Simplifying Contract-Violating Traces

Christian Colombo Adrian Francalanza Ian Grima
Department of Computer Science, University of Malta
{christian.colombo | adrian.francalanza | igri0007}@um.edu.mt

Contract conformance is hard to determine statically, prior to the deployment of large pieces of software. A scalable alternative is to monitor for contract violations *post-deployment*: once a violation is detected, the trace characterising the offending execution is analysed to pinpoint the *source* of the offence. A major drawback with this technique is that, often, contract violations take time to surface, resulting in long traces that are hard to analyse. This paper proposes a methodology together with an accompanying tool for simplifying traces and assisting contract-violation debugging.

1 Introduction

Ensuring that real-world complex systems observe contract specifications is a difficult business. Due to the large number of system states that need to be analysed, exhaustive formal techniques such as model checking are generally not feasible. Sound static analysis techniques [11, 16] also suffer from these scalability issues, and often end up being too coarse, ruling out valid systems. Testing — a scalable solution in these cases — is not exhaustive, thus unsound, in the sense that passing a series of tests does not imply that the contract will not be violated once the system is deployed.

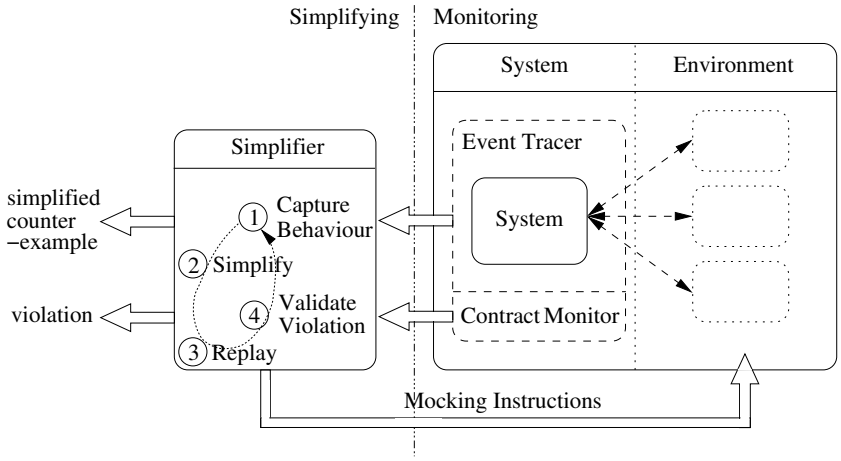

Figure 1: Monitoring a contract (Right) enhanced with counterexample simplification (Left)

A possible technique for dealing with this problem is to complement contract testing with the post-deployment *contract monitoring* — see Figure 1 (Right): the contract is synthesised as a *monitor*, instrumented to run in parallel with the system (executing under an arbitrary environment) so as to check for contract violations *at runtime*. Once a violation is detected, the monitor produces a *violation trace* describing the execution that led to the contract violation and, from this trace, the cause of the violation can be inferred and rectified *manually*[1]. This technique works in principle, and can be used to prevent

[1] We are not aware of automated techniques for pinpointing the source of a contract violation from a violation-trace.

Submitted to:
FLACOS 2012

repeated contract-violations — at some additional runtime cost associated with monitor verification. In practice, however, contract violations may be detected after a long period of monitoring, yielding violation traces that are *too complex* to feasibly analyse manually.

Pinpointing the source of the problem from a violation trace can be facilitated if the trace is *simplified*; this typically involves generating a trace describing a shorter contract-violating execution, perhaps using simpler data values while abstracting away certain events. This debugging aide has proved very effective in both counterexample minimisations in model checking [9, 12] and test shrinking [2, 3, 15].

However, trace simplification in a post-deployment setting poses challenges. In the absence of a system model — as is often the case for real-world systems with a large number of execution states — trace simplification typically relies on *system replaying*: this involves re-running the offending system with simplified parameters and reduced stimuli, in the hope of obtaining a simpler execution trace that still produces the same contract violation. But post-deployment trace simplification based on replaying is complicated by aspects relating to system *capture* and *replay* [3, 10, 13, 14]:

1. In order to replay a system for trace-simplification analysis, one needs to infer the environment stimuli and parameters inducing the contract violation.
2. The system being monitored, together with the environment it executes in, may be non-deterministic, exhibiting different behaviour under identical parameters and stimuli.
3. System replay, which may need to be carried out iteratively, may produce undesirable side-effects such as writing to a database or printing on I/O terminals.
4. System replay may require interactions with either the environment — which cannot be controlled — or with systems that cannot be reset (in order to recreate the same starting point).
5. Certain computation may be too expensive and time consuming for system replay to be a viable method of finding a simpler violation trace.

