Testing Implementations of Formally Verified Algorithms

Thomas Arts
IT University of Göteborg, Box 8718, 402 75
Göteborg, Sweden
thomas.arts@ituniv.se

Koen Claessen
John Hughes
Hans Svensson
Dept. of Computing Science
Chalmers University of Technology
Göteborg, Sweden
{koen,rjmh,hanssv}@cs.chalmers.se

ABSTRACT

Algorithms described in literature can often be used to solve practical, industrial problems. In safety-critical industrial settings, algorithms that have been formally verified should be even more attractive candidates for implementations. Nevertheless, we observe little transfer of algorithms from research papers into products. In this paper we describe a case study on the implementation of algorithms for the widely known and broadly studied problem of *leader election*. Despite thousands of articles on that topic, it still requires a lot of engineering to select the relevant articles, and get a correct algorithm implemented in an industrial setting. Modifications are necessary to meet all requirements. We propose adaptation and *testing* of formal properties as a realistic and cheap way to check the correctness of the modifications, since performing a formal proof seems unrealistic for industrial systems. We show how we use the properties stated in the articles to guide our tests.

1. INTRODUCTION

For the majority of algorithmic problems that arise in practical software development today, there exist books and papers describing possible solutions. However, for a software engineer, it is often a non-trivial task to first find the right source of information, and then adapt the described solution to the specific setting at hand. Many software errors are made because (1) the wrong algorithms were chosen, or (2) the right algorithms were adapted in the wrong way.

In this paper, we describe a case study where we develop a *fault-tolerant leader election* implementation. Leader election occurs frequently in distributed systems, when a number of processes quickly need to agree on who is the one deemed leader among them.

Getting distributed applications correctly implemented is a well-known problem, partly because of the non-deterministic behaviour. Even the design of simple looking distributed algorithms may give rise to the introduction of tricky faults, e.g. due to messages that arrive in a different order than expected. In addition to a collection of implementations of distributed algorithms in libraries, there is ample literature describing all kind of such algorithms. In many industrial cases, there is a need for the kind of algorithms described in literature.

The leader election algorithm on which we concentrate in this paper is needed as a component in telecommunication switching software. The leader election problem is a well-known and a well-studied problem. A literature search for algorithms leads to an extensive list of results. For example, Google Scholar [12] returns over 500 articles on the topic. With the use of CiteSeer [7] one can identify the most cited articles, which might be the starting point for reading. For safety-critical applications, such as in our case, a software engineer may even require a formally verified version of the demanded algorithm. Thus, an obvious possibility for a software engineer is searching the web for a paper describing exactly the algorithm with all properties necessary for a specific application.

The task is now to find an algorithm that fits all requirements prescribed by the application. If this algorithm has been formally verified, it seems to be a cheap way of using formal methods in the development process! However, it is almost never the case that such a perfect match is found. There are two main problems: (1) The assumptions on the semantics of the environment of the given algorithm are often incompatible with the assumptions on the development platform, e.g., certain assumptions seem either impossible to fulfill in a realistic setting, or they simply mismatch the actual setting. (2) Often extra features, such as exception handling cases, have to be added to the given algorithm, thus modifying the original algorithm and creating a different one.

Nevertheless, even if none of the available algorithms fits the requirements exactly, one can probably still benefit from the fact that formally verified versions exist. These versions may form a source of inspiration to develop a new version, where the properties proved for the algorithm indicate properties one would like to see satisfied for the modified version as well. A big disadvantage of such home-cooked algorithms is that, given the complexity of the problem, the risk for making a slip is relatively large, while at the same time the software developer is lured into thinking that the algorithm

Software Engineering Research and Practice 2005 Västerås, Sweden

at hand has been formally verified. Providing a new mathematical proof or even formally verifying the home-cooked algorithm is considered an unreasonably difficult task, even if the proofs of the original algorithm are available. Therefore a method to more easily check the implementation is needed.

We would like engineers to benefit from existing formal verification attempts to verify their algorithm. The simplest form of using the accumulated knowledge in the area, is to copy (and possibly adapt) the properties presented in different articles and to check in some way if they hold for the home-cooked algorithm. Therewith, a lot of effort in formulating meaningful and correct mathematical properties can be reused.

We argue that it is vital that some kind of connection is established between the actual implementation and the formal properties of such papers. It is important that (1) this connection is established in a relatively cheap way (so that it is doable by software engineers), and (2) this connection targets the actual code, not a high-level abstract description of the code (so that we avoid errors in implementation details).

We propose in this paper that *testing* is a practical way of doing this, fitting well into daily practice in industry. We propose two concrete testing methods that fit well with developing formally verified distributed applications. The proposed test methods are property based, or rather property oriented, which make them especially suited since we want to benefit from (adapted) existing properties.

In the rest of the paper we describe a case study, where a leader election algorithm is developed for an industrial telecommunication switch. Certain components of the system, such as a resource manager, have actually been formally verified [2]. The leader election component was deemed too complex to be formally verified from scratch. Therefore, a version was developed based upon a leader election algorithm that was developed and proved correct by Singh [16]. This new version was released as open source [22] and later adopted by industry.

