Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang

Thomas Arts[1], Koen Claessen[2], and Hans Svensson[2]

[1] IT University in Göteborg, Box 8718, 402 75 Göteborg, Sweden,
`thomas.arts@ituniv.se`

[2] Chalmers University of Technology, Göteborg, Sweden,
{`koen,hanssv`}`@cs.chalmers.se`

Abstract. We present a semi-formal analysis method for fault-tolerant distributed algorithms written in the distributed functional programming language Erlang. In this setting, standard model checking techniques are often too expensive or too limiting, whereas testing techniques often do not cover enough of the state space.

Our idea is to first run instances of the algorithm on generated stimuli, thereby creating *traces* of events and states. Then, using an abstraction function specified by the user, our tool generates from these traces an abstract state transition diagram of the system, which can be nicely visualized and thus greatly helps in debugging the system. Lastly, formal requirements of the system specified in temporal logic can be checked automatically to hold for the generated abstract state transition diagram. Because the state transition diagram is abstract, we know that the checked requirements hold for a lot more traces than just the traces we actually ran.

We have applied our method to a commonly used open-source fault-tolerant leader election algorithm, and discovered two serious bugs. We have also implemented a new algorithm that does not have these bugs.

1 Introduction

The company Ericsson has developed a telecommunication switch called the AXD 301 [7]. The control software of this switch is written in the distributed functional programming language Erlang [2]. A major challenge in the development of the switching software is to get the almost one million lines of code tested in the relatively short time between releases of the product. A typical time consuming and difficult activity is testing fault-tolerance properties. The particular fault-tolerance we investigate here is the effect of taking down parts of a switch (because of maintenance or hardware problems) and restart them later in time.

We report on our case study to take away part of the testing load by analyzing a critical part of the code by semi-formal methods. The part we looked at is a *leader election* protocol. In the AXD 301, a module of about 2000 lines of code implements both a leader election protocol and a resource manager. In order to be able to deal with the complexity of this module, Ericsson's engineers rewrote the module in two parts, separating the resource manager and the leader election protocol. The simplified resource manager has been formally verified in earlier work by using a model checking approach [3]. The slightly generalized and cleaned up leader election protocol contains about 800 lines of code and is available as open source [21].

The leader election problem is a well-known and extensively studied problem. The objective of the protocol is for the processes among themselves to establish a designated process, called the *leader*. Leader election protocols have been designed for many different settings. In our case, we are interested in a solution that is fault-tolerant. Fault-tolerance is based on communication links breaking or on processes that may die or revive again at any point in time. If the currently elected leader dies or is disconnected, the surviving processes need to elect a new leader amongst them. However, during the election process, other processes may cease to work. We consider asynchronous communication with buffered messages. Among the many articles published on the leader election protocol [9, 6, 16, 17, 1], we know of only few that address all these problems. It was actually hard to find a paper describing exactly the setting that Erlang uses: asynchronous message passing, reliable communication channels between every pair of processes, possible failure and/or revival of a process at any point in time and a reliable notification mechanism of when processes die.

The algorithm used in the leader election protocol implementation we analyzed was an adaptation of a previously published algorithm [16]. There is a fixed set of processes that can die arbitrarily, and they have to negotiate a leader among them. The first process that comes up has priority to become leader, in order to have a selected leader as soon as possible. Only when the current leader dies, a new leader should be elected.

There are two basic properties that the leader election implementation needs to obey: (1) **Safety** — it is never the case that there are two or more leaders at the same time; (2) **Liveness** — in a stable situation (i.e. processes stop dying for a while), a leader will eventually be elected.

We have considered using model checking techniques to formally verify these properties. However, we found that dealing with the fault-tolerance lead to state-space explosion in the used model checkers, which severely limited the number of processes we could deal with. More informal methods based on testing seemed to be necessary.

The Erlang runtime system has built-in support for generating *traces* of the events occurring during execution. With simple means, one can specify what one considers an event (sending a message, receiving a message, a process dying, a function call, etc.). Tracing can be switched on and off on demand. Studying the traces reveals not only that an error occurred, but can also demonstrate the chain of events that lead to the error.

