel, and ask for an explicit acknowledgement of receipt from the receiver. If the acknowledgement message is received on the sender side, the sender knows for sure that the message it sent reached its destination. However, the lack of the acknowledgement message when the timer expires can be interpreted as any of the following situations:

- The sender's message was lost before reaching the receiver.
- The acknowledgement message has been lost on its way back do the sender.
- The sender's message, or its acknowledgement message, is still in the channel.

Hence, there is no way of being sure that the sender's message has been lost; all we can do is set a time after which it is unlikely that the acknowledgement message will arrive and consider the sender's message as lost after that time. To prevent a message or acknowledgement mistakenly assumed to be lost from interfering with later interactions, we assume a numbering mechanisms for filtering them out.

We introduce a solution for a reliable notification service in the form of a building block, *Notify R*, shown in Fig. 8. (We use the naming convention that *R* is added as suffix to the name to mark that a building block is reliable, i.e. that it has passed analysis A2.) This building block encapsulates the behaviour just described to detect message loss: When a token arrives through pin *in*, the sender side puts it onto the channel while at the same time starting a timer. Upon receiving the token from the channel, the receiver side delivers it to the enclosing building block via pin *out*. The receiver also sends an acknowledgement back to the sender side. When the sender receives the acknowledgement, the building block terminates via the *ack* pin. Terminating the block also removes any token in the timer. Should the timer expire before the acknowledgement arrives from the receiver side, the building block will terminate via the *noAck* pin, indicating to the enclosing building block that the token may not have reached the receiver (yet).

The behaviour of *Notify R* is abstracted by its ESM, as shown in Fig. 9. An important property is that the block always emits a token through *ack* or *noAck* before it terminates, even if tokens are lost between partitions.

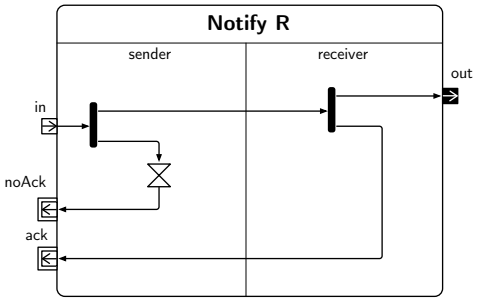

Figure 8: Behaviour of the Notify R building block

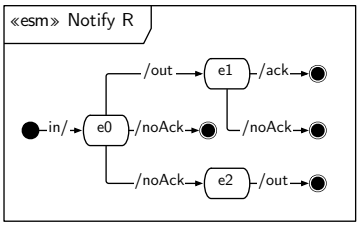

Figure 9: ESM of the Notify R building block

4.2 Reliable Authenticate

Following our method from Fig. 1, we start our work on the access control system by looking at the collaborations it consists of, and begin with the one for authentication, as introduced in Fig. 3.

We run analysis A2 on it, with realistic transmission semantics. It finds that a deadlock occurs when the token from *pid* is lost in the channel between client and server, represented by the point where the topmost flow crosses the partition border. As a result, we invoke a syntactic substitution action on this flow that inserts an instance of *Notify R*, changes the target of the old flow to its *in* pin and adds a new flow going from pin *out* to the original target of the first flow (the *validate* operation). The substitution algorithm also connects *ack* to a flow final node and *noAck* to a newly inserted pin in *Authenticate* named *failed*. The result can be seen in Fig. 10.

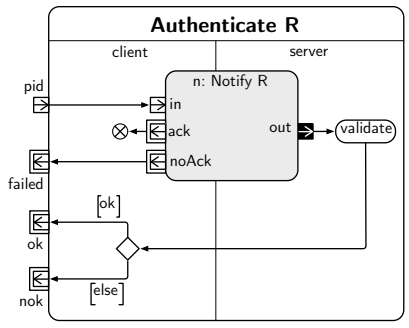

Figure 10: The Authenticate building block after syntactic substitution of topmost flow

The analysis tool will detect property violations introduced by the changes. For Fig. 10, the analysis finds a deadlock to happen in the event that the token returning from the server to the client is lost. In other words, the client is waiting for a response from the server before terminating, and we realize that the block must somehow detect if this could also be lost. Substituting the flow carrying the response for a new *Notify R* would only help in notifying the server side of the possibly lost token, which is not helpful as the server has no interest in what happens after it has sent its response. Instead, we connect the flow from the *ack* pin to a timer, which we merge with the flow going to the *failed* pin, as shown in Fig. 11. This way, *Authenticate R* always terminates, something which the analysis in step A2 confirms.