In Sect. 2 we present some details about the programming language we use for our implementations and two test tools that we used. Detailed understanding of the language and tools used is not necessary for reading the paper; the method holds in a more general setting. However, certain details must be presented for reproducibility and deeper understanding of some issues. In Sect. 3 we present an overview of the methodology ussd in the case study. In Sect. 4 we describe the problems encountered when searching for a suitable algorithm. In Sect. 5 we focus on the challenges one meets when implementing a given algorithm in a context different from the intention of the creators of the algorithm. We describe two implementations of algorithms in more detail. The first is the previously mentioned algorithm based on Singh's algorithm [16], which turned out to contain an error. (Note that the error is introduced by the modifications and does not exist in the original algorithm) Unable to correct the error, we based our new implemention on a paper by Stoller [17].

In Sect. 6 we describe how we test properties of the implementations, which is how we found the error in the first implementation. The properties are inspired by properties presented in the articles describing the algorithms and by properties used in other formally verified versions of similar algorithms. We conclude in Sect. 7.

Contributions With the case study described in this article we have shown the difficulty in (and also the necessity of) adapting formally verified algorithms to a realistic setting. We demonstrate how such an implementation, even when the original algorithm is modified, can be tested by using properties from the formally verified algorithm. As such, we have proposed a method of reusing theoretical results in the area of formal verification in an industrial setting.

2. LANGUAGE AND TOOLS

In this section we briefly summarize important characteristics of the language and the test tools we use. A lot of these issues are not very language-specific and may easily be transferred to a different context.

2.1 Erlang

Erlang [1] is a functional programming language developed at Ericsson, and especially suited for implementing distributed control systems. It consists of a functional core together with additional constructs, such as process creation, message passing and code replacement. Most software written in Erlang is running in distributed environments and is highly concurrent and dynamic in nature. In Erlang terminology, a distributed system consists of *nodes*, e.g. workstations, which commumicate over a network. Each node can contain multiple light-weight *processes* that all have their own part of memory. Processes use *asynchronous message passing* which is implemented with a per-process mailbox, which is always ready to receive messages. A process can use a powerful pattern matching syntax to retrieve messages from its mailbox.

The language supports *process linking*, which means that a process A can obtain a *link* on a process B. If process B fails, the linked process A gets a message informing it about B's failure. Implemented on top of the linking construct, is the concept of a *monitor*, which provides reliable ways to monitor vital processes. By creating a hierarchy of supervising processes, failing processes may be restarted. By restarting a process, most faults of subsystems never lead to failure of the system; they at most either slow down the system, make the system more deterministic than intended, or reduce the functionality of the system.

Another feature of Erlang is that it supports *tracing* of events, i.e. every process can be instructed to implicitly send an event message to a collecting process before executing a specified function. In Erlang all possible computations and all message handling consists of calls to functions. Therefore we can define whatever detailed action that we like to be an event and to have that event generated when running our application. The only two shortcomings in this are that (1) it takes a few milliseconds to switch on event generation, such that certain events may have taken place before we start recording, and (2) the scheduling behaviour

of the application is affected. The timing behaviour is affected a little as well, although there is almost no overhead in tracing events. Timing, though, is not a critical part of the algorithms considered in this paper.

2.2 Traces, abstractions and analysis

The built-in trace functionality in Erlang is a very useful tool when testing an implementation. However, the raw trace data has a tendency to get very verbose, containing lots of events and also a lot of data per event. Manual inspection of traces is therefore often both tedious and time consuming, and alternative approaches has been proposed. In [5], one approach is presented where *abstraction functions* are applied to state based trace data, in order to remove unnecessary data and reduce the state space. The state space is reduced since different concrete states are reduced to the same abstract state when the abstraction function is applied. While collapsing different concrete states to the same abstract state, cyclic behaviors can be detected. The abstract state space is also visualized, something that gives a good intuition about the inner workings of an implementation.

This *abstract trace* approach is taken even further in [4], where we demonstrate the effectiveness of the method by testing the first leader election implementation [22] based on Singh's algorithm. We discovered two errors in this implementation, which led to the development of the second implementation based on Stoller's algorithm. In [4] we also introduce a small language for constructing abstraction functions, as well as checking LTL-properties for the abstract state space.

2.3 QuickCheck

QuickCheck is a property-based tool for random testing. Developers write properties in a restricted logic as a part of the program under test, then invoke QuickCheck to test the property in a large number of cases. Properties can check conditions using Erlang code, quantify over sets, and express preconditions. For example, the property

```
?FORALL(N,int(),
  ?FORALL(L,list(int()),
    ?IMPLIES(ordered(L),
      ordered(insert(N,L))))).
```

specifies that the result of inserting an integer into an ordered list is itself an ordered list (provided `insert` and `ordered` are defined in Erlang appropriately). Here, `FORALL` and `IMPLIES` are examples of logic operators provided by the QuickCheck library. QuickCheck generates test cases according to the stated types and preconditions, and checks that the condition is true in each one. QuickCheck thus allows the developer to focus on the *properties* that the code should satisfy, rather than on the selection of individual test cases. QuickCheck is implemented entirely as an Erlang API and associated macros, so properties can be compiled and run by the same tools as any other Erlang program.

