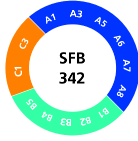
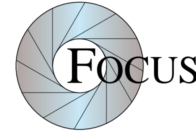
Modeling Faults of distributed, reactive Systems ⋆



Max Breitling

Institut für Informatik
Technische Universität München
D-80290 München, Germany
`http://www.in.tum.de/~breitlin`



Abstract. Formal methods can improve the development of systems with high quality requirements, since they usually offer a precise, non-ambiguous specification language and allow rigorous verification of system properties. Usually, these mainly abstract specifications are idealistic and do not reflect faults, so that faulty behavior - if treated at all - must be specified as part of the normal behavior, increasing the complexity of the system. It is more desirable to distinguish normal and faulty behavior, making it possible to reason about faults and their effects.
In this paper the notions of faults, errors, failures, error detection, error messages, error correcting components and fault tolerance are discussed, based on a formal model that represents systems as composition of interacting components that communicate asynchronously. The behavior of the components is described by black-box properties and state transition systems, with faults being modeled by modifications of the properties or transitions.

1 Introduction

One of the goals of software engineering is the development of *correct* software. Correctness needs to be defined, usually by a *specification* that describes the system to be constructed in a precise and unambiguous way. The most rigorous approach to establishing the correctness of the system under consideration are formal methods, which allow us to *prove* that the system indeed meets its specification.

Nevertheless, systems developed using formal methods can still fail: subcomponents can be unreliable, some (possibly undocumented) assumptions turn out to be invalid, or the underlying hardware simply fails. It can be argued that this was caused by mistakes introduced during the formal development, e.g. by making too idealistic assumptions about the environment. In this paper, we explore another approach: We embed the notion of a *fault* in the context of formal methods, targeting two major goals:

– Support for the development of *fault-tolerant systems*, requiring a precise definition of faults and errors.

⋆ This work is supported by the DFG within the Sonderforschungsbereich 342/A6.

- Reduction of the complexity of formal development by allowing a methodological separation of normal and faulty behavior. After the fault-free version of the system is developed, the possible faults and appropriate countermeasures can be integrated seamlessly in the system.

To model faults already at the level of specifications could sound contradictory, because the specification is intended to describe the *desired* behavior, and nobody wants faults! But in an early development phase it is normally unknown which faults can occur in a system, simply because it is even still unknown what components will be used and how they can fail. Nevertheless, certain kinds of faults can be anticipated already during system development in general, as e.g. by experience or for physical reasons: a transmission of a message can, for instance, always fail. If these faults can be treated already at an abstract level by a general fault handling mechanism, it is sensible to describe the faults already within the specification, and not postpone it to a later phase in the development process.

In this paper, we enrich the model of FOCUS with the notions of faults, errors, failures and fault-tolerance and discuss their connections and use. Since FOCUS offers methodological support for specifying and verifying reactive systems including a formal foundation, description techniques, a compositional refinement calculus and tool support, we expect benefits when FOCUS is combined with results from the area of fault-tolerance. While most other approaches are concerned mainly with foundations of fault tolerance, we try to keep an eye on the applicability for users that are not experts in formal methods. Therefore, our long-term target - not yet reached - are syntactic criteria for certain properties instead of logical characterizations, diagrams instead of formulas, and easy-to-use recipes how to modify systems to their fault-tolerant versions.

In the next section, we describe very briefly our system model of distributed, interacting, reactive components. In Section 3 we introduce faults as modifications of systems. Section 4 contains a discussion how the formal definitions can be used to describe fault assumptions, and detect, report and correct faults. In the last section we conclude and discuss future work.[1]

2 System Model

Our system model is a variant of the system model of FOCUS [5, 6]. A system is modeled by defining its interface and its behavior. The system's interface is described by the (names of the) communication channels with the types of the messages that are sent on them. The (asynchronous) communication along all channels is modeled by (finite or infinite) message streams. The behavior of a system is characterized by a relation that contains all possible pairs of input and output streams. This relation can be described in (at least) two ways on different abstraction levels.

[1] Due to lack of space, all examples are omitted but can be found in an extended version of this paper on the author's homepage.