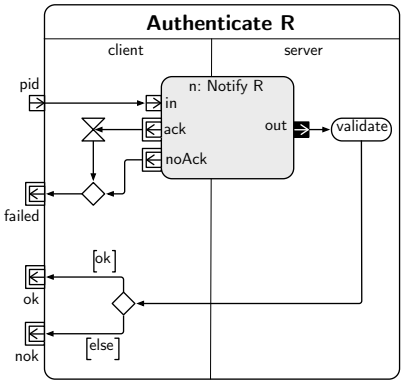

Figure 11: The reliable Authenticate building block completed

The authorize building block is modified in exactly the same way as the authenticate block, and the description is hence omitted. The blocks for door control and panel control are not further specified in this example, and are assumed to be infallible.

4.3 Reliable Inquiry

We notice a pattern in the final version of *Authenticate R*. The parts that we just added are useful for all instances of this request–response pattern. Hence, we create a new building block, *Inquiry R*, as shown in Fig. 12 to encapsulate this communication pattern for future reuse. The ESM of the block is depicted in Fig. 13.

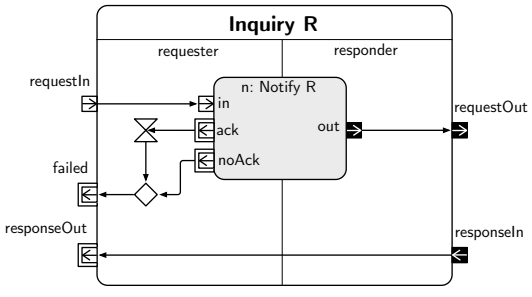

Figure 12: Behaviour of the Inquiry R building block

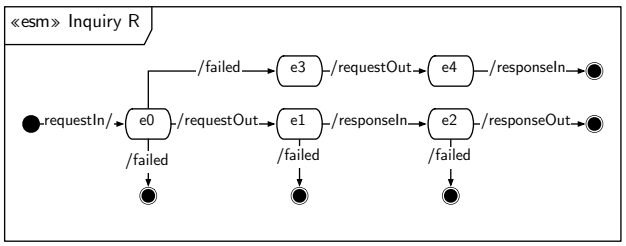

Figure 13: ESM of the Inquiry R building block

This building block now uses two timers in total, one within *n: Notify R* waiting for the acknowledgement of the request, and one that waits for the response. One could say that the latter makes the first redundant since sending the response is of course an indirect acknowledgement for the initial request. However, when the computation time on the server for producing the response is considerably longer than the transmission time for a signal (since it for example could include further communication with other components), the extra acknowledgement would allow the client to detect communication problems earlier. Anyway, if computation for the response would be very short, one could provide an extra version for Inquiry that uses the response as acknowledgement for the request.

4.4 Reliable Access Control System

Having created reliable versions of the authenticate and authorize blocks that reflect that messages may be lost, we now make the access control system reliable as well. We see a request–response pattern similar to that of *Inquiry R* between local station and central station. The difference here is that there are two alternative responses, out of which only one should arrive. This is easily added to our existing *Inquiry R* building block to produce the *Inquiry 2 R* block by simply adding a new pair of pins, *responseIn2* and *responseOut2*, and a flow between them.

Using this new block, we obtain the version of the access control system shown in Fig. 14, which satisfies analysis A2. In particular, it is free from deadlocks. As the application already has logic to handle a negative authentication or authorization result (from the *nok* pins), we simply merge the outputs of the *failed* pins with these. The same is the case when a token is lost between the local station and the central station; the flow from *failed* is merged with the negative response. Note that this is a design decision that has to involve an expert on the application domain. The application specified under the ideal transmission semantics assumption does not, in the general case, contain the necessary information to algorithmically transform it into a reliable version. Specifying the best action to take upon detecting message loss hence requires manual intervention, just as the specification of the functionality itself.