QuickCheck tests concurrent programs by collecting a trace of events, which should have the properties the developer specifies. The events are defined by instrumenting the code under test with calls to the QuickCheck function `EVENT`. QuickCheck delays these calls randomly, thus in effect overriding the Erlang scheduler, and forcing a random schedule on the code under test. This can elicit faulty behaviour that would appear only very rarely with the normal scheduler, or perhaps only in a distributed setting.

A limitation of the version we used is that it only delays the *processes* under test, it does not delay the *messages* in transit. This will turn out to be important later.

3. METHODOLOGY

In this section we give a summary of the methodology used in the case study. The different tasks are further described in Sections 3-6.

1 - Identifying the problem

The first step is to realize that the solution to a given problem could be found by using a pre-existing (formally verified) algorithm. This step requires some intuition of what kind of algorithms exist, and in which situations one could benefit from an existing algorithm as opposed to implement the algorithm from scratch.

2 - Looking for candidate algorithms

The next phase is searching the literature and other resources for candidate algorithms. Before looking into the literature, it is good to (loosely) specify what the requirements are. Thereby many algorithms which are not suitable can be discarded at an early stage.

3 - Choosing an algorithm

Having identified some candidates, one should look in more detail to see which of the candidates that is best suits the requirements. This is a critical step, since it is often not obvious how well the algorithm fits into the implementation language semantics. A good choice here is an algorithm that fits well into the given requirements, and where the assumptions in the algorithm description match the given setting.

4 - Identify needed adaptations

It is not likely that one finds a perfectly matching algorithm for the given problem, since feature sets and environment assumptions can vary greatly. If adaptations are necessary, and we argue that that is almost always the case, these should be identified and thought through. Hopefully, with a good choice in step 3, the adaptations should not be too profound.

5 - Implementation

The implementation of the algorithm (and the adaptations) should be quite straightforward if one has managed to select a good algorithm for implementation.

6 - Finding and testing properties

Thorough testing of the implementation is crucial, since the fact that the underlying algorithm is correct often give the implementor false security that this also holds for the implementation. We propose that one uses property-based testing, and that it should be possible to re-use properties from

an existing verification of the algorithm. The biggest problem with a testing technology is to know when one has tested 'enough', and an adequate *coverage criteria* is necessary.

4. IDENTIFYING A SOLUTION

In order to successfully identify a possibly exisiting solution to a design problem, a software engineer is required to be able to describe the problem in such a way that it can be matched to solutions she is aware of. This in turn requires education and experience.

For example, in our case, the software architect was confronted with the following specific problem:

> Among four to sixteen workstations (depending on the configuration), one workstation needs to quickly be appointed to handle some administrative tasks for all workstations. From an efficiency point of view, there is one particular workstation that is preferred, but in case any other workstation starts earlier, it can perform the designated tasks as well. In case of a failure of the appointed workstation performing the administrative tasks, a different workstation has to be re-selected among the remaining ones.

In our specific case, the architect was aware of standard algorithms in the field and quickly saw the applicability of what in the literature is called a *leader election algorithm* or *leader election protocol*. The next step in the process seems to be to find a concrete description of such an algorithm. There are classical course books on distributed algorithms [15, 18]. They systematically describe several leader election algorithms in different contexts (synchronous ring, general synchronous networks, etc). A quick study of these does, however, not satisfy our specific needs.

There exist several algorithm repositories on the web (e.g., Intelligence United [20], the NIST Dictionary of Algorithms and Data Structures [19], and the Stony Brook Algorithm Repository [21]). Unfortunately, none of the searched repositories contained a leader election algorithm. Searching Google Scholar [12] for *"leader election" algorithm* and *"leader election" protocol* both give a more than 10,000 hits. The query *"leader election"* on its own results in almost 90,000 hits.

The situation we are dealing with is a configuration of a number of workstations connected via an ethernet network, running TCP/IP as communication medium. At any point in time, it is possible that one of the workstations stops working, in which case a new leader should be elected. Many articles solve the leader election problem for a different network topology than ours (mostly ring topologies). None of these is likely to contain a correct solution for our situation. Of course, one could implement a virtual ring on top of a fully connected network, but the ring structure is broken as soon as one of the workstations fails. Many articles also assume that none of the nodes ever fail, and are therefore discarded.

Adding additional keywords *"fault tolerant"* and *"fully connected"*, learned by the insight that reading a few articles give, reduces the number of hits to about 50. In theory it is doable to quickly judge all of these on their usefulness.

