A Case Study in the Mechanical Verification of Fault Tolerance

Heiko Mantel
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany
mantel@dfki.de

Felix C. Gärtner[*]
Department of Computer Science
Darmstadt University of Technology
D-64283 Darmstadt, Germany
felix@informatik.tu-darmstadt.de

Abstract

To date, there is little evidence that modular reasoning about fault-tolerant systems can simplify the verification process in practice. We study this question using a prominent example from the fault tolerance literature: the problem of reliable broadcast in point-to-point networks opposed to crash failures of processes. The experiences from this case study show how modular specification techniques and rigorous proof re-use can indeed help in such undertakings.

Introduction

A system is said to be *fault-tolerant* if it exhibits a well-defined system behavior in the presence of faults. The importance of fault-tolerant systems stems from their omnipresence throughout today's technical infrastructure. The failure of a critical computer system can have catastrophical consequences, resulting in loss of considerable industrial value or even loss of human life. Thus, there is an increasing need for systems with verifiable fault-tolerance properties.

Because it is necessary to precisely describe faulty behavior and its interaction with normal system operation, fault-tolerance considerations place an additional complexity burden on a formal verification process. In theory, the additional complexity can be dealt with by first reasoning about the system in fault free environments and — after placing it into a faulty environment — reasoning only about those aspects of the system which have changed (Gärtner 1999). Many case studies exist which prove certain algorithms correct in the case of faults using theorem provers like PVS (Lincoln & Rushby 1993; Qadeer & Shankar 1998). However, these case studies do not exploit the theoretical ideas sketched above and thus the proofs seem more complex than they need to be. Consequently, there is little evidence to date that such modular reasoning can indeed simplify the verification process using theorem provers in practice.

The basic notion underlying most of the theory behind modular reasoning about fault-tolerant systems is that of a *transformation* (Gärtner 1999). Today, there exists a solid basis of elegant transformational techniques in the literature (Peled & Joseph 1994; Liu & Joseph 1992; Arora & Kulkarni 1998). However, the examples used to show the usefulness of these theories have been rather small and academic. To the best of our knowledge, the only real case study which has been performed using theorem provers is the component-based mechanical verification of a self-stabilizing mutual exclusion protocol by Kulkarni, Rushby and Shankar (1999). While it shows that modular reasoning does have advantages, it also concludes that — being the first such case study — more experiences are obviously needed.

Using the industrial-strength Verification Support Environment (VSE) (Hutter *et al.* 1996), we study the benefits of modular reasoning in fault-tolerance using another prominent example from the fault tolerance literature: the problem of reliable broadcast in point-to-point networks with crash failures (Hadzilacos & Toueg 1994). Again, the formalization and hand-written proofs of correctness have appeared in the literature (Hadzilacos & Toueg 1994; Gärtner 1998), but to our knowledge there has been no attempt to study whether the methodologies involved really lend themselves to rigorous mechanical verification. So neither are we presenting a new algorithm, nor are we presenting a new tool or proof method; we are proposing a modular specification method and evaluate it in practice.

We begin by presenting a formal specification of broadcast in point-to-point networks and — by this example — introduce the VSE system. Subsequently, we transform the broadcast system into one which is opposed to crash faults of processes and prove its correctness. It turns out that transformational reasoning can indeed lessen the complexity of the verification task; the benefits lie not so much in simplification of the proofs (they still remain lengthy and cumbersome), but rather in the massive potential of rigorous re-use of subspecifications and proofs. Finally, the results are summarized and their impacts are discussed.

Formal Specification of Broadcast

Informally, *broadcasting* a message in a distributed system means to send the same message to all nodes in the network. Usually it is defined using two primitive operations *broadcast* and *deliver*. As such primitives are often not

[*]This author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the PhD program (Graduiertenkolleg) “Intelligente Systeme für die Informations- und Automatisierungstechnik” at Darmstadt University of Technology.

directly supported by the communication system (as is the case, e.g., in wide area networks), it must be implemented using low-level operations like *send* and *receive* of individual messages. Broadcast is an important concept easing, among others, the design of observation and control mechanisms in distributed systems. The broadcast algorithm we consider is the well-known algorithm by Hadzilacos and Toueg (1994). We build upon a prior formalization of it by Gärtner (1998) which, however, was done in the UNITY theory (Chandy & Misra 1988). We briefly recall the main points before we describe the formalization in VSE.

Overview of the broadcast system. The broadcast system consists of n processes which communicate using a point-to-point network of reliable unidirectional channels. The processes need not be fully connected but it is required that there is a communication path between every two processes.