5. RELATED WORK

There are several other approaches that combine model-driven development with fault tolerance and techniques for fault removal, like model checking or testing.

Bucchiarone et al. [8] present plans for an approach that utilizes techniques for both fault tolerance and fault removal to increase the dependability of systems. They specify func-

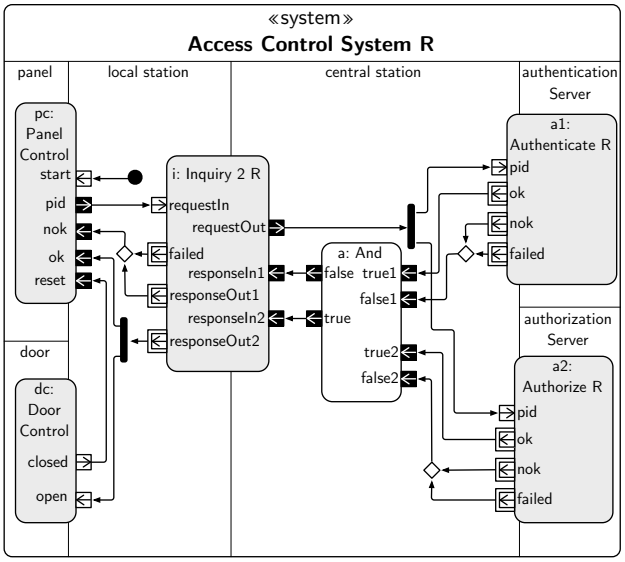

Figure 14: Reliable Access Control System

tional and fault-tolerance requirements by UML use cases and validation scenarios by sequence diagrams. A system architecture is created in the form of UML component diagrams for structure and state machines for behaviour. Each component has one state machine describing normal behaviour and another one describing exceptional behaviour. They already have a tool-supported method, CHARMY capable of model checking and test case generation that they expect to be able to extend for fault-tolerant system architectures without major changes. While combining fault-tolerance mechanisms with model checking to uncover design faults is similar to what we are planning, our approach differs in that we generate the code directly from our specifications, instead of manually implementing the system and then testing it.

In [11], Ermagan et al. specify services using interaction models (sequence diagrams) that are assumed to be complete. The authors propose to add detectors and mitigators to manage these services without altering the services themselves at all. Detectors observe the communication of the service and attempt to detect cases of unexpected behaviour (a message was sent when it should not have been) or of non occurrence behaviour (a message that should have been sent is not). The idea of keeping the functional service specification separated from the fault-tolerance mechanisms that detect and handle errors seems very elegant, but we suspect it is somewhat restrictive in terms of what solutions for error handling can be employed. This will typically also require a complex platform that can, for example, completely transparently re-route messages between service roles, if mitigators are activated to help the system recover from a process crash.

Guelfi et al. present the DRIP Catalyst method in [13]. Here, coordinated atomic actions (CAAs, [29]) are used to specify all system behaviour. A CAA is represented by an activity diagram with each role in its own activity partition, similarly to the way we use activity diagrams to describe the collaboration of roles. The authors intend to follow the MDA approach [24] of refining a platform-independent model to a platform-specific model (PSM) and then generating code, but at the time of writing they create the PSM directly. Verification of the system behaviour is planned as future work. The main difference from our work is that this approach is built around the concept of CAAs, and the DRIP framework for expressing them in Java, so that all behaviour is specified as CAAs from the start. Hence both normal behaviour and fault-tolerance mechanisms are specified at once. We, instead, allow for a naive initial specification and then utilize tool-assisted analysis to help developers introduce fault-tolerance mechanisms in a following step.

Our approach of adding fault tolerance to a functional model bears similarities to aspect-oriented modelling where such cross-cutting concerns are also specified in separation as *aspects* and then weaved into the model. An aspect consists of a *pointcut* model that specifies a matching place to insert the *advice* model, which is the additional logic for handling the aspect.