In the end, none of the articles we studied really fit. Some assumptions seem harmless, e.g., "*Processors are anonymous, they do not have identities*" [10], since these assumptions appear stronger than what actually need to be assured. Other assumptions may be simulated, like assuming that all processors work on the same memory, but that requires a change in the algorithm. Yet other assumptions are very hard to fulfill, e.g., "*For ease of presentation, we regard each processor as a CPU whose program is composed of atomic steps. An atomic step of a processor consists of an internal computation followed by a terminating action. The terminating actions are read, write and coin toss. ... Processor activity is managed by a scheduler. In any given configuration, the scheduler activates a single processor which executes a single atomic step. To ensure correctness of the protocols, we regard the scheduler as an adversary.*" [10]. This means that on the one hand, the scheduler is an adversary, meaning that every possible scheduling of events in considered, and on the other hand, there are atomic actions that the scheduler cannot interrupt. This works fine in certain (theoretical) settings, but in our case we deal with workstations that are running independently without a scheduler stopping them after they have sent a message.

Apart from knowing what to look for, a very good understanding of the semantics of the development platform turns out to be a necessity in the selection of candidate solutions. Therefore, the search task should be carried out by experienced programmers with awareness of the overall system requirements.

Well aware of the fact that relevant articles might have been overlooked, the first implementation was inspired by Singh's "Leader election in the presence of link failure" [16] and the second implementation was inspired by Stoller's "Leader election in asynchronous distributed systems" [17].

5. BRIDGING SEMANTICS

In this section we describe in detail some of the semantic assumptions posed by the algorithms of Singh and Stoller. We explain the difference between these assumptions and the situation we have at hand with our Erlang environment. In order to be able to use the algorithms in the demanded environment, some changes have to be made. We discuss these changes and show how easy they can lead to faults.

In general, there are many semantic assumptions that have to be carefully considered when implementing a distributed algorithm in a real environment. What does the environment require and what is offered by the algorithm? The following is an incomplete but illustrative list of such issues: Handling failure of the system during execution, using message passing or shared memory, connection failures during operation, having connections that break and are re-established, different network topologies, synchronous vs. asynchronous communication. It is also the case that a typical implementation environment have too many features to be adequately described in an article about an algorithm.

5.1 First algorithm

In Singh's algorithm [16] the participating processes behave as follows. When the protocol is started each process is given a list of all the participating processes; the position in this list is also the priority order for the processes. A process always plays one of the following four roles: *candidate*, *captured*, *surrendered* or *elected*. When a process is started it is always a candidate to become a leader. First the candidate tries to capture all the other processes, by broadcasting a 'capture'-message. If a candidate receives a 'capture'-message, it reacts based on the priority; it will silently accept messages from processes with lower priority, and reply by sending an 'accept'-message to processes with higher priority. After accepting, the candidate changes its role to being captured. A captured process will ignore 'capture'-messages and forward 'accept'-messages to the process that has captured it. Whenever a candidate has captured more than half of the participating processes, it will announce itself as the leader by broadcasting an 'elect'-message. If a process receives an 'elect'-message it will immediately surrender.

The following assumptions about the problem are stated: "*In the paper, we study the problem of leader election in the presence of link failures. In the problem of leader election, there are N processors in the network, each having a unique identity. Initially all nodes are passive. An arbitrary subset of nodes, called the candidates, wake up spontaneously and start the protocol. On the termination of the protocol, exactly one node announces itself the leader. We consider the problem of electing a leader in an asynchronous complete network. In a complete network, each pair of nodes is connected by a bidirectional link and we assume that a node is initially unaware of the identity of any other node. By electing a leader, one can execute centralized protocols in a distributed system.*" This summarizes exactly what we need, apart from the fact that we want a re-election if the leader fails. The stated algorithm simply halts when a leader is elected. What is surprising is the remark "*Although complete networks are not practical, they have been widely used for theoretical studies. They provide a lower bound for more practical networks (which have less connectivity)*", since we consider Ethernet networks with TCP/IP as rather practical. Careful reading reveals a misunderstanding from the implementors side; complete networks really means a cable from each node to each other node. The algorithm in the paper had to be adapted in several ways to fit our setting. Firstly, we want a re-election to occur whenever the current leader dies. In the modified implementation, all surrendered processes restart the protocol as soon as they detect failure of the leader. Secondly, for optimization purposes, we implement detection of dead nodes. A process is announced leader if it captures at least half the nodes that are alive. Thirdly, the bidirectional link is implemented by an Ethernet network, each workstation can communicate with each other workstation. Thus, we believe to meet the complete network requirement. In that setting, nodes do know each other's identity from the beginning (e.g. IP number), but that can hardly be a hindrance.

Singh's algorithm described assumes that failures occur in the communication links, not in the nodes. The fault-tolerance of the algorithm is expressed "*given N nodes,* $N^2/4 - N/2$ *links may fail*". In our setting, the algorithm is used in a situation where all nodes can communicate with all other nodes in a network that contains no failures. After realizing this, we should probably have discarded the algorithm. However, the interpretation was made that if a node dies, $N - 1$ links fail (*i.e*, all links from that node to any other node). Given 6 nodes or more, one node failure causes at most the allowed number of link failures and such failure is therefore assumed to be recoverable.