A *Black Box Specification* defines the behavior relation by a formula Φ with variables ranging over the input and output streams. The streams fulfilling these predicates describe the allowed black-box-behavior of a system. We can use several operators to formulate the predicates, as the prefix relation $\sqsubseteq$, the concatenation of streams $\frown$ and the expression $s.k$ for the k-th element of a stream s, to mention just a few [5].

A more operational *State-Based View* is offered by *State Transition Systems* (STS) that describe the behavior in a step-by-step manner: Depending on the current state, the system reads some messages from the input channels, and reacts by sending some output and establishing a successive state. A STS is defined by its internal variables with their types, an initial condition, a set T of transitions and T^{ϵ} of environment transitions, precisely formalized in [3]. The possible behaviors of a system are described by the set $\langle\langle\mathcal{S}\rangle\rangle$ containing all executions ξ of the system. Executions are defined in the usual way as sequences of states α. A STS can be defined in a graphical or tabular notation.

Both views on systems can be formally connected: An infinite execution of a STS defines least upper bounds for the message streams that are assigned to the input/output channels, and therefore establishes a black-box relation. In [3, 4] the language, semantics and proof techniques are investigated in detail.

Focus offers notions for *composition* and *refinement* supporting a top-down development of systems. The behavior of a composed system $\mathcal{S}_1 \otimes \mathcal{S}_2$ can be derived from the behavior of its components. The interface refinement $\mathcal{S}_1 \overset{R_I \triangleright R_O}{\rightsquigarrow} \mathcal{S}_2$ states that the executions of $\mathcal{S}_2$ are also executions of $\mathcal{S}_1$ with modifications at the interface described by the relations R_I, R_O. Compositionality ensures that refining a systems component means refining the overall system.

3 Modifications and Faults

Intuitively, faults in a system are connected with some discrepancy between an intended system and an actual system. To be able to talk about faults, their effects and possible countermeasures, we need a clear definition of the term *fault*. We suggest to identify faults with the *modifications* needed to transform the correct system to its faulty version.

In this section, we define modifications of systems, both for the black-box and the operational view, and base the notions of fault, error and failure on these modifications.

3.1 Modifying a System

In the process of adapting a specified system to a more realistic setting containing faults, we have to be able to change both the interface and the behavior.

Interface modifications We allow the extension of a type of a channel and the introduction of new channels. The behavior stays unchanged if the specification is adjusted so that it ignores new messages on new input channels, while it may behave arbitrarily on new output channels. For development steps towards a

fault-tolerant system it is normally expected that the behavior does not change in the case faults do not occur. Therefore we are interested in criteria for behavior maintenance that are easy to be checked. For interface modifications, these criteria can be defined syntactically according to the description technique used, as e.g. black-box formulas, tables or state machines. We do neither allow the removal of channels nor a type restriction for a channel, because this could easily lead to changes of the behavior. A change of the types for the channels follows the idea of *interface refinement*. Under certain conditions, these changes maintain (the properties of) the behavior. In this paper, we will not investigate this topic.

Behavior modifications A fault-affected system normally shows a different behavior than the idealistic system. Instead of describing the fault-affected system, we focus on the *difference* of both versions of the system and suggest a way to describe this difference for black-box views and state machines.

Having Φ as the black-box specification of the fault-free system, we need to be able to strengthen this predicate to express further restrictions, but also to weaken it to allow additional I/O-behaviors. We use a pair of formulas $\mathcal{M} = (\Phi_E, \Phi_F)$ and denote a modified system by

$$\Phi \triangle \mathcal{M} \quad \text{(read: } \Phi \textit{ modified by } \mathcal{M}\text{)}$$

whose black-box specification is defined by

$$(\Phi \wedge \Phi_E) \vee \Phi_F$$

The neutral modification is denoted by $(\mathsf{true}, \mathsf{false})$, and the modification towards an arbitrary Ψ is expressed by (false, Ψ).