Domokos and Majzik [10] look at how to incorporate dependability via aspect-oriented modelling. They operate at a purely architectural level, so that the behaviour of the system is not included. The method does, however, output an analysis model in the form of stochastic Petri nets, which can be used to determine the dependability properties of the system, i.e. the failure and repair processes of the system components and how errors propagate between them.

Both Fuentes and Sanchez [12] and Cui et al. [9] use activity diagrams to model system behaviour and aspect advice. The former operates on executable models, like our approach, so that design faults can be found before implementation. However, neither of them apply aspect orientation for fault tolerance, rather persistence, authorization and other aspects whose addition has little consequence for the application.

Kienzle and Guerraoui [17] argue that mechanisms for fault-tolerance (they use transactions) may not be suitable to be separated into an aspect of their own. As a reason they identify that the addition of fault-tolerance mechanisms, at least in their case, requires big changes to the application logic that cannot be made automatically. The experiment described is, however, conducted in the context of aspect-oriented *programming*, not at a modelling level. The applicability to the works in the previous paragraph could hence be questioned.

There are also approaches that incorporate fault tolerance already at the requirements stage of the development process.

Berlizev and Guelfi [6] use UML activity diagrams to detail the behaviour of UML use cases that are again used to express system requirements. In addition to the system functionality, they also incorporate fault tolerance into the requirements, specifying both deviations from normal behaviour and recovery strategies.

Mustafiz et al. [25] present a way to express degraded service outcomes, as well as exceptional modes of operation during the requirements engineering phase of development. They elaborate use cases with UML activity diagrams and mark the possible outcomes of an activity by stereotypes ≪success≫, ≪degraded≫ or ≪failure≫ in order to clearly distinguish them. The authors also advocate the explicit specification of exceptional operation modes that may result from degraded service outcomes. This allows for adjusting the expectations and behaviour of the system users to match

the current situation.

Although the syntax of the two papers above is somewhat similar to ours, the semantics are not. For example, Berlizev and Guelfi use activity partitions to separate between normal and abnormal behaviour, not to separate distributed collaboration roles that could be physically distributed, and hence subject to unreliable communication. As these works focus on requirements, they cover an earlier stage of the development process than ours; what must happen, not how.

Both Bucchiarone et al. [8] and Mustafiz et al. [25] structure their specifications similar to *ideal fault-tolerant components* [3], meaning that all behaviour at the interface of a component is specified for both normal and exceptional cases. Our building blocks also completely specify their interface behaviour, but we do not syntactically separate the exceptional from the normal behaviour. This is because there is (currently) no semantic difference between normal and exceptional behaviour; everything is alternative behaviour that is enabled under certain conditions.

There is some work on automating the process of adding fault tolerance to fault-intolerant systems. In [23], for example, Kulkarni and Arora automatically transform a fault-intolerant program into a fault-tolerant one. However, if all variables cannot be read and written in a single atomic step, which is the case for our asynchronously communicating components, their algorithms have exponential complexity, limiting their practicality.

To sum up, our work looks for a middle ground between the approaches that require developers to design for functionality and fault tolerance at once and the approaches that would add the fault-tolerance aspect automatically at the end. In our experience, fault-tolerance mechanisms can be encapsulated and reused in another application, but they still need a human developer to make the decisions on how to integrate them with that particular application. Combining this with tools for analysis that keep developers from introducing design faults in the integration process, as well as tools for code generation, makes for a practical approach that stands apart from the alternatives that we are aware of.

6. DISCUSSION

We have described an incremental, corrective approach that takes a potentially unreliable specification and incrementally introduces fault-tolerance mechanisms. For instance, during the development of the access control system, we developed reliable versions of *Authenticate* and *Authorize*. From their externals, they differ from their idealized counterparts just by the pin *failed*. Of course, once the reliable versions of these blocks exist, domain experts can directly refer to the reliable blocks in the first place. This reduces the effort in the second development phase, since fewer blocks have to be made reliable.