The modified algorithm was implemented and manually tested by observing screen outputs notifying which node became the leader and thereafter manually killing one node randomly. These tests showed that there is a leader elected after a few milliseconds and killing the leader results automatically in a new leader election and a stable situation in which another leader is selected. Therefore, the algorithm was released.

A later, more intensive analysis of the implementation, identified two errors further described in [4]. Both were the result of incorrect modifications to the original algorithm. The first error made it possible for two leaders to exist simultaneously and was caused by the addition of the possibility to elect a leader with less than half of the nodes being alive. The second error was that a deadlock situation occured after the election process was restarted due to the failure of the leader.

In Singh's algorithm these problems are absent, since restarting the election process is not considered, and the rules for electing a leader are more restrictive.

5.2 Second algorithm

Directly after implementing Singh's algorithm (with home-cooked modifications) a new search for leader election algorithms was performed in order to document the source of Singh's algorithm. Coincidentally, Stoller's algorithm was discovered [17] and reviewed to be a good alternative to Singh's algorithm. After identifying the errors in the modified version of Singh's algorithm, it was decided to implement Stoller's algorithm, instead of trying to further modify the algorithm with the risk of introducing different errors.

In Stoller's algorithm the participating processes behave as follows. When a process is started, it first checks whether any process with higher priority is alive. If there is, it waits for one of these processes to become leader. If not, the process itself decides to try to become leader. It then checks that all other processes of lower priority either are aware of its existence, or are dead. If so, it announces itself as leader.

Stoller states the following assumptions: "*The system comprises a fixed set of nodes and a communication network. Nodes may crash and recover; other types of failure are assumed not to occur.*" This is expressed in very general terms, but seems to meet our requirements nicely. The assumptions continue with: "*Each node has access to a small amount of stable storage (i.e., storage whose contents survives crashes). Nodes communicate by sending messages. Communication is FIFO. For synchronous systems, we assume also that communication is reliable.*" The stable storage part can be a bit troublesome in embedded systems, but we may use a local disk.

The communication statement is very vague. We interpret synchronous systems as systems that communicate with synchronous message passing. It is far from obvious what is meant by FIFO communication. Here we assume that every process has a message queue which is FIFO. We also assume that the synchronous communication is synchronization between sending process and message queue of the receiving process. This is concluded from studying the algorithm and excluding other possible interpretations. Luckily, the assumption fits the Erlang semantics well. In our setting, the communication is reliable so these requirements could be met.

There are actually two different algorithms described in the paper, one algorithm with synchronous message passing and one with asynchronous message passing. What is perhaps most surprising, and at the same time shows how difficult it is to select a good candidate algorithm for implementation, is that we chose the synchronous algorithm! Erlang is a *functional language with asynchronous communication*, and therefore it would be reasonable to choose the asynchronous algorithm. A more careful reading of the article reveals that the difference between the synchronous and the asynchronous algorithm lies mostly in how the failure detection works (how node failures are detected and reported), and here the Erlang *monitor* works in the same way as the failure detection with synchronous message passing. It is also the case that the asynchronous message passing in Erlang could be regarded as rather synchronous in the way the underlying mailboxes are implemented. Messages more or less synchronize with the mailbox of the receiving party. This shows how important it is to have a thorough understanding of the inner workings of one's implementation language.

Contrary to the demand that the node that is ahead in the startup process should become the leader, it is stated that "*The basic idea in the Bully Algorithm is that the operational node with the highest priority becomes the leader.*" This is actually not what we want, if a node with lower priority became the leader, this node should remain the leader even if a node with higher priority is (re-)started. So yet again we have the situation where we need to make changes to the algorithm as to match the specific requirements of the intended usage. Therefore, one may wonder whether we can trust the modified Stoller algorithm.

6. TESTING IMPLEMENTATIONS OF ALGORITHMS

In the previous section, we show that few algorithms directly fit the requirements of a particular application. Even in the cases where they do, we often use an underlying programming language which differs from the pseudo-code in which the algorithm is written. Therefore, it is important to create a strong connection between the model of the algorithm presented in the paper and the code of the actual implementation.

We argue that thorough testing is the only reasonable way to establish this connection. The immediate questions then are: (1) What to test? and (2) How to test it?

What to test

Leader election is a well-known and clearly defined problem, which means that the requirements are also well defined: (1) Eventually, a leader should be elected, and (2) At most one of the participants is considered the leader. Variations of these properties are also stated in the articles. For a leader election algorithm, the properties are fairly obvious. In general, the properties are not always obvious, and most articles about algorithms state properties that are supposed to hold. In the property-based testing, we try to adapt these properties into testable properties.

However, there are significant differences between testing and formally proving properties. One issue is that *eventuality properties*, i.e. properties like *'Eventually a leader is elected'*, must in a testing setting be bound by some maximum time or maximum number of events. In our case, this is easily solved, because if no leader is elected within a second (an infinity in telecommunication applications), then the application is not functioning correctly. But in other situations, it is often far from obvious what a correct value of such a maximum time is.