We have developed a methodology for semi-formal analysis of such traces of distributed systems (c.f. [5]). The idea is to first produce traces of the system by generating stimuli, as in testing. Then, we build *abstractions* of the traces with the help of an *abstraction function* specified by the user. An abstraction function basically maps data structures in the events and states to different (simpler) representations. An abstraction reduces the number of states, by mapping the actual concrete states onto a set of abstract states. This also allows us to detect cyclic behaviour in the trace, since different concrete states can be mapped to the same abstract state. The accompanying abstract state transition diagram concisely indicates the different abstract states visited during an execution, together with the messages sent and received during the transitions. A path through such a state diagram represents a trace, but it is not necessarily the case that the trace is a possible trace for the system since the diagram is really a diagram for an abstracted model of the system.

We propose to generate traces of simple instances of the software first, such as a reduced number of processes or executing only one possible scenario. For these traces it is easy to define abstraction functions. The same functions can be used for more complicated instances of the software.

The generated abstractions are used in two ways: (1) They increase understanding of the system, and can help to more easily spot the causes of bugs (as explained in Section 2.4); (2) We can formally verify properties of the abstraction, thus ensuring that the desired properties not only hold for the generated traces of the system, but also of all other paths through the abstract state diagram (as explained in Section 2.7).

When we applied our methodology on the implementation of the leader election algorithm, we discovered two serious bugs. Failing to correct the bugs in an efficient way, we also tried to implement a different algorithm for leader election. This implementation is based on [17] and was tested with the methodology described in this paper without finding any errors.

2 Methodology

In this section, we describe our methodology in more detail. We do this by concretely following the analysis of a leader election algorithm in chronological order. We start by describing the original algorithm, how we generate stimuli to obtain system traces, and how we use abstractions to find bugs. Then, we describe our own implementation of leader election, where all of our analyses failed to find any errors, and discuss coverage issues related to our method.

2.1 Fault-Tolerant Leader Election Version 1

The Erlang code for the algorithm we started with is publicly available on the web [21]. However, for simplification purposes we actually analyze a cut-down version of this code here. All code we used in this case study is available on the web [18].

The implementation is loosely based on an algorithm described in [16], which is an algorithm designed for fault-tolerance where faults occur due to failing communication links. Therefore it was necessary to adopt the algorithm to the Erlang setting, where faults occur mainly due to processes dying.

The participating processes behaves as follows. When the protocol is started each process is given a list of all the participation 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. The first thing it does is to try to capture all the other processes, by broadcasting a 'capture'-message. If another candidate-process receives a 'capture'-message, the receiving process will take action based on the priority; it will ignore messages from processes with lower priority, and accept messages from processes with higher priority by sending an 'accept'-message. After accepting, the process 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 living processes, it will announce itself as the leader by broadcasting an 'elect'-message. If a process receives an 'elect'-message it will immediately surrender.

Whenever the leader dies (processes discover this since the Erlang runtime system will send a 'DOWN' message to all interested processes), a new election round is started. Whenever a process revives, this process will be notified of the (possible) presence of a leader via an 'elect'-message as a reply from the leader to the 'capture'-message sent when the process revived.

When we got the algorithm, it was said that it would always eventually choose a leader if more than 50% of the processes are alive and if the system is stable for long enough. (It is though possible that a leader is elected with fewer processes alive.) The algorithm is not supposed to elect a new leader unless the leader dies.

2.2 Generating Stimuli and Tracing

The Erlang Runtime System (ERTS) has built in functionality for tracing running processes. The tracing can be switched on or off at any given time, without interfering with the execution. It is possible to trace sent and received messages, function calls, process related events, process scheduling and garbage collection. In a distributed environment there exists functionality for redirecting trace messages to a central collection process, in order to collect all trace data into the same log file.

Stimuli for leader election implementation In order to generate traces of the leader election protocol, a set of nodes is started, and a leader election process is started on each node. The stimuli for a leader election system is killing and reviving processes. Therefore, for the purposes of generating stimuli, each node also runs a controller process that can kill and restart the leader election process. A simulation process then randomly selects which process to kill/revive by sending messages to the control processes. How many processes can be dead at the same time is configured in the simulator process.

In order to further test the robustness of the leader election protocol, we implemented a variant where we also delay messages between nodes in a random way. The idea is that this simulates slow and/or overloaded connections. Since the leader election implementation is supposed to be fault-tolerant, it should be capable of working with slow/overloaded connections. Note that this is not tested in a standard setting where one runs all nodes on the same hardware, since communication delays will be rather static in such a setting.