Building blocks can be analysed separately for their suitability in systems with realistic transmission semantics, since they are encapsulated by ESMs. Once fault-tolerance mechanisms are added, the external behaviour has to be extended in some cases, such as the additional pin *failed* for the authenticate block in Fig. 11. This additional pin is necessary since the transmission failure cannot be (guaranteed to be) handled within the authenticate block, but has to be propagated to the surrounding application, which must decide what should be done. This can trigger a kind of domino effect, where in the worst case the entire system specification has to be extended, starting at the innermost building blocks until the highest level of composition is reached. To prevent this effect, one could follow a specification style in which building blocks are equipped with failure pins by default, as a default hook for any exceptions related to communication errors.

The soundness of our approach can be demonstrated by the behaviour of the executable state machines that it produces. They expose the same behaviour as state machines that would have been designed manually. For a comparison, we consider the state machines for the central station's component:

- The state machine for the central station of the unreliable system as specified in Sect. 2, for instance, has 1 initial state, 6 control states, 6 decisions and 14 transitions as shown in Fig. 15.
- The state machine for the central station of the reliable system as specified in Sect. 4, in comparison, consists of 1 initial state, 5 control states, 6 decisions and 40 transitions as can be seen in Fig. 16.

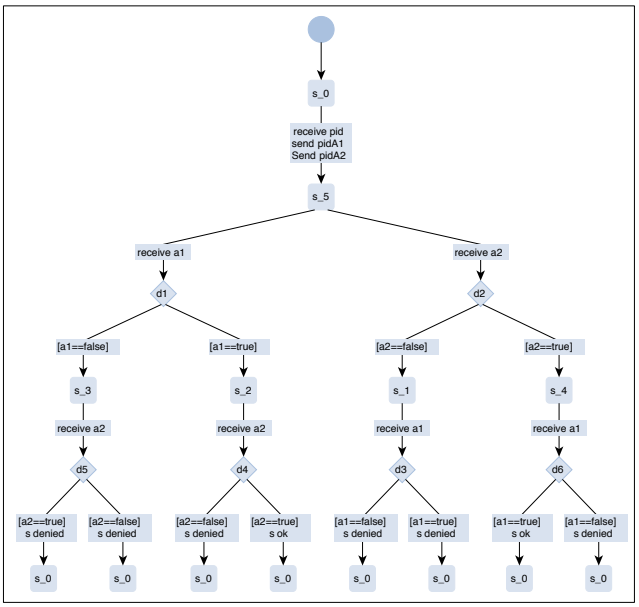

Figure 15: Idealized executable state machine for central station component

These numbers demonstrate how the additional mechanisms for fault tolerance considerably increase the complexity of the resulting logic. We must note that the number of transitions is also due to the automation within our approach. If we would have designed these state machines by hand, some transitions could be saved by using composite states or transitions that are declared for a group of states. However, this does not change the intrinsic complexity of the problem: Due to the fault-tolerance mechanisms, we must also keep track of 5 timers, and the number of signals to receive by the central station has grown from 3 in the idealized central station to 7 for the reliable one.

The comparison of the activity models and the resulting state machines reveals another benefit of our method:

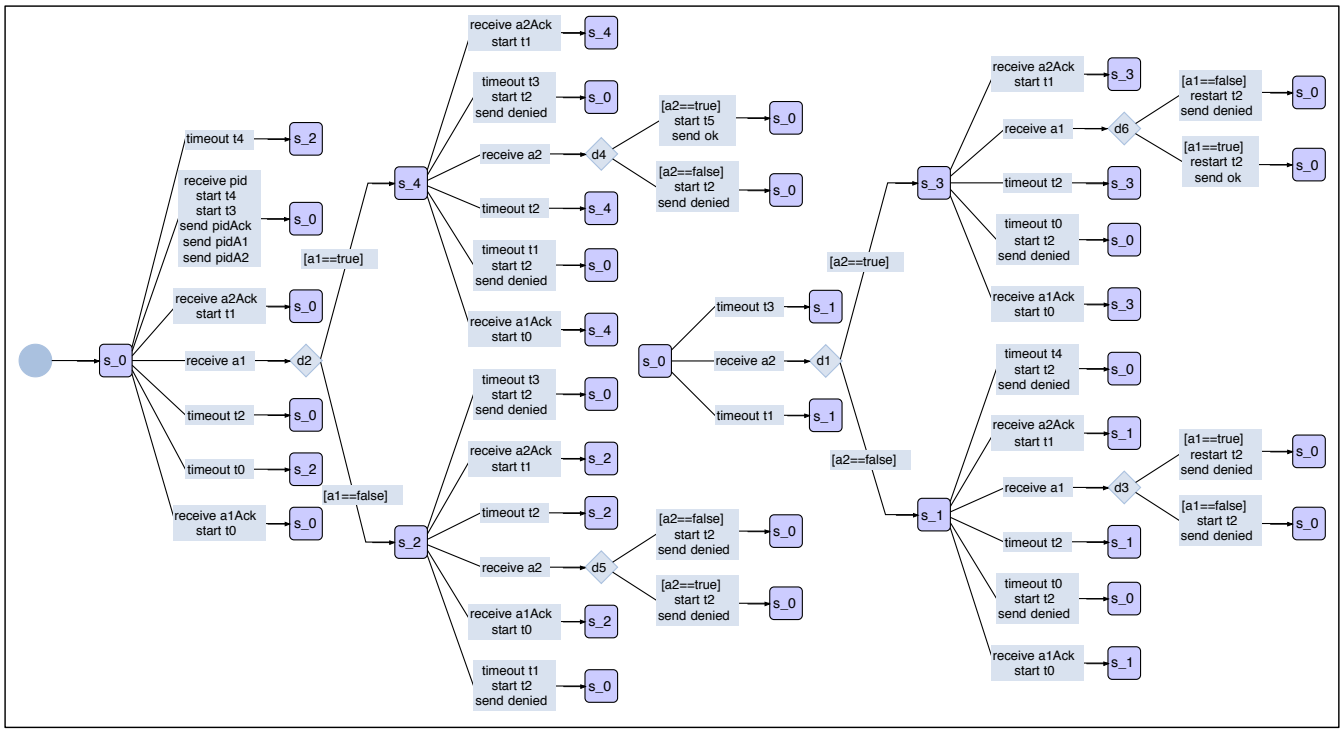

Figure 16: Reliable executable state machine for central station component

On the state machine level, we are not able to reuse fault-tolerance mechanisms in the way we are on the level of activity diagrams. In state machines, elements belonging to one interaction partner are mixed together with elements responsible for other things. Furthermore, the collaborative nature of our building blocks allows developers to study fault-tolerance properties from a holistic viewpoint, where the behaviour of all participants is contained in a single diagram. The activity for *Authenticate*, for instance, specifies both participants in the collaboration. This means that also fault-tolerance mechanisms can be provided as self-contained building blocks.

Another way to separate fault-tolerance mechanisms from application logic is to move the first into a middleware layer providing robust communication primitives to the application layer. However, no matter how sophisticated this middleware layer is, failure-free communication cannot be guaranteed (since somebody can still unplug the ethernet cable), and at some point the application has to be involved, as described by the end-to-end arguments of Saltzer et al. [28]. Moreover, putting advanced mechanisms into the middleware layer makes it a critical asset that demands development and maintenance resources, especially once it should be provided for a wide range of software platforms and devices. We therefore explicitly propose in this paper a method of including these mechanisms into a model that does not require perfect channels from an implementation. This makes an adaptation to different platforms easier, since they only need to provide simple communication primitives. For this reason, we can quickly create code generators for different execution platforms, as the ones named in the introduction. This strong focus on models, however, does not rule out the possibility to have a middleware layer that offers some form of fault tolerance; while a *noAck* within *Notify R* is triggered in our case by a timeout in the model, it could also be triggered by an exception of a middleware layer. The actual integration with the application logic (namely by pin *noAck*) would be equivalent. Figure 17 shows how we would model an *Authenticate R 2* like this. This illustrates the flexibility we have in implementing fault-tolerance mechanisms; we can use mechanisms from both middleware and application-layer libraries as long as the interface behaviour of the mechanisms are described in the model so that the integration with the application logic can be analysed. Middleware will often have a performance advantage, whereas the libraries can provide better portability.