The broadcast algorithm is described by six guarded commands (denoted B_1–B_5 and B_1') which are executed by each process locally. The state of each process consists of three buffers ($\tilde{b}$, b, and d), each of which can hold a single message, and a multiset of messages (D). In order to broadcast a message m, a process places it into its broadcast buffer $\tilde{b}$ (command B_1). This corresponds to the invocation of *broadcast*(m). Before delivery, the message is transfered from $\tilde{b}$ to the incoming buffer b (B_1'). If the message has not been delivered by that process before ($m \notin D$) then it is sent on all outgoing channels, added to D, and put into the delivery buffer d (B_3), otherwise, b is cleared (B_4). After a message has been processed, d is cleared (B_5). A message on an ingoing channel is delivered to a process by putting it into b (B_2) from where the process may resume by command B_3 or B_4. The respective guards ensure that none of the buffers are accidently cleared. For details of the algorithm and its formalization in UNITY the reader is referred to Gärtner (1998).

The correctness of a broadcast system is defined by the following properties.

S (safety) Every delivered message was previously broadcast and every message is delivered at most once.

L (liveness) Every message which is broadcast at some process x will eventually be delivered locally, and delivery at x will lead to delivery at all other processes.

Formalization in VSE. We perform our correctness proofs using the *Verification Support Environment* (VSE), a system which is described in detail by Hutter *et al.* (1996). Apart from offering verification methods, VSE explicitly contains means to specify complex systems in a structured way which facilitates modular specification and verification, and supports proof re-use. In this section we will introduce the central concepts of VSE and its specification and verification methodologies along the lines of the broadcast algorithm sketched above.

In VSE, we have modeled a distributed system in a modular bottom-up fashion. The central concept of VSE to support modular specification and verification is the *development graph* consisting of *development objects* and *links* between them. Briefly spoken, development objects are subspecifications and links are special types of relations between such specifications. There are mainly two different types of development objects: *abstract data types* and *state-based systems*. For this case study we have formulated all parts of the system using abstract data types.

Using abstract data types a system is modeled by *algebras*, i.e., structured collections of sorts (sets of values) and sorted collections of functions which operate on these values. Elementary algebraic specifications are called *theories* in VSE and introduce the types (the sorts) and functions (the operations) necessary to describe the modeled system. The semantics of the operations are defined axiomatically (i.e., by a set of axioms which they are supposed to respect) or algorithmically (i.e., by a small program from a restricted programming language). Theories may *import* other theories, which corresponds to making types, operations and the corresponding axioms visible in another theory. This makes it possible to specify systems in a modular fashion.

"Importing" specifications also makes it possible to perform proofs in a modular way. Proofs are always considered local to a specification, i.e., the proof of a lemma within a specification module only depends on the proofs of "imported" theories. Thus, the system can give assistance in providing only *relevant* lemmas when proving a theorem. Furthermore, in VSE the validity of proofs is managed automatically. When a theory has changed, all proofs which rely on that specification are flagged as invalid and have to be proved again.

We have depicted a simplified version of the final development graph of *broadcast* in Fig. 1. On the "lowest-level" the theory `Messages` defines a datatype of messages. Unidirectional channels are modeled as multisets of messages in `UChannel`. There are two operations defined on channels, `send` and `remove` which, respectively, place a message into the channel and remove it again. Channels can be combined to a `ChannelMatrix`. Overall there is an $n \times n$ matrix of channels, whereby a channel of type `nochannel` at position (x, y) of the matrix expresses that there is no connection from process x to process y. This is the basis for local connectivity.

Processes are modeled in module `Processes` as a datatype which has an identification and local data structures ($\tilde{b}$, b, d, and D). Processes can be combined to lists.

The `State` of the entire system consists mainly of the states of all processes (i.e., a `ProcessList`) and the state of all channels (i.e., a `ChannelMatrix`). To specify the assumption that every message is broadcasted only once, we add to the global state a set B of messages which have been broadcast.

The low-level specification `Actions` identifies the possible actions which a process can perform. Using the identifiers defined in `Actions`, traces are modeled as sequences of states and actions. Starting with a state, actions and states alternate within a sequence. Intuitively, a subsequence $\ldots, s_1, a, s_2 \ldots$ models that the execution of action a in state s_1 resulted in state s_2. Note that the set of traces contains *all* possible sequences, i.e., not only those allowed by