Tracing the implementation We first collected trace data for the simplified version of the leader election protocol, without using message delays. When running a leader election system with three processes, everything worked fine, but when running with five processes something was obviously wrong; there were two processes simultaneously announcing themselves as leader! In the search for this

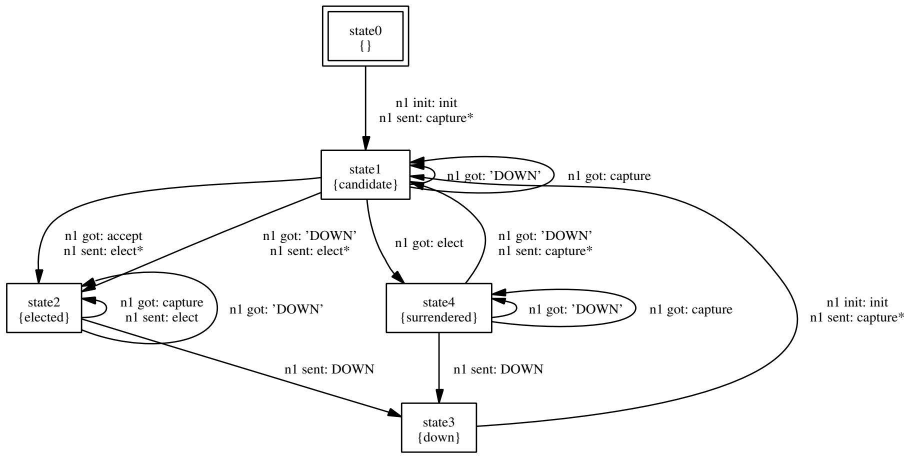

Fig. 1. Abstract trace for one process in a three node setting

error, we focused on the trace data for one of the nodes which was elected as leader. The raw trace data contained roughly 120 states and 200 message events, a bit too many for easy overview. The problem here is that it is easy to spot where in the trace the fault happened (two leaders are elected), but not where in the trace the event happened that triggered the fault (the first illegal state).

2.3 Abstractions

It is clear that in order to understand larger traces of systems, one has to reduce the information in the trace to a relevant subset of all information. One way of doing this is by using an *abstraction* (c.f. [5]). Abstractions are made by applying an *abstraction function* that converts each concrete state in the trace to an abstract state, which contains less information. Several different concrete states from the trace might actually be mapped to the same abstract state. Thus, we can redisplay the trace by means of a state transition diagram, where each abstract state occurs only once, and transitions occur between two abstract states if there exists a transition in the trace between two corresponding concrete states. However, by doing this we also lose some context, for example a state visited exactly N times in the actual trace is represented by a loop, and thus potentially infinitely visits, in the abstract trace. Moreover, we can also make the sent and received messages more abstract by applying a message abstraction function.

An example of an abstract state transition diagram of the leader election protocol is displayed in Fig. 1. Our tool automatically generates this diagram, given an abstraction function specified by the user. The original trace used for this diagram is a trace of a correct execution with three leader election processes. Here, the abstraction function on states is tracking the state of only one process, abstracting away the states of the other processes. Moreover, it has removed all other information in the concrete state, but for the role a process is playing. This diagram shows that with help of an abstraction, one can get an understanding for the basic parts of the algorithm, since it is easy to follow how the process moves between the different roles. We call the transition diagram generated from a trace and an abstraction function an *abstract trace*.

Common abstraction function building blocks We have implemented a library of common abstraction function building blocks. Commonly used functions are: removing parts of state data, replacing a list by its length, focusing on the state of one process, merging states of two processes into one state, etc. This library makes it easy to quickly define new abstraction functions.

2.4 Abstractions for Bug Finding

The idea is now to find an abstraction function which clearly helps us to establish where in the code the bug is located. It is hard to give a general approach on how to come up with an appropriate abstraction. Most of the time, the programmer has some sort of intuition about what parts of the states and which events influence a particular bug.

In the case of our bug, we have applied the following principles. Some of the state data, such as the list of participating nodes, is the same in all states, and such data can often be abstracted from. In the state data there are also two lists, containing the references to monitored nodes and the nodes which are down. The contents of these lists are not really useful, it is enough to know how many elements there are in the lists. So, we abstract away from these lists by remembering their length, but not their content. Concerning the events, most of the message data can be abstracted away, only keeping the type of a message.