Another issue is that the properties stated in the articles are often global properties, they state something for all processes at the same time. For example, in the second implementation based on Stoller's article [17] one could only observe that a leader is elected and accepted by all other nodes by observing that for each (active) node a certain local variable has the value `Norm` and a second local variable stores the identity of the elected leader. Unfortunately, in a distributed system, global properties are very hard to observe. By the time you have collected all state information, a few nodes may have already changed their local values, such that you never know whether the actual global state had that value at one time instance. There are many well-known solutions to the distributed global state detection problem, for example a *snapshot algorithm* [6], global clocks or logical clocks [14]. The result of such algorithms is not necessarily a true global state, but rather a global state that could have been. Using a snapshot algorithm is however not an ideal solution in our setting, since these algorithms are superimposed on the underlying computations and will therefore affect the system under test.

The problem of verifying global properties when you only have access to local state information is not new. In a formal verification approach, the same phenomenon occurs. The events that are visible in the state space typically give local information about the state of one or two processes. Properties that depend on information from a collection of local states are verifiable if one has access to all possible runs of the program, but these properties are rather hard to formulate. Verifiers often add an observable action, similar to printing 'I am the leader' to make it easy to formulate the property (c.f. "Formal verification of a leader election protocol in process algebra" [11]). In that way one verifies that one process indeed gets the leader role, but not that all other processes are recognizing this leader or are aware of this leader. Additional properties need to ensure that. We apply the same approach in our testing; by observing a certain subset of the local state variables, we are able to distinguish whether a process considers itself the leader.

Apart from the two properties discussed earlier, the articles we studied have a lot of detailed properties that are specific for the chosen algorithm. One example is (from Singh's article [16]): *If there exists at least one node in phase l of iteration k then at least one node will enter either phase $l+1$ of iteration k or the first phase of iteration $k+1$.* Testing these more specialized properties as well, as opposed to only testing top-level properties, leads to a more fine-grained testing scheme, which greatly improves the effectiveness of random testing [13]. A disadvantage with these algorithm specific properties is that we can (in general) only use specific properties from the article we are implementing and not from other papers presenting similar algorithms. Another disatvantage could be that theses more specific properties do not (and sometimes *should not*) hold in the modified version of the algorithm, thus some extra thought is needed. In the QuickCheck testing described in Sect. 6.2, we also tested some specialized properties, most of those were invariants of the type: *the number of alive nodes is not greater than the number of participating nodes.*

How to test it

Given that we know what properties we should test for, how can we now test the implementation? There are two issues that need to be addressed here. (1) How do we generate stimuli to the system under test, and (2) How do we check that the desired properties hold for the runs of the system we executed?

The "input" to the leader election algorithm is rather simple; it basically consists of a sequence of events, where an event either indicates that a particular node has died, or that a node has revived again. However, testing a distributed system by varying just those events is far from sufficient. The scheduler in Erlang is deterministic and therefore running the software in a certain configuration results in a relatively low number of covered execution paths. Therefore, when performing stimuli, we also would like to influence the behaviour of the underlying run-time system. In particular, while testing, we can arbitrarily vary the behaviour of the scheduler, and we can arbitrarily delay messages sent from one process to another. In this way, more execution paths can be explored.

Below we discuss two different testing approaches, based on two different concrete testing tools that exist for Erlang. There are mainly two things that differ between the two approaches, namely (1) how we control the scheduler and (2) how properties are checked. The first difference affects the execution paths explored in the tests, the second difference affects what kind of properties we can express, and thus what kind of bugs we can detect.

6.1 Testing with trace collection

In the first tool, described in [4], we stimulate the system by arbitrarily killing and reviving nodes, and by arbitrarily delaying messages sent between processes. Moreover, we use recording of event traces in order to observe faults that are not manifested in failures. We were able to use this technique to identify certain errors in the implementation of the first algorithm. The first error lead to the possibility to have two leaders existing at the same time. The second error caused falsification of the eventuality property "Eventually a leader will be detected"; we could detect a situation where the system would end up in a deadlocked situation without a leader.

Using the same trace recording techniques for the implementation of the second algorithm did not reveal any fault. This gave us a certain confidence, but we want to increase our confidence by an additional testing technique. Our concern, namely, is whether we have tested enough of the possible runs of the system.

One big advantage with the trace collection testing technique is that it is detached from the test case generation technique. In this case study we have used random stimuli. However, any test case could have been used since the trace collection is built into the run-time system and not a specific part of the testing.

6.2 Testing with QuickCheck

In the second tool, QuickCheck, we stimulate the system again by arbitrarily killing and reviving nodes, but now we randomly steer the possible execution paths of the processes by directly influencing the Erlang scheduler. For that reason, QuickCheck has been extended with the option to create randomly scheduled executions.