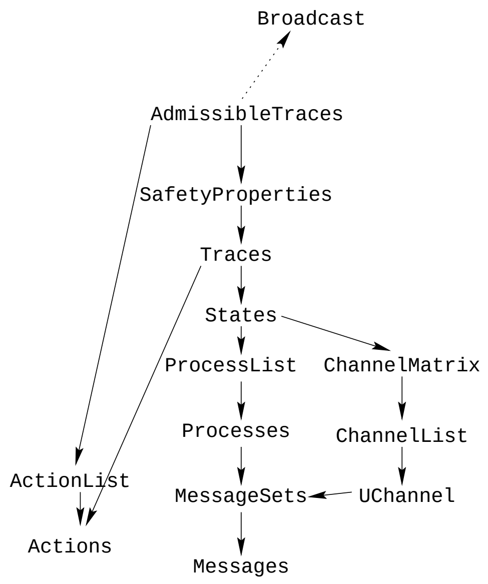

Figure 1: Simplified development graph of *broadcast*.

the algorithm because the guarded commands have not yet been specified. Nevertheless, it is now possible to specify the `SafetyProperties` (S) of the broadcast specification: A state s is safe if for every process in s

S1 the multiset of delivered messages D is a subset of the set B of messages which have been broadcast, and

S2 the multiset of delivered messages D has no duplicates.

Finally, the set of possible traces is restricted to those allowed by respecting the actions of the broadcast algorithm. This is encoded in the specification `AdmissibleTraces`. Here, the action identifiers defined in `Actions` are associated with their corresponding actions, i.e., the guarded commands described informally earlier. An action consists of a boolean expression on the current state (the *guard*) and a description of how the state changes if the action is executed (the *command*). An action may only be executed in some state s if its guard evaluates to true on s. Thus, the specification of admissible traces describes all traces which the algorithm might generate.

At the level of admissible traces it is now possible to formulate the safety specification of broadcast as follows: *S1 and S2 hold for every state in every admissible trace*.

These two properties are formulated as axioms within the safety specification `Broadcast`. The special link between the specification `Broadcast` and `AdmissibleTraces` means that all admissible traces should *satisfy* the `Broadcast` specification. (Note that the relations between the other specifications have up to now merely signified that one specification *imports* the other in the sense described above).

Proof obligations. The *satisfies* relation between `AdmissibleTraces` and `Broadcast` leads to proof obligations which are generated automatically by the system. In order to guarantee correctness all these proof obligations must be proven. Other proof obligations arise from the necessity to prove that the specification in its entirety is *consistent* (i.e., was not self-contradictory). This proof is often not considered being at the heart of the verification task. But proving consistency is usually considered a mandatory part in arguing that a specification is in fact *adequate*.[1] As expected, the work involved in proving consistency was considerable. The number of proof obligations concerned with consistency exceeded the ones concerned with correctness slightly. However, the more difficult proofs were among the correctness proofs in which the safety properties had to be proved by induction and case analysis.

A Transformation from Broadcast to Reliable Broadcast

The broadcast system presented in the previous section was one in which all components were assumed to work correctly. As shown by Hadzilacos and Toueg (1994), their broadcast algorithm will also work correctly in situations where a limited number of components may fail. The description of the number and type of component failures which the algorithm can tolerate (i.e., without failing altogether) is usually called the *fault assumption*. The fault assumption is a necessary starting point for any type of fault tolerance considerations.

The fault assumption under consideration here is usually termed *crash*, meaning that at most $t < n$ processes may at some point in time simply stop executing steps. Fault assumptions can be described formally as transformations (Gärtner 1999). Consequently, at the level of processes, a fault assumption can be formulated as a function F mapping a "fault-free" program A into a "fault-affected" version $A' = F(A)$ (Gärtner 1998). For the crash fault assumption, F will add an additional boolean variable *up* to the state space of every process which is initialized to the value true. Additionally, an action (called the *fault action*) is added to A which sets *up* to false if *up* holds. Finally, *up* is added as a precondition to every remaining action of A. Overall, this means that A' will initially behave like A. However, if the fault action is executed, $\neg$*up* holds, and so the preconditions of all actions become invalid. Hence, the process is not able anymore to perform steps, mimicking a crash.

Since the transformation adds behavior to A, it is obvious that in general the original correctness specification of broadcast (i.e., S and L) must be weakened to reflect this fact. A simple and mechanical way to do this is to restrict the original specification to the behavior of only the correct processes. Obviously, this can also be described using a transformation. A property P for some process x in the

[1]To prove consistency we had to introduce additional nodes into the graph and had to perform some additional proofs. For simplicity, these nodes are omitted from Fig. 1. The complete development graph which can be imported into VSE is available on the Internet at `www.informatik.tu-darmstadt.de/BS/Gaertner/vse/Reliable.out`

original specification is transformed into the weaker property "x is up" $\Rightarrow P$. The resulting specification is called the *tolerance specification*.