For a state-based system description, we express modifications of the behavior by modifications of the transition set (as e.g in [1, 9, 12]). Obviously, we can add or remove transitions and define a behavior-modification $\mathcal{M}$ by a pair (E, F) of two sets of transitions. The set E contains transitions that are no longer allowed in an execution of the modified system. The set F contains additional transitions. The transitions in F can increase the nondeterminism in the system, since in states with both old and new transitions being enabled, the system has more choices how to behave. We can use F to model erroneous transitions the system can spontaneously take. The executions of a modified system are defined by

$$\langle\!\langle \mathcal{S} \triangle \mathcal{M} \rangle\!\rangle \stackrel{\text{df}}{=} \{\xi \mid (\xi.k, (\xi.k+1)) \in (T \setminus E) \cup F \ \cup \ T^{\epsilon}\}$$

i.e. a non-environment transition has to be in F or in T but not in E. In this formalism, $(\varnothing, \varnothing)$ is the neutral modification, and choosing E to contain all transitions and F as arbitrary set of transitions shows that this formalism is again expressive enough.

It is an interesting but open question if and how both notions for modifications can be connected. If Φ is a property of a STS $\mathcal{S}$, and both are modified in a similar way, then $\Phi \triangle (\Phi_E, \Phi_F)$ should be the modified property of the modified system $\mathcal{S} \triangle (E, F)$. Similar approaches and partial results are discussed in [2, 7, 13].

3.2 Combining Modifications

To explore the effect of multiple modifications, we define the composition of modifications. For black-box specifications, the operator + combines two modifications (Φ_E^i, Φ_F^i) of a system $(i = 1, 2)$, assuming $\Phi_F^1 \Rightarrow \Phi_E^2$ and $\Phi_F^2 \Rightarrow \Phi_E^1$, to one modification by

$$(\Phi_E^1, \Phi_F^1) + (\Phi_E^2, \Phi_F^2) \stackrel{\text{df}}{=} (\Phi_E^1 \wedge \Phi_E^2, \Phi_F^1 \vee \Phi_F^2)$$

We reuse the operator + for transition systems, and define for (E_i, F_i), assuming $E_1 \cap F_2 = \varnothing$ and $E_2 \cap F_1 = \varnothing$, the combination

$$(E_1, F_1) + (E_2, F_2) \stackrel{\text{df}}{=} (E_1 \cup E_2, F_1 \cup F_2)$$

The assumptions avoid confusion about executions resp. transitions that are added by one modification but removed by the other, and asserts the following equalities, with S representing Φ resp. $\mathcal{S}$:

$$S \triangle (\mathcal{M}_1 + \mathcal{M}_2) \quad = \quad (S \triangle \mathcal{M}_1) \triangle \mathcal{M}_2 \quad = \quad (S \triangle \mathcal{M}_2) \triangle \mathcal{M}_1$$

We can use this operator to express combinations of faults for defining the notion of fault-tolerant systems.

For a composite system $S = S^1 \otimes S^2$ we can derive the modification of this system from the modifications of its constituents, and can calculate the impact of a fault of a component upon the overall system. For black-box specifications, we define the derived modification of the system by

$$\Phi_E = \Phi_E^1 \wedge \Phi_E^2 \qquad\qquad \Phi_F = (\Phi^1 \wedge \Phi_F^2) \vee (\Phi^2 \wedge \Phi_F^1) \vee (\Phi_F^1 \wedge \Phi_F^2)$$

For modifications of the transition sets of the components, we can define $\mathcal{M} = (E, F)$ with ($\barwedge$ denotes the pairwise conjunction of elements of both sets)

$$E \stackrel{\text{df}}{=} E_1 \barwedge T_2^{\epsilon} \ \cup \ T_1^{\epsilon} \barwedge E_2 \quad \text{ and } \quad F \stackrel{\text{df}}{=} F_1 \barwedge T_2^{\epsilon} \ \cup \ T_1^{\epsilon} \barwedge F_2$$

With the same assumptions for the component's modifications as above, this results for both formalisms in

$$\mathcal{S} \triangle \mathcal{M} = (\mathcal{S}_1 \triangle \mathcal{M}_1) \otimes (\mathcal{S}_2 \triangle \mathcal{M}_2)$$

3.3 Faults, Errors and Failures

In the literature the meaning of the terms *fault*, *error* and *failure* is often described just informally (e.g. [10, 11]). In our setting, we can define these notions more precisely.