The above abstraction reduces the state space from 120 to 23 states, which is small enough to overview. It is now possible to spot the bug by just looking at the abstract trace (Fig. 2). The state where the process is elected as leader is dark shaded in the figure (it is in the lower left half). This state is part of a long almost non-forking path, and it is likely that the first illegal state is to find at the top of this path. This is indeed the case and if we zoom in on two lightly shaded states in the upper left part of Fig. 2 the result can be seen in Fig. 3.

Let us examine closer what happens in the bug-containing trace. The state data contains three fields: the role, the number of nodes that are down, and the number of monitored nodes. In the state labeled 'state9' we can see that the list of dead processes contain one process, and the list of monitored processes contain four processes. Since processes should not monitor themselves, this is clearly one process too many.

This bug turned out to be a mistake we made ourselves, when implementing the cut-down version of the algorithm. We had been uncareful in the implementation and mixed up variable names. It shows, however, the usefulness and simplicity of the approach.

If we compare the abstract trace in Fig. 2 where the bug is present with an abstraction made from a trace where the bug is fixed in Fig. 4, one can clearly see from the graph structure that the erroneous behaviour is gone.

The first serious bug After correcting this bug, we collected a new set of traces. This time we initially observed no obvious faulty behaviour, we therefore activated the random delaying of messages in the generation of stimuli. Now we

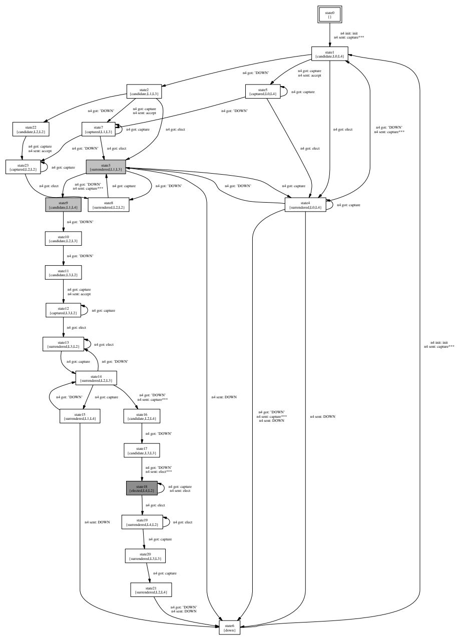

Fig. 2. Abstract trace containing bug

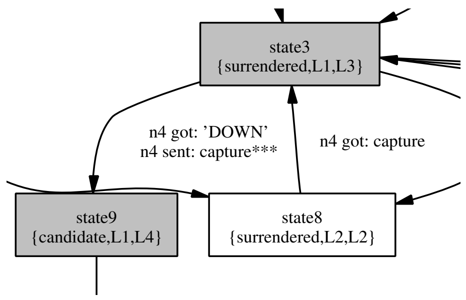

Fig. 3. Faulty part of abstract trace

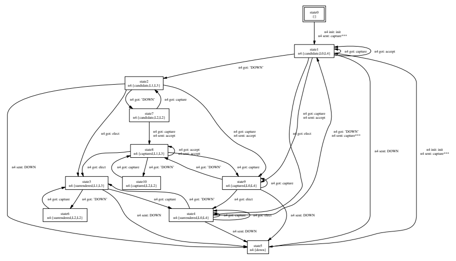

Fig. 4. Bug-free abstract trace

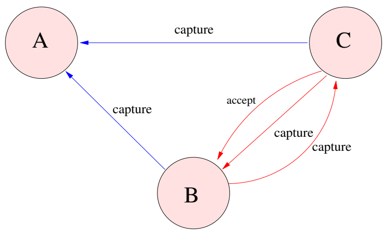

Fig. 5. Deadlock situation in leader election protocol

could observe a faulty behaviour, this time in a leader election system with only three nodes, and again it was a violation of the safety property: Two nodes simultaneously announced themselves as leaders. Again we turned to abstract traces in the search for an explanation. In this case we found the error to be present in situations where many nodes failed simultaneously.

Consider the situation in Fig. 5, where initially only process A is alive and the priority of the processes is A > B > C. If then B and C revive more or less simultaneously and the present leader (A) is suffering from slow connections, it is possible that the newly revived processes will agree on a leader before the present leader is able to announce its presence.