The transformational approach makes it possible to re-use much of the given development graph. In our case study we have built both fault-free and fault-affected scenarios into one development graph. The additions made to the original graph of Fig. 1 are shown in Fig. 2. In general, those parts of the specification which need to be altered, are generated from the corresponding parts of the original specification.

In the resulting specification, a crash action has been added to the actions of the broadcast algorithm in `CrashActions`. A theory `UpDownlist` models a list of boolean *up* flags which is used as an additional component of states as defined in `CrashStates`. Using the modified definition of states, traces are modeled exactly like in the fault-free case. The safety properties are weakened by the precondition "x is up" which is expressed using the list of *up* flags. In `CrashAdmissibleTraces`, the crash action allows for additional behaviour and "x is up" is added to the guard of each action.

Our modular specification allows for a precise identification of which parts are affected by the transformation and in which way. Although this transformation has been performed by hand, our case study suggests that it can be automated in the following way: (1) add theories with auxiliary datatypes, (2) add actions for faulty behaviour, (3) add auxiliary datatypes to the state space, (4) weaken the properties, and (5) add conditions to the guards and add faulty behaviour. Note that this approach is not restricted to our case study but should cover the whole family of transformations for fault tolerance as described by Gärtner (1999). The various fault assumptions differ in the auxiliary datatypes, in the actions for faulty behaviour, in how the properties are weakened, and in how the guards are affected.

After the transformation, the proof that the broadcast system running under the crash fault assumption satisfies the tolerance specification, can be done using the same methods as in the fault-free case. If the proof succeeds, we have shown that the broadcast algorithm is fault-tolerant regarding the crash fault assumption for the given tolerance specification. Again, the transformational approach offers much potential for re-use. Proofs which belong to unchanged parts of the specification remain valid. Fig. 2 demonstrates that at least the specification modules `Messages`, `MessageSets`, `UChannel`, `ChannelList`, `ChannelMatrix`, `Processes`, and `ProcessList` could be re-used. To be exact, 35 of 45 theories remained unchanged during the transformation.

In the construction of proofs which need to be re-done, many of the old proofs or parts thereof could be re-used. Part of this re-use was performed automatically by VSE but more often proofs were used as guidelines such that we could concentrate on those aspects which involved fault actions. As might be expected, we identified the following two questions as important: (1) Does the occurrence of a fault directly violate the desired properties? (2) Does the occurrence of a fault affect the system state such that later actions of the system violate the properties? Under the crash fault assumption these questions can be answered easily since after a crash of an affected process this process does not execute further actions. For other fault assumptions we expect this to be more difficult.

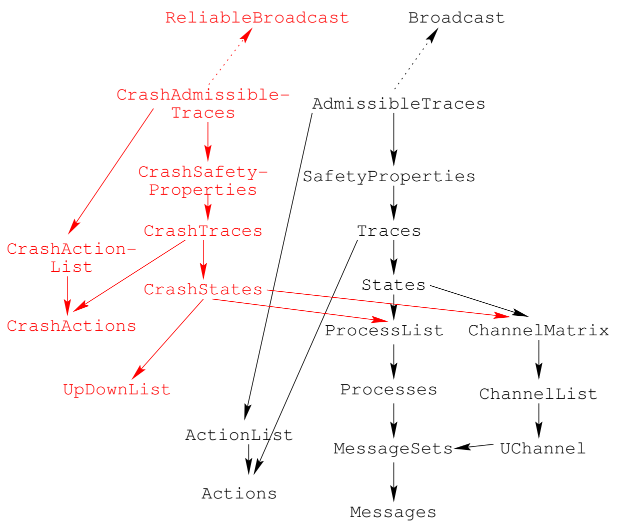

Figure 2: Changes made to the original graph for the fault-affected case.

Overall, our case study shows that the transformational approach in combination with our modular specification technique is a good basis for the mechanical verification of fault tolerance. The main argument for this is the potential for re-use of specifications as well as of proofs. In the case study this re-use has mostly be done by hand, however, further automation is desirable. Based on our experiences we have outlined how the transformation could be automated such that much of the specification can be re-used. Many of the interactions in the construction of proofs for the transformed specification were rather mechanical and could be done using the original proofs as guidelines. Further automation of the re-use appears to be achievable. However, the realization of this automation is a different issue which needs further investigation.

Results and Conclusions

Modeling the effects of faults as a transformation has opened the domain of verifying fault-tolerant systems to the realm of mechanical verification. As indicated above, fault assumptions can be translated into transformations and "applied" to given systems and their specifications rather mechanically. The transformed systems can then be reasoned about using standard tools and methodologies. It has often been argued that the difficulty in reasoning about fault-tolerant systems mainly lies in the additional system complexity caused by faults (mainly the state space explosion). However, a modular view of the system reveals that — when proving a fault-tolerant system correct — many parts of the proof do not actually involve reasoning about faults. These refer exactly to those parts of the system which are not affected by the fault assumption. Thus, proofs for the fault-free case can be re-used directly.