The *faults* of a system are the *causes* for the discrepancy between an intended and actual system. Therefore, it makes sense to call the transitions of a modification $\mathcal{M}$ the faults of a system. What is called a fault of a system cannot be decided by looking at an existing system alone; this normally depends on

the intended purpose of the system, on an accepted specification and even on the judgment of the user or developer. What one person judges as fault, the other calls a feature. The definition of modifications given in the previous sections is intended to offer a possibility to document that decision, and explicitly represent the faults in a modified system. Of course, the modified system could be described by one monolithic specification without reflecting the modifications explicitly, but it is exactly this distinction between "good" and "bad" transitions that allows our formal definitions.

A fault can lead to an erroneous state, if an existing faulty transition is taken during an execution of the system. We define a state α to be an *error* (state) if this state can only be reached by at least one faulty transition. The set of errors of a system $\mathcal{S}$ under the modifications $\mathcal{M} = (E, F)$ is defined as

$$\mathit{ERROR}(\mathcal{S}, \mathcal{M}) \stackrel{\text{df}}{=} \{\alpha \mid \forall\, k \in \mathbb{N}, \xi \in \langle\!\langle \mathcal{S} \Delta \mathcal{M} \rangle\!\rangle \bullet \xi.k = \alpha \Rightarrow \exists\, l < k \bullet (\xi.l, \xi.(l+1)) \in F\}$$

Note that all unreachable states are error states, and the set E enlarges the set of unreachable states. The set of correct states can be defined as the set of valuations that can be reached by normal transitions (in T) only. As long as we do not require $F \cap T = \varnothing$, it is possible that states are both correct states and error states. We cannot sensibly define errors for the black-box view, since neither states nor internals do exist in that context.

A *failure* is often defined as a visible deviation of the system relative to some specification. Since we can distinguish the inside and outside of systems, we can also reflect different visibilities of errors. Our definition of a failure depends on the kind of specification: If we regard a black-box specification Φ as the specification of a system, a failure occurs in a state α if the property gets violated in that state. But we can also define a failure if the unmodified STS $\mathcal{S}$ is understood as specification, and $\mathcal{S} \Delta \mathcal{M}$ as faulty system. An error state α is additionally called a failure if all states with the same visible input/output behavior are error states:

$$\mathit{FAILURE}(\mathcal{S}, \mathcal{M}) \stackrel{\text{df}}{=} \{\alpha \mid \forall\, \beta \bullet \beta \stackrel{I \cup O}{=} \alpha \Rightarrow \beta \in \mathit{ERROR}(\mathcal{S}, \mathcal{M})\}$$

Two valuation α and β coincide on a set of variables V, if they assign the same value to all variables in V, i.e. $\alpha \stackrel{V}{=} \beta \Leftrightarrow \forall\, v \in V \bullet \alpha.v = \beta.v$.

3.4 Internal vs. External Faults

Up to this point, we focused on internal faults: The behavior deviation resp. the faulty transitions occurred *inside* the system. But a system can also suffer from faults taking place *outside* a system, i.e. in its environment. A discussion of failures of the environment requires explicit or implicit assumptions about its behavior. An explicit assumption can be formulated in the context of black-box views by a formula that describes the assumed properties of the input streams. If this assumption is not fulfilled, the system's behavior is usually understood to be not specified so that an arbitrary, chaotic behavior may occur. We think

this situation relates to an external fault, and should be treated by a reasonable reaction of the system instead of undefined behavior. We need further methodological support offering notions of refinement for these cases: Given an assumption/guarantee specification A/G, we need to be able to weaken A and adapt G so that the original behavior stays untouched if no external faults occur, but a sensible reaction is defined if they do.

The type correctness of the input messages can be regarded as another explicit assumption about the environment. If the interface is changed so that new messages can be received, we have to refine the behavior of the system in an appropriate way.

If the system is specified by a STS, but no explicit environment assumptions are defined, we can nevertheless try to find implicit assumptions. If the system is in a certain state, it is normally expected that at least one of the transitions should be eventually enabled. It some cases, it can indeed be meant that a system gets stuck in certain situations, but normally a weak form of liveness is wanted: The inputs should finally be consumed, and a state where a system gets stuck is a kind of error state with invalidated liveness. We regard these questions and the distinction of internal and external faults as an interesting area for future research.