This is indeed a serious bug, and this bug is present also in the original Erlang code. But it is also the case that this situation will not occur if the system is simulated in such a way as to always have more than half of the processes alive. So, we continued the analysis of the protocol with less aggressive stimuli.

2.5 Sanity Checks on Abstractions

We call an abstract trace *sufficient* if all real traces of the system are embedded in it. Note that by construction, it is guaranteed that at least the original trace is embedded in the abstract trace. If all possible traces of the system are embedded, we cover all possible executions of the system. If an abstract trace is sufficient and a property holds for this abstract trace, then it also holds for all real traces. However, in general we do not know whether an abstract trace is sufficient. This is related to coverage and is discussed in Sect. 2.8.

No terminating states There is a basic sanity check we can make in order to check whether an abstract trace is obviously insufficient. We are mainly interested in systems that are constantly up and running, i.e., systems that never terminate. In this setting, we want to be able to represent any (infinite) behaviour of the system with an (infinite) path through the state transition diagram. Any state transition diagram which allows a path that simply stops, i.e. that has states without any outgoing arc, cannot be a correct representation of a continuously running system. Such terminating states can occur because the trace we used to generate the abstraction terminated in the middle of a behaviour that was not covered by any earlier parts of the trace.

Terminating states are automatically reported by our tool.

No quiescent states There are other problematic states where the system can get stuck. Remember that we stimulate the system by taking down and reviving processes arbitrarily during tracing. If there exist a state in an abstract trace that has only one outgoing arc labeled with a 'DOWN'-message of a process, something is wrong as well. This means that the system is in a state where the only way to get out is for a process to die. Since there are no guarantee that processes eventually will die, the system is stuck in that state.

There might be two reasons for this. One is that the abstract trace is insufficient (which means that we should have chosen a different abstraction function, or collected more trace data). The other is that the system has a deadlock in that state (which could indicate an error). Our tool automatically reports such quiescent states.

The second serious bug When we investigate quiescent states for our leader election algorithm, there is a warning for some potential deadlock nodes. Most of those can immediately be discarded, since these are states in which there is a leader elected and hence is not problematic states.

But there is indeed a quiescent state which indicates a real deadlock! In some cases when the leader process dies, the remaining processes end up in a state where a process is waiting for a message that is not going to be sent. Consider the situation in Fig. 6, where all the processes are initially alive, A is the leader and the priority of the processes is A > B > C. Then if A is killed, B and C are notified of this and each receive a 'DOWN'-message. Now, if the message to B is faster than the message to C, it is possible for B to start a new election round and

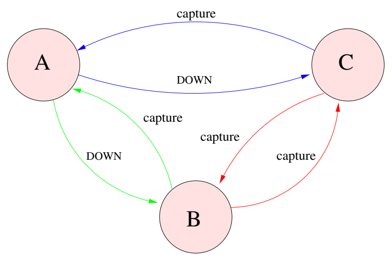

Fig. 6. Deadlock situation in leader election protocol

send a ‘capture’-message to C before C receives the original ‘DOWN’-message. In that case C will simply ignore the ‘capture’-message, since C (falsely) thinks that A is alive and will answer the ‘capture’-message on behalf of C. When C finally gets the ‘DOWN’-message, and starts its new election round B will ignore the ‘capture’-message from C with the motivation that B is higher prioritized than C, which means that C should reply to B’s ‘capture’-message instead. Therefore we end up in a situation where B is waiting for a message that C is not going to send. This deadlock situation is not broken until another node dies or revives.

Thus we have discovered yet another bug in the leader election algorithm, this bug is also present in the original, non-simplified, implementation! The error would probably never occur when all nodes run on similar hardware, however, our addition of delays in messages reveals a very tricky error that may show up in very rare circumstances or when the protocol is used with nodes on different hardware.

2.6 Fault-Tolerant Leader Election Version 2

At this point, we had discovered two serious bugs in the original leader election implementation. We were unable to repair the implementation. So, we decided to try to implement and analyze another algorithm for leader election. Our new algorithm is based on ‘The Bully Algorithm for Synchronous Systems’ in [17], but again we were forced to make some modifications in order to adapt the algorithm to our setting.

The algorithm is quite simple and it is easy to understand how it works. When a process comes up, it first checks whether any process of 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.