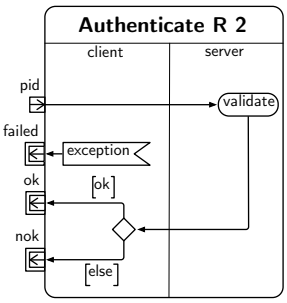

Figure 17: An alternative reliable Authenticate building block using middleware exceptions

To estimate productivity gains from our modelling method based on building blocks, we compared it in [5, 18] to manual programming. The numbers indicate that development time when building blocks are provided is only a fraction of

that needed to manually program a system. These gains are of course only real when building blocks are actually reused among several systems. This, however, is very likely when we consider the high reuse proportions of building blocks observed within our case studies summarized in [20].

7. CONCLUDING REMARKS

For the example, we developed the simple *Notify R* collaboration that uses a timer to protect the access control system against deadlocks. In the absence of a reliable end-to-end transport protocol at lower layers, one may want to handle message resending or reordering within the models as well. This requires a more advanced version of *Notify R*, similar to the Alternating Bit Protocol [4], but leaves the application models utilizing it unchanged.

The error handling presented for the example is rather simple, as a failure is simply reported back as any other negative result. For more complex error handling, one could find that the additional behaviour specified amounts to much more. Hence, we would look into separating the specification of this error handling behaviour from the functional one, so that the functionality of the application (in an ideal world) can still be easily understood. An exception handling mechanisms utilizing UML exceptions [26] may facilitate this. Aspect-oriented modelling is another possibility. We may also modify the syntax of the error handling behaviour to improve the readability of the diagrams.

We currently ensure a finite state space of our specifications by limiting the number of messages in a channel. This way, we can use model checking to verify some properties of our specifications. Abdulla and Jonsson [1] prove that even with unbounded channels, some verification problems are decidable. In [2], they prove that model checking liveness properties [27] is not decidable. More recent work in [7] states that using probabilistic system models, this limitation can be overcome. If modelling unbounded channels should be desirable in the future, we will look into incorporating these techniques into our tools.

In the current approach, the error handling mechanisms are introduced manually, motivated by deadlocks identified in analysis A2. For example, in the *Access Control System R* shown in Fig. 14, we manually insert an *Inquiry 2 R* block to protect the communication between local station and central station from message loss. By utilizing our existing analysis tool, we see the possibility for automatically suggesting inserting these blocks where suitable, based on patterns identified in the state space obtained during model checking.

Further, we want to expand the scope of our method to also deal with unreliable processes that may crash and provide building blocks for detecting this, as well as for maintaining a consistent shared state. Also, in [14] we developed an analogous approach in which unsecured system specifications are extended with security mechanisms in a separate development phase. In an ideal setting, this security enhancement of the system would just follow our development phase that introduces fault tolerance.

In the introduction, we started our argumentation by the observation that fault tolerance mechanisms can draw the attention away from the actual application logic. In the state machines as in Fig. 16, this is clearly the case: The original logic of an access control system is not immediately obvious from it. With our method, however, this state machine representation is generated automatically, and does not have to be understood by humans. Instead, systems are specified in the form shown in Fig. 14, where separate functions are represented by separate blocks, which can be studied in isolation, like the one in Fig. 11. So far, our approach has been tested on several academic examples. For its suitability for real-sized systems, we rely on the scalability of the underlying development method SPACE. Consequently, we have started a larger evaluation based on an industrial system, within the project ARCTIS V, supported by the Research Council of Norway.

8. ACKNOWLEDGEMENTS

The research presented in this paper is partially funded by the Research Council of Norway (projects *ARCTIS V* and *ISIS*).

9. REFERENCES

[1] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Logic in Computer Science, 1993. LICS '93., Proceedings of Eighth Annual IEEE Symposium on*, pages 160–170, Jun 1993.

[2] P. A. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Inf. Comput.*, 130(1):71–90, 1996.

[3] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[4] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.

[5] H. A. Berg. UML Building Blocks for Android Location Services. Project Thesis, December 2009. Norwegian University of Science and Technology, Trondheim, Norway.