4 Dealing with Faults

Introducing a formal framework for formalizing faults needs to be accompanied with some methodological advice how the formalism can be used. In this section, we discuss how fault occurrences and dependencies between fault models can be expressed by virtual components, mention requirements for error detection and the introduction of error messages and define fault-tolerance.

4.1 Refined Fault Models

To describe a system with certain faults, we can modify a system accordingly by adding fault transitions. In specific cases, these modifications could change the behavior too much, since these transitions can be taken whenever they are enabled. Sometimes, we want to express certain fault assumptions that restrict the occurrence of faults. For example, we would like to express that two components of a system can fail, but never both of them at the same time, or we want to express probabilities about the occurrence of faults, e.g. state that a transition can fail only once in n times, for some n.

To be able to formalize these fault assumptions, we suggest to introduce additional input channels used similar to *prophecies*. The enabledness of the fault transitions can be made dependent on the values received on these prophecy channels. We can then add an additional component that produces the prophecies that represent the fault assumption. During the verification, these virtual components and prophecy channels can be used as if they were normal components, even though they will never be implemented.

4.2 Detecting Errors

Error detection in our setting consists, in its simplest case, of finding an expression that is true iff the system is in an error state. The system itself must be able to evaluate this expression, so that this expression can be used as a precondition for error-correcting or -reporting transitions.

An easy way to detect errors is a modification of the fault transitions so that every fault transition assigns a certain value to an error-indicating variable. For example, a fault transition can set the variable *fault* to true, while normal transitions leave this variable unchanged, as suggested in [12]. But this approach assumes the fault transitions to be controllable, which is in general not the case: The faults are described according to experiences in the real world, e.g. messages are simply lost from a channel without any component reporting this event. We could change this lossy transition to one that reports its occurrence, but this new variable *fault* may only be used in proofs for investigating the correctness of the detection mechanism, but this is not a variable that is accessible by the system itself. We have to deal with given faults described by modifications that we must accept untouched, but nevertheless we want to detect them.

We suggest a way to handle errors that can be detected by finding inconsistencies in the state of the system. The consistency can be denoted as a formula Ψ that is an invariant of the unmodified system. It can be proved to be an invariant by the means of [3]. We can then remove all transitions with $\neg \Psi$ as precondition (via E) and add a new error reacting transition with an intended reaction (via F). Normally, a system occasionally contains transitions that are enabled if $\neg \Psi$, simply because a set of transitions can be indifferent to unspecified properties. Such a modification does not change the original system, but allows the specification of reactions, e.g. by sending an error message.

This approach is conceptually the easiest way, since error detection is immediate, but it is not always realistic. In [1] a more general approach is presented, that also allows delayed error detection. We have to integrate this idea also in our stream-based setting, being specially interested in a notion of a delayed detection that still occurs before an error becomes a failure.

4.3 Error Messages

Once we enabled a system to detect an error, we want it to react in an appropriate way. If errors cannot be corrected, they should at least be reported. Sending and receiving of error messages has to be integrated in the system without changing its fault-free behavior.

In Section 3.1 we already saw that by adding an additional output channel, with arbitrary messages sent, the behavior will only be refined. So, extending a system to send error reporting messages is easy: We can add a transition that sends an error message in the case an error is detected while it leaves all other variables in V unchanged, and we refine the other transitions to send no output on this channel.

We also want to react to error messages from other components. Therefore, we must be able to extend a component by a new input error message channel, and adapt the component to read error messages and react to them. A further transition in the system that reads from the new channel and reacts to it can easily be added while other transitions simply ignore the new channel.

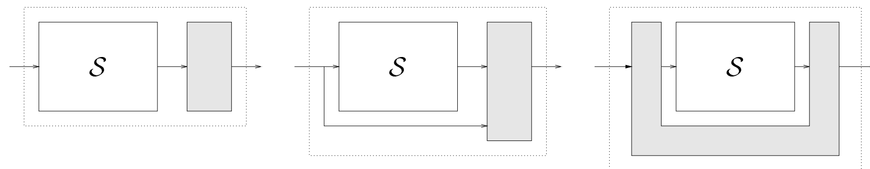
4.4 Correcting Faults

We described ways how a system can be modified to contain anticipated faults already at the abstract level of specifications. The deviations of such a modified system can show different degrees of effect: The effects of the faults are harmless and preserve the properties of a specification, or the faults show effects that violate the specification, but they are correctable, or the faults lead to failures that are not correctable. The first case is of course the easiest since no countermeasures have to be taken for the system to fulfill its specification. In the last case, faults can only be detected and reported, as described in the previous sections.