We used QuickCheck to test both the first and the second implementation of the algorithm. Here, we could reuse the top-level properties for both implementations. In Figure 1 a fragment of the property file is presented, showing the property that there are not two leaders present at the same time. The `FORALL` construct together with the `longtrace` will randomly generate traces, which are passed to a checking function `not_two_leaders` which checks the property for the trace. In general, code like this is written by the tester/implementer, hopefully with a lot of guidence from the article. We wrote about 10 such properties during the tests, most of those applied to both implementations (except for a few specific invariants). Note, here `election` is the record containing the state information needed to decide whether a node is the leader or not.

Testing the first implementation

We tested the first implementation using QuickCheck, to see if we could reveal the same errors as with the trace recording technique [4]. To this end, we constructed one (parametrized) function to start the application, and further instrumented the main loop of the implementation with the QuickCheck `EVENT`-construct, which automatically transfers scheduling control to QuickCheck and generates traces of the execution. By running QuickCheck with the property: "Two leaders should not be present simultaneously", we were able to observe the same fault as with the trace recording technique.

However, formulating and testing for the second fault, the deadlock situation, proved to be much harder. As discussed earlier, the way to detect a deadlock in testing is via a timeout. However, timeouts are not always necessarily a bad thing, since a timeout only says that there has not been any activity in the system during a given time period. So, it is necessary to specify which timeouts indicate deadlocks, and which ones do not, something that is hard to do in the

```
leaderprop() ->
  ?FORALL(Trace,?longtrace(3,400,traceit:start()),
          not_two_leaders(Trace,sets:new()));

% Not two simultaneous leaders
not_two_leaders([],_) ->
  true;
not_two_leaders([{event,Pid,E}|Es],Leaders) ->
  case Pid == E#election.leader of
    true ->
      NewLeaders = sets:add_element(Pid,Leaders);
    false ->
      NewLeaders = Leaders
  end,
  (sets:size(NewLeaders) < 2) and
  not_two_leaders(Es,NewLeaders);
not_two_leaders([{exit,Pid,_}|Es],Leaders) ->
  not_two_leaders(Es,sets:del_element(Pid,
                                      Leaders));
not_two_leaders([_|Es],Leaders) ->
  not_two_leaders(Es,Leaders).
```

Figure 1: Fragment of property-file

current QuickCheck version. There is also a second reason to why we where unable to detect the deadlock error using QuickCheck. The deadlock depended very much on specific message timing circumstances; the fault occurs only if a certain message arrives much earlier than another message. Since we do not influence the speed of delivery of a message in this approach, the fault is very unlikely to appear.

Testing the second implementation

Using QuickCheck to test the second implementation, we could not produce any trace where the properties were violated. Nevertheless, we could observe some failures, namely that a leader election process crashed unexpectedly from time to time. This did not lead to any faulty behaviour, but it indicated that something was wrong.

Closer analysis revealed a very tricky error, which would have been extremely unlikely to be found without control of the scheduling. The problematic situation occurrs whenever a process A is about to contact another process B. To do this in a controlled way, process A first request a monitor (a link mechanism such that a notification is received whenever the other process fails) on process B before sending the message. What can occur now is that process B is down when process A requests the monitor, but alive just some time later when process A sends the message. In this case, process A receive both a failure-notification and a message reply. This situation was overlooked in the implementation and led to a crash. Luckily, the error could be easily corrected.

In this example we can see how important it is to have control of the scheduling, since this situation occurred frequently (like once every 150 tests) while testing with QuickCheck but could not at all be observed when we tested the implementation with the trace recording technique.

In this respect, our erroneous implementation followed the algorithm in the paper quite closely. Does this mean that the same error is present in the article? That is a rather hard question to answer, since Stoller's article [17] is not very specific about the semantic assumptions made regarding link requests between processes. This yet again shows the difficulties of bridging the semantics from the article, where underlying assumptions often hide important and problematic issues, to the implementation language.

	Killed nodes	in election	as leader	surrendered
QuickCheck	1601	23.6%	6.4%	70.0%
Trace rec.	101	4.0%	10.9%	85.1%

Table 1: Coverage results

6.3 Coverage

When working with test methods, the issue of *coverage* is central. Coverage should provide a measure of how exhaustively one has exercised the system, and is therefore important when evaluating the results of testing. Though it is very rare that a coverage measure can tell when we have tested enough, rather the coverage measure will warn of potential situations when we have *not* tested enough.

The simplest form of coverage is *code coverage*, which measures whether (or how many times) a certain line of code has been executed. This measure is rather useless here, since we cover 100% of the relevant code. Instead it is the complicated interactions of several different instances of the implementation that should be studied. Therefore a better coverage measure would be how much of the statespace (for a combination of several processes) we have exercised.

In [4] we discuss some coverage mesaures for the abstract state space in the trace recording technique. Those measures are hard to compare with the QuickCheck tests. Instead we choose to look at how many nodes were killed, and at what stage in the election process the nodes were killed. Killing nodes in different stages means exercising different parts of the statespace, thus only killing nodes at a certain stage in the election process is not a good idea. We should also observe that it is possible to affect the size of the testcases, via the `sized` functionality in QuickCheck, this is necessary since the default sized traces are too short. To overcome this, we created our own `longtrace` macro which takes a size factor as parameter.