The main change we made to the algorithm in the paper was to avoid restarting the election process each time a process revives. This is inefficient and not applicable to the situation where our leader election protocol is supposed to be used. We made the change in two steps, first we changed the algorithm such that no new election would be started if a process with lower priority than the leader revived and later we took care of the situation where a process with higher priority than the leader revived. This second change was surprisingly complex. We also made some changes that did not affect the functionality, but which reduced the number of messages sent by the system. The code is available on the web [18].

After making the changes, we collected a new set of traces. We created some different abstractions and from what we can see in those, the system is working correctly. A couple of states are reported as potential deadlock states, but all of them can be discarded since they are artifacts present because we have an insufficient abstract trace.

2.7 Abstractions for Verification

So far, we have been able to spot errors exhibited by our abstractions either visually or by means of simple sanity checks. However, when the abstractions or desired properties get more complicated, to be sure that an abstract trace obeys a given property, an automated technique is needed. Our idea is to simply formally check properties of the abstract traces using a model checker.

LTL properties We formulate the properties that we want to verify in linear time logic (LTL). In the introduction we mentioned two basic properties for a leader election protocol: (1) There are never two elected leaders at the same time; (2) If the system is stable, eventually a leader will be chosen. For a leader election situation with 3 nodes, the first property can be expressed in LTL as follows:

$$\Box(\neg((l_1 \wedge l_2) \vee (l_1 \wedge l_3) \vee (l_2 \wedge l_3))). \tag{1}$$

Here, l_i is defined to be true exactly when the leader election process running on node i is the elected leader. So, the property can be read as: "It is never the case that node 1 and 2 are leader at the same time, or node 1 and 3, or node 2 and 3."

The second property can be expressed as follows:

$$\Box(\neg(\Box\Diamond(d_1 \vee d_2 \vee d_3)) \Rightarrow \Diamond(l_1 \vee l_2 \vee l_3)). \tag{2}$$

Here, l_i is defined as above, and d_i is true exactly when node i dies. This property can be read as: "It is always the case that if there are not an infinite number of dying processes, eventually there will be a leader elected." Alternatively, flipping the implication sign and negating both sides, we can read the property as: "The only traces where no leader is chosen are those traces where process die infinitely often."

Checking if the above properties hold for a given abstract trace is done using standard LTL model checking techniques [10]: First, a Büchi automaton that accepts counter models of the property is constructed, and then we check if there are accepting runs in the parallel composition P of the system and the Büchi automaton.

Improper cycles Our setting however is a bit special, since we are modeling asynchronous message passing using transition systems. The information needed to represent the real state of our system consists of more than simply the state of the transition diagram; we also need to know what messages have been sent that have not arrived yet.

This problem is illustrated by the typical accepting runs of P we get when looking for counter examples of our properties, which we call *improper cycles*. An

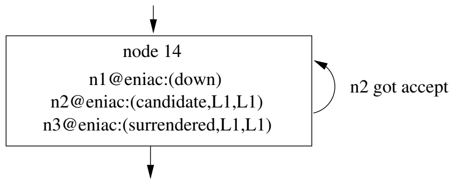

Fig. 7. Loop which is not a possible trace.

example is displayed in Figure 7. A cycle through P that contains a message M that is being received by a transition on the cycle, but not sent by a transition of the cycle can of course never represent a real run of the system.

The example in Figure 7 is a situation that cannot correspond to an actual trace, since such a trace only consumes 'accept'-messages. This could not be an infinite chain of events, since that would mean that an infinite number of 'accept'-messages has to be produced. That is not the case, since we have a closed system and no messages will be input to the system from outside. So, we do not only have to search for accepting runs of P, but we also have to check that the found cycle contains the production of all the messages that are consumed. Our property checker automatically rejects runs that contain such improper cycles.

Results We have checked that both properties 1 and 2 hold for abstract traces of our new implementation of the leader election algorithm, for up to N processes. We have done most of the testing with $N = 3$ and $N = 5$, but has also used larger N ($N = 7$ and $N = 10$). Note that this does not mean that we have formally verified the above properties for the system; only that all generated abstract traces satisfied the properties.

2.8 Coverage

When discussing test-based methods, the issue of coverage is central. Coverage methods should provide some sort of measure of how much of the system one has exercised, and this is important for evaluating the result of the testing. In general, coverage methods can warn of potential situations where we have *not* tested enough; very seldom we can know that we have indeed tested enough. Therefore, it is good practice to apply as many different coverage measures as possible.