For correctable faults the system usually must be extended by mechanisms that enable the system to tolerate the faults. Several mechanisms are known, implementing e.g. fail-stop behavior, restarts, forward or backward recovery, replication of components, voters and more. All of these are *correctors* in the sense of [1].

A methodology supporting the development of dependable systems should offer *patterns* that describe when and how these mechanisms can be integrated in a specified system, together with the impact on the black-box properties. For example, a fail-stop behavior can be modeled by introducing a new trap state that was not yet reachable before, and that does not consume or generate any messages, while safety properties are not compromised.

There is a special case of (local)correction of faults that can be done by new components in a system that catch the effect of faults of a component before they spread throughout the system. These new components, that we call *drivers*, are placed between the fault-affected component and the rest of the system. Depending on the characteristics and severity of the faults, the driver controls just the output of the component, or controls the output with the knowledge of the input, or even controls input and output, as showed in the following figure. The last variant is the most general one, and could tolerate arbitrary failures by totally ignoring the faulty component and simulating its correct functionality.



Since we already know how to specify components and how to compose components to systems, fault correction can be integrated as an ordinary development step, so that results concerning methodology [5], tool support [8] and proof support [3, 4] can be used.

4.5 Fault-tolerance

Usually, fault-tolerance is interpreted as the property of a system to fulfill its purpose despite the presence of faults in the system, but also in their absence (as pointed out e.g. in [9]). In our formalism, this could be expressed by the following monotonicity property, stating that all partial modifications of a system should maintain a certain property.

$$\forall (E', F') \bullet E' \subseteq E \wedge F' \subseteq F \quad \Rightarrow \quad \mathcal{S} \vartriangle (E', F') \models \Phi$$

We think this condition is too strong, since too many partial modifications have to be considered. Assume a fault - being tolerable - that can be modeled by a change of a transition, expressed by removing the old and adding the new transition. If we just add the new one, but do not remove the old transition, we have a partial modification that could never happen in practice but results in a system with intolerable faults. Partial modifications are too fine-grained if they are based on single transitions.

We suggest that a statement about fault-tolerance must be made explicit by specifying the faults and combinations of faults for which the system should have certain properties. As opposed to other approaches [9, 12], a modification (with a nonempty E) can change a system so that it cannot show any execution of the original system. So, if a property is valid for the modified system, it is possibly not valid for the unmodified system.

In our setting, explicit fault-tolerance can be expressed by generalizing our expressions to allow sets of modifications. The following expression is defined to be valid if $\forall\, i \bullet \mathcal{S} \vartriangle \mathcal{M}_i \models \Phi$.

$$\mathcal{S} \vartriangle \{\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots\} \models \Phi$$

For a statement about fault-tolerance, the empty modification $(\varnothing, \varnothing)$ has to be contained in the modification set, and the desired combinations of modifications must be explicitly included. The induced number of proof obligations needs further methodological support.

5 Proving Properties

The additional effort imposed by the use of formal methods for formalizing a system is rewarded by the possibility to *prove* that the systems have certain properties. While many formalisms offer this possibility theoretically, it is also important to offer methodology to find the proofs. In [3, 4] we presented a way of proving properties for our system model, using proof rules, quite intuitive diagrams and tool support.

It is crucial for a successful methodology that proofs can be found with reasonable effort. For fault-tolerance, it is desirable that proof obligations can be shown in a modular way. Results for an unmodified system should be transferred to *modified results* for the *modified system*. If properties of the correct system

are already shown, this result should not be invalidated totally by modifying the system so that the verification has to start again from scratch. The existing proof should only be adapted accordingly, reflecting the modifications, using already gained results.