In Tab. 1 we can see the coverage result (labeled "QuickCheck") for a QuickCheck run with 5 nodes and the longtrace parameter 10 and as a comparsion results for a run with the trace recording technique (labeled "Trace rec"). The first column shows the total number of killed nodes, second column the percentage of nodes killed during an election, third column the percentage of nodes killed when elected as leader, and fourth column the percentage of nodes killed when being surrendered to a leader. In the coverage results we can note a difference between the two techniques,

since we do not influence the scheduler in the trace recording thechnique it is quite rare that we manage to kill a node in the middle of the election process (merely 4% of the kills) compared to the QuickCheck approach where this happens a lot more frequently (almost 25% of the kills).

7. DISCUSSION AND CONCLUSIONS

The problem that we address in this paper is that there in general exists a big gap between the idealized world where algorithms are presented and proved correct, and the real world where the algorithm should work in a concrete environment. We do not propose to change this situation, as we believe that there certainly is value in solving problems in an idealized setting, and that perfect solutions to realistic problems are often out of reach. Instead, we try to present a recipe and guidelines for how to approach a problem in this setting.

It is often impossible to find an algorithm that exactly matches the specifications of the situation at hand. However, as our case study shows, picking the right solution to the problem at hand is essential. So, selecting the right article to use seems to require both education and experience from the software developer.

What we argue is that, when implementing an algorithm from an article, a software engineer cannot only benefit from the description of the algorithm itself, but also from the other information, such as theoretical properties the algorithm should have. We propose a concrete method (and two concrete tools) to cheaply use this extra information, namely property-based random testing.

The reuse of this information becomes even more important when the algorithms are adapted to new settings and augmented with new features, which is almost always the case in practice.

We identify testing as the main tool for checking whether the implementation really behaves as intended, and we demonstrate how one can (re-)use properties that are verified in the articles. We also discuss what one can expect to find in an article about an algorithm, and identify several potential problems when the properties have to be adapted to the implementation environment.

Model checking [9] could be regarded an alternative to testing. The model used should be closer to an implementation than a pen-and-paper proof. There are even tools to create the model from the implementation [2, 3]. Such a model is (and should be) an abstraction, leaving out some details that can lead to a defect. This is necessary in order to deal with the state space size of realistic examples. For example, we have a fault-tolerant system, which means that it is possible for any participant to fail at any time. This means that the resulting state space grows very quickly in the number of participating processes, limiting the applicability of the technique. Moreover, the mentioned tools for model checking Erlang programs use an Erlang semantics which only deals correctly with a single run-time system [8]. Therefore model checking would limit us to systems with a small number of participants, and still there is the probability that errors slip through due to the underspecified semantics. It is certainly a fact that the real world is too complex to fully model, and there exist no tool that can handle all the features provided by a typical implementation environment.

We have used two different techniques for testing the leader election implementation. In many ways the testing techniques are very similar; both use (directed) random testing, and both use traces. What especially differs between the two methods are (1) how we control the scheduler and (2) how properties are checked. The first difference affects the execution paths explored in the tests, the second difference affects what properties we can express (and so what bugs we can find). It is important to note that the combination of these techniques are rather arbitrarily chosen; one could certainly imagine an approach with abstraction functions and an influenced scheduler or the other way around. Another important fact is that although we have limited ourselves to random testing here, the trace collection technique is not in any way limited to random testing, on the contrary, any type of test case generation can be used.

The concrete test results for the two implementations of the leader election protocol also show that the two test methods are complementary. We believe this is because most non-obvious errors manifest themselves in rare schedulings, and influencing the run-time behaviour in certain ways affects the likelihood of certain executing paths.

Future work The most important issue is whether the proposed methodology can be generalised to other algorithms. To fully answer this question, many more cases are needed. However the results so far are positive, and the overall methodology should be applicable in many cases although the details may vary significantly.

8. REFERENCES

[1] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[2] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 2004. to appear.

[3] T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating Erlang to mCRL. In *Fourth International Conference on Application of Concurrency to System Design*, Hamilton (Ontario), Canada, June 2004. IEEE computer society.

[4] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume Vol. 3395, pages 140 – 154, Feb 2005.

[5] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[7] Citeseer. http://citeseer.ist.psu.edu/.

[8] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proceedings of the 4th ACM SIGPLAN Erlang Workshop*, pages 78 – 87, 2005.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[10] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[11] L-Å. Fredlund, J.F. Groote, and H. Korver. Formal verification of a leader elction protocol in process algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.

[12] Google scholar. http://scholar.google.com/.

[13] R. Hamlet. Random testing. In J.Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[16] G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

[17] S.D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[18] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[19] Dictionary of algorithms and data structures. http://www.nist.gov/dads/.

[20] Intelligence united. http://www.intelligenceunited.com/.

[21] The stony brook algorithm repository. http://www.cs.sunysb.edu/ algorith/.

[22] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.