Code coverage Erlang has a built-in module, `cover`, for various basic kinds of coverage analysis. It is a very standardized set of tools, which basically provides information of how many times each executable line of code has been accessed. The limitations of point-coverage are well-known.

For our new leader election algorithm, we have traced the system such that we exercised almost all lines of the code. The lines we did not exercise were lines that were solely there for robustness reasons; they were expected never to be run.

Abstract trace coverage Instead of looking at how the actual generated traces have exercises the different parts of the system, we can investigate coverage properties of the abstract traces.

A simple way of doing this is to specify temporal properties of expected events in the abstract traces. For example, for each node, does there exist a state where that node has been elected as leader? For each state and each process, is it possible to reach a state where that process is dead? Does each state have a 'DOWN'-transition going out of that state (except for the state where all processes are dead)?

Such properties can of course be given a quantitative aspect: We can ask how many states there are where all alive processes have a possibility to die. Lastly, a more global kind of coverage measure is to compare the size of the theoretically available state space to the size of the actually reached state space.

For our new leader election algorithm, coverage results are of course affected by how much tracing is done, and how good the stimuli are chosen. It is interesting to study what will happen with coverage numbers for different amount of tracing. The measures that we considered here are the percentage of reached states and the percentage of the states which could be left via a 'DOWN'-transition. The results are not very surprising, the number of reached states as well as the number of nodes with an outgoing 'DOWN'-transition increased with the length of the traces. The numbers are quickly rising for small amounts of tracing, but levels out after further tracing. In the longest traces we reached 87% of the states in the complete state space (it is not entirely clear that all of the states could indeed be reached). About 30% of states had an outgoing 'DOWN'-transition, a somewhat low number. This could be explained by the fact that the stimuli system was not fast enough, so in many situations a killing could not happen with our simulation technique. It is possible that this can be improved with better stimuli generation.

3 Related Work

The leader election protocol has been extensively studied. There are many variations of this algorithm with different assumptions about the network topology and other constraints. Published leader election algorithms are often proved correct on paper, but implementations tend to divert a bit from the actual algorithm, after which correctness is no longer guaranteed. This happened for example with both implementations we studied, which were based on published algorithms [16, 17].

Formal verification and formal testing are supplementary techniques. We deal with real code, whereas there have been other approaches to deal with models of leader election algorithms. For example, the formal verification of the IEEE 1394 leader election protocol [14] has results of that verification cannot be directly applied to our leader election protocol, since different assumptions are made on the network topology and detection of faults. It is an interesting, but still open question, whether we could use our abstraction functions and tracing on an implementation of the IEEE 1394 protocol in order to obtain exactly those abstract traces that are generated by the manual abstraction in [14].

The two other model checking approaches that we are aware of [11, 12] deal with algorithms that have constraints that differ from our case. Model checking is possible because the algorithms that are verified are essentially less complex than the one we consider.

Different from formal testing, we do not have a formal model of the software to generate test cases (e.g. [8, 19, 20]). We more or less construct an incomplete model from the real traces. This model is on one hand shown to the engineers for visual verification and on the other hand input to our model checking approach. Given that we call all our traced events observable, we obtain an abstraction in which all real traces are observable in the abstract trace, however, not vice versa. We use executions of the software to obtain a model for the software with a good coverage and apply model checking techniques on the model to test the software.

Compared to the initial work on trace analysis for Erlang [5], we went further than visualizing the traces as graphs, but we actually performed model checking on those graphs. We improved the trace collection mechanism to simulate delays in communication and to be able to handle events that occur quickly after that tracing starts (events that we missed in the earlier setting). The latter was necessary to be able to deal with re-starting processes.

Another project working with trace analysis is the Java PathExplorer [13]. With this tool it is possible to specify properties for Java programs in temporal logic. The program is instrumented to emit events when executed. The properties are then checked for the event stream. The related tool Java MultiPathExplorer [15] takes the concept a bit further by also being able to generate more possible traces from a single observed trace. This is done by reordering of unrelated events. This technique could be complementary to our method that uses abstractions to generate more possible traces.

4 Conclusions and Future Work

In this paper we describe a case-study in which we use abstraction of traces to analyze a complex software component. By using this technique, we were able to identify two errors in the code. We re-designed the code and verified it by the same technique of trace abstraction, not finding any errors this time.