So it seems to be an interesting research topic to find notions for modifying a proof. Since a proof can be represented by a proof diagram, it can be promising to investigate modifications of proof diagrams. If a transition is removed (by E), a safety diagram stays valid also without this transition. In a liveness diagram, new proof obligations emerge in this case, since the connectivity of the graph must be checked again. Adding a transition via F will - in most cases - destroy the validity of a safety diagram, and will even introduce new nodes. These new nodes have to be checked relative to all other transitions of the system, and they will also appear in the liveness diagram, leading to a bunch of additional proof obligations there. Nevertheless, parts of the diagram stay unchanged and valid, representing a reuse of the existing proof.

6 Conclusions

This paper discusses how faults can be modeled in the context of distributed systems, composed of components that interact by asynchronous message passing. We have shown how the behavior of such systems can be specified, using an abstract black-box view or an operational state-based view. Faults of a system are represented by the modifications that must be applied to the correct system to obtain the faulty system. Modifications can change both the interface and the behavior. For a modified system we can characterize its error states and failures. Once the faults resp. modifications of a system are identified, the ways how errors can be detected, reported, corrected and tolerated are also discussed, mostly informally, in this paper.

Future Work The topic of the formal development – including the specification, verification, and stepwise refinement – of fault tolerant systems is not yet explored to a satisfying degree with concrete help for developing systems with faults. It is a challenging task to combine various results found in literature with this paper's approach based on message streams and black-box views. An ideal formal framework combines the benefits of different approaches, and offers solutions to several aspects as formal foundation, methodology and verification support.

For a framework to be formal, precise definitions for all notions must be defined. We need a formal system model that is enriched by notions for faults and their effects, errors, failures, changes of interfaces and internals, fault assumptions, adaption of properties to modifications of the system, composition and refinement of faults. But a language to express statements about fault-affected or -tolerant systems is not enough, some methodological advice for its use is also needed, offering ideas how to use this language: When and why should faults be described, how can we refine a system to stay unchanged in the fault-free

case, but improve its fault tolerance in the presence of faults? Formal methods allow for formal verification. This has to be supported by suitable proof rules, but even this is not enough: We also need description techniques for proofs and tool support for generating proof obligations and finding and checking proofs. Finally, only convincing case studies are able to show a recognizable benefit of the idea to formally develop fault-tolerant systems.

References

1. Anish Arora and Sandeep Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *IEEE Transactions on Software Engineering*, 1999.
2. Max Breitling. Modellierung und Beschreibung von Soll-/Ist-Abweichungen. In Katharina Spies and Bernhard Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme. FBT'99*, pages 35–44. Herbert Utz Verlag, 1999.
3. Max Breitling and Jan Philipps. Step by step to histories. In T. Rus, editor, *AMAST2000 - Algebraic Methodology And Software Technology*, LNCS 1816, pages 11–25. Springer, 2000.
4. Max Breitling and Jan Philipps. Verification Diagrams for Dataflow Properties. Technical Report TUM-I0005, Technische Universität München, 2000.
5. Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems - FOCUS on Streams, Interfaces and Refinement.* Springer, 2000. To appear.
6. Homepage of FOCUS. http://www4.in.tum.de/proj/focus/.
7. Felix C. Gärtner. A survey of transformational approaches to the specification and verification of fault-tolerant systems. Technical Report TUD-BS-1999-04, Darmstadt University of Technology, Darmstadt, Germany, April 1999.
8. Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus - A Tool for Distributed Systems Specification. In *FTRTFT'96*, LNCS 1135, pages 467–470. Springer, 1996.
9. Tomasz Janowski. On bisimulation, fault-monotonicity and provable fault-tolerance. In *6th International Conference on Algebraic Methodology and Software Technology.* LNCS, Springer, 1997.
10. J.C. Laprie. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems.* Springer, 1992.
11. P.A. Lee and T. Anderson. *Fault Tolerance - Principles and Practice.* Springer, second, revised edition, 1990.
12. Zhiming Liu and Mathai Joseph. Specification and verification of recovery in asynchronous communicating systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 137 – 166. Kluwer Academic Publishers, 1993.
13. Doron Peled and Mathai Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 1994.

Acknowledgments I am grateful to Ingolf Krüger and Katharina Spies for inspiring discussions and their comments on this paper, and thank the anonymous referees for their very detailed and helpful remarks.