This case study has shown that these ideas can indeed help proving fault-tolerant systems correct in practice. Using a simple but important problem of fault-tolerant computing as an example, we observed that over two thirds of the specification modules remained unchanged under the transformation. Even in the modules which were affected by the transformation many parts remained unchanged. This saved an substantial amount of work in developing the specification for the fault-affected case. It also facilitated the verification process because the proofs for the unchanged parts of the specification remained valid. This allowed us to focus on those proofs which involve reasoning about faulty behaviour.

In our case study, the transformation has been performed by hand. However, we have outlined how the transformation can be automated. By generalizing these results, our approach appears to be feasible also for fault assumptions other than crash. There is much evidence that transformations can model *all* types of faulty system behavior (Gärtner 1999). Intuitively this is due to the fact that to an outside observer, faulty behavior is just another type of (programmable) system behavior. This means that the methods which we have studied here and also the core of our results can be applied to other fault assumptions and other systems. Thus, the example we have given here is only the starting point for a whole new set of investigations for formal methods research.

It could be argued that the benefit of modular reasoning decreases with the percentage of specification modules which are not affected by fault transformations. However, note that fault assumptions which merely affect process behavior (like even the very unfavorable *Byzantine* fault assumption) will not change more specification modules than our case study has made necessary. Thus, re-use of specification modules can be expected to be constant for this type of fault assumption. The differences between fault assumptions will become apparent only in the parts of the specification which are affected. However, we have argued that for each specification module the effects of different transformations are similar and have conjectured that they can be formalized in a uniform way.

It could also be argued that we have restricted our attention to proving safety properties. However, note that the traces we have considered are always finite by definition. Consequently, systems will always be trivially *live* in the sense of Alpern and Schneider (1985). In the future, we plan to study also liveness issues using the successor tool VSE-II (Hutter *et al.* 1998). Also, we are interested in automating the fault-tolerance transformations which we have performed by hand as well as in improving the automated re-use of proofs. This may contribute further to easing the fault-tolerance complexity burden.

Acknowledgments

We would like to thank Serge Autexier, Dieter Hutter, and Axel Schairer for valuable comments on an earlier version of this paper which helped us to improve the presentation.

References

Alpern, B., and Schneider, F. B. 1985. Defining liveness. *Information Processing Letters* 21:181–185.

Arora, A., and Kulkarni, S. S. 1998. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering* 24(1):63–78.

Chandy, K. M., and Misra, J. 1988. *Parallel Program Design: A Foundation*. Reading, Mass.: Addison-Wesley.

Gärtner, F. C. 1998. Specifications for fault tolerance: A comedy of failures. Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Darmstadt, Germany.

Gärtner, F. C. 1999. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science (J.UCS)* 5(10):668–692. Special Issue on Dependability Evaluation and Assessment.

Hadzilacos, V., and Toueg, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department.

Hutter, D.; Langenstein, B.; Sengler, C.; Siekmann, J. H.; Stephan, W.; and Wolpers, A. 1996. Verification support environment (VSE). *High Integrity Systems* 1(6):523–530.

Hutter, D.; Mantel, H.; Rock, G.; Stephan, W.; Wolpers, A.; Balser, M.; Reif, W.; Schellhorn, G.; and Stenzel, K. 1998. VSE: Controlling the complexity in formal software developments. In *Proceedings of the International Workshop on Applied Formal Methods – FM-Trends*, number 1641 in Lecture Notes in Computer Science, 351–358. Boppard, Germany: Springer-Verlag.

Kulkarni, S. S.; Rushby, J.; and Shankar, N. 1999. A case-study in component-based mechanical verification of fault-tolerant programs. In Arora, A., ed., *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, 33–40. Austin, TX, USA: IEEE Computer Society Press.

Lincoln, P., and Rushby, J. 1993. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis, C., ed., *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, 292–304. Elounda, Greece: Springer-Verlag.

Liu, Z., and Joseph, M. 1992. Transformation of programs for fault-tolerance. *Formal Aspects of Computing* 4(5):442–469.

Peled, D., and Joseph, M. 1994. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science* 128:99–125.

Qadeer, S., and Shankar, N. 1998. Verifying a self-stabilizing mutual exclusion algorithm. In Gries, D., and de Roever, W.-P., eds., *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, 424–443. Shelter Island, NY: Chapman & Hall.