[6] A. Berlizev and N. Guelfi. Fault Tolerance Requirements Analysis Using Deviations in the CORRECT Development Process. In *Methods, Models and Tools for Fault Tolerance*, pages 275–296. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] N. Bertrand and P. Schnoebelen. Model Checking Lossy Channels Systems Is Probably Decidable. In *Foundations of Software Science and Computation Structures*, pages 120–135. Springer Berlin / Heidelberg, 2003.

[8] A. Bucchiarone, H. Muccini, and P. Pelliccione. Architecting fault-tolerant component-based systems: from requirements to testing. *Electronic Notes in Theoretical Computer Science*, 168:77 – 90, 2007. Proceedings of the Second International Workshop on Views on Designing Complex Architectures (VODCA 2006).

[9] Z. Cui, L. Wang, X. Li, and D. Xu. Modeling and integrating aspects with uml activity diagrams. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 430–437, New York, NY, USA, 2009. ACM.

[10] P. Domokos and I. Majzik. Design and analysis of fault tolerant architectures by model weaving. In *HASE '05: Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems*

Engineering, pages 15–24, Washington, DC, USA, 2005. IEEE Computer Society.

[11] V. Ermagan, I. H. Kruger, and M. Menarini. Model-Based Failure Management for Distributed Reactive Systems. In F. Kordon and O. Sokolsky, editors, *13th Monterey Workshop, Composition of Embedded Systems, Scientific and Industrial Issues*, number 4888-2007 in LNCS. Springer, Paris, France, December 2007.

[12] L. Fuentes and P. Sánchez. Towards executable aspect-oriented uml models. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 28–34, New York, NY, USA, 2007. ACM.

[13] 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.

[14] L. A. Gunawan, P. Herrmann, and F. A. Kraemer. Towards the Integration of Security Aspects into System Development using Collaboration-Oriented Models. In *Security Technology. Proceedings of the 2009 International Conference on Security Technology (SecTech 2009), Jeju Island, Korea, December 10-12, 2009*, volume 58 of *Communications in Computer and Information Science*, pages 72–85. Springer Berlin Heidelberg, 2009.

[15] S. Haugsrud. Developing Android Applications with Arctis. Master's thesis, Norwegian University of Science and Technology, June 2009.

[16] A. Herstad, E. Nersveen, H. Samset, A. Storsveen, S. Svaet, and K. E. Husa. Connected Objects: Building a Service Platform for M2M. In *Beyond the Bit Pipe. Proccedings of the 13th ICIN Conference*, 2009.

[17] J. Kienzle and R. Guerraoui. AOP: Does it make sense? the case of concurrency and failures. In *ECOOP 2002 – Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 113–121. Springer Berlin / Heidelberg, 2002.

[18] M. Knutsen. Towards Model-Driven Engineering of Android Applications. Project Thesis, December 2009. Norwegian University of Science and Technology, Trondheim, Norway.

[19] F. A. Kraemer and P. Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In *Prooceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, 2007.

[20] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In A. Schürr and B. Selic, editors, *Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (Models), Denver, Colorado, USA, October 4-9, 2009*, volume 5795 of *Lecture Notes in Computer Science*, pages 571–585. Springer-Verlag Berlin Heidelberg, 2009.

[21] F. A. Kraemer, V. Slåtten, and P. Herrmann. Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In A. Bilgic, R. Gotzhein, and R. Reed, editors, *SDL 2009*, volume 5719 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag Berlin Heidelberg, 2009.

[22] F. A. Kraemer, V. Slåtten, and P. Herrmann. Tool support for the rapid composition, analysis and implementation of reactive services. *Journal of Systems and Software*, 82(12):2068 – 2080, 2009.

[23] S. S. Kulkarni and A. Arora. Automating the Addition of Fault-Tolerance. In *FTRTFT '00: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, London, UK, 2000. Springer-Verlag.

[24] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), June 2003.

[25] S. Mustafiz, J. Kienzle, and A. Berlizev. Addressing degraded service outcomes and exceptional modes of operation in behavioural models. In *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 19–28, New York, NY, USA, 2008. ACM.

[26] Object Management Group. Unified Modeling Language: Superstructure, Version 2.2, February 2009.

[27] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[28] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[29] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 499–508, June 1995.

[30] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.