The described verification methodology of analysis and abstraction of traces is generally applicable to Erlang programs, in particular to the kind of software that is written in industrial projects. The primitives necessary to create a trace are part of the standard Erlang runtime system. Generating traces is rather common testing technology for engineers working with Erlang software. However, so far, engineers look at the output traces as a textual long list of events. By the possibility of visual verification, i.e. inspection of the graphs obtained from an abstracted trace, motivation is created to write those abstraction functions [5]. Compared to writing extra code for testing Erlang code, writing the abstraction functions really is a minor job, since they only address data conversion of state and messages.

The first thing we achieve by using abstracted traces instead of analyzing the real traces, is that there is less 'noise' in the output. With manual inspection of a trace, it makes a difference whether one looks at 2000 long events or a few dozens of short events. The second advantage is that the abstraction allows us to detect cyclic behaviour, which need not necessarily be cyclic behaviour in the original trace. For example, if one abstracts from a time stamp, one would be able to see a certain message repetitively been sent from a certain state, whereas with the time stamp, it occurs as non-cyclic in the trace. The additional cycles not only make the trace shorter, but they also give extra insight in the behaviour of the software. Third, one can prove properties over the abstract traces, which then hold for many more than just the original trace. A property proved for an abstract trace holds for all traces that result in the same abstraction. In that way, we achieve a larger coverage by only looking at a few traces.

Since the methodology of generating traces in general cannot guarantee full coverage, we use it for identifying errors instead of proving correctness. By proving properties that should hold, we know that something is wrong if we get a counter example. If we cannot exhibit the found error in the actual trace, we might have used an inadequate abstraction. For example, the abstract state space may contain cycles that do not correspond to a cycle in the real code. Thus, we can detect errors in the code, but we pay the price of possibly seeing some false negatives. However, these false negatives also give us a certain feedback; they indicate that our abstraction probably can be improved. In such a way, it results in a better insight in the code.

As mentioned in the introduction, part of the AXD 301 software was verified by using a model checking approach. Is the same approach applicable here? First of all, the tool to generate the state space of an Erlang program [4] could not be directly applied to the code. The tool abstracts away from process failures, thus we could only verify all runs in which none of the processes died. Here we could confirm that indeed a leader was selected on all branches.

It is ongoing work to add process failure and recovery to the tool. We added it by hand to the model we obtained from the tool, immediately spotting two major problems. First, there is the obvious state space explosion problem, resulting from the explosion in possible events that can happen in different orders. Second, the way message passing is modeled by the tool is too restrictive and excludes particular orders of events that could happen in reality. Thus, with the present available technology, it is a real challenge to verify the properties we are interested in with a model checker. Therefore we think that one should first apply the much cheaper tracing technology to find errors in the code. In case one cannot find any error, it might be beneficial to generate the whole state space and use the same abstraction functions to reduce the model and prove the properties of interest.

Future Work Possible future work includes automating the creation of the used abstraction functions. We have also considered developing a design document that helps software engineers to quickly create useful abstraction functions.

We would also like to see if it is possible to integrate our abstraction functions with standard model checking techniques based on abstraction. In order to increase the capacity (e.g. number of participating processes) of model checking techniques even more, we probably even need to use symmetry reduction or symbolic model checking.

References

1. M.K. Aguilera, C. Delporte-Gallet, and H. Fauconnier. Stable leader election. In *Distributed Computing, 15th International Conference DISC2001*, volume 2180 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2001.

2. J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.
3. 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.
4. 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.
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. N. Bjørner, U. Lerner, and Z. Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Proceedings of the 2nd International Conference on Temporal Logic*. Kluwer, 1997.
7. S. Blau and J. Rooth. AXD 301 - A new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.
8. E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing and Verification*, VIII:63–74, 1988.
9. J.J. Brunekreef and S. Mauw J.-P. Katoen, R. Koymans. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157–171, 1996.
10. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.
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. H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1-2):171–197, 1996.
13. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189 – 215, March 2004.
14. J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001. special issue of FMICS 1999.
15. K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 337–346. ACM Press, 2003.
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. H. Svensson. Various material related to the paper. http://www.cs.chalmers.se/ ∼hanssv/ erlang_testing.
19. J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.
20. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
21. U. Wiger. Fault tolerant leader election. http://www.erlang.org/.