In this paper we discuss a methodology for simplifying violation traces of large systems in a post-deployment setting; a novel aspect of our methodology is the use of the *contract information assisting the trace simplification process*. We also present a prototype implementation of a trace-simplification tool for violating executions of Erlang programs, based on this methodology.

The rest of the paper is structured as follows: Section 2 describes our proposed methodology. Section 3 discusses an instantiation of this methodology through a tool implementation whereas Section 4 describes a case study using this tool. Section 5 discusses related work and Section 6 concludes.

2 A Methodology for Contract-Violating Trace Simplification

Contracts play a central role in our methodology. Related techniques for post-deployment debugging fundamentally rely on *terminating program executions* (successful and not.) Contracts enables us to extend debugging to non-terminating system executions that violate a contract within a finite execution prefix; they also allowing for better separation of concerns between error definitions and system executions.

The methodology requires the contract to be translated into an *automata representation*, where transition labels correspond to actions recorded in the trace and bad states denote contract violations.[2] Contracts represented as automata simplify monitor synthesis, facilitate the definition of trace simplification (see Section 2.2) and provide a lingua franca for various logics specifying contracts.

[2] Work such as [8] show how contract languages such as $\mathscr{CL}$ can be automatically translated to automata representations.

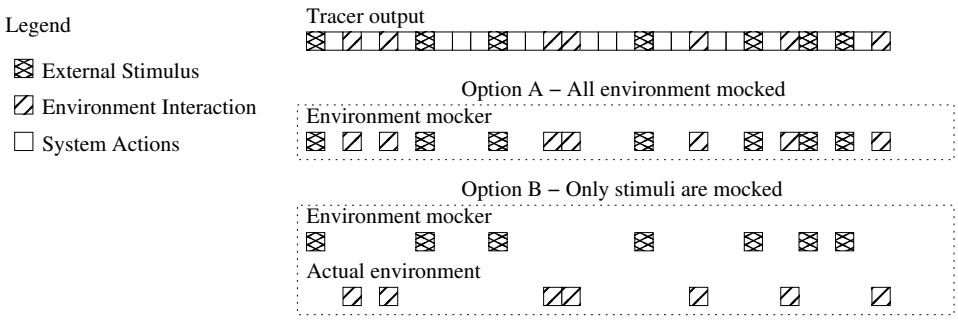

Figure 2: Mocking the environment for replaying counterexamples

As a starting point, our methodology assumes the existence of two items: a violation trace and the corresponding automata-based contract that it violates. Using the mechanism depicted in Figure 1 (Left), the methodology uses the violated contract to guide the search for a simpler violation-trace, which constitutes the main output of the mechanism. For this search, the methodology by-passes any analysis on the system source code since this analysis would not scale well for systems of considerable size. Moreover, the absence of a system model forces the methodology to work with *partial information*, which limits its ability to give stronger guarantees for its output; *e.g.*, it is not able to efficiently state whether the simplified trace is minimal or not, as in the case of counterexample minimisation in Model Checking [9, 12]. This imprecision stems from the fact that the methodology has to use *simulated reruns of the system itself* as an approximating predicate for determining whether a simpler violating trace can be reached: these simulations may either not correspond to actual system executions (see Section 2.1) or be fairly hard to verify because of system non-determinism (see Section 2.2).

2.1 Capture and Replay

In order to be able to rerun the system for trace simplification purposes, the methodology needs to identify the points of interaction it had with the environment so as to be able to replicate the same execution through environment *mocking*; this process is often referred to as *system capture* [10]. It involves the use of an additional subsystem during system execution, whose role is to record and identify the relevant system interaction events so that these can be later replayed (addressing complication no. 1 of Section 1). As shown in Figure 2, the methodology classifies events recorded under three categories:

External Stimuli: These are computation steps *instigated by the environment* on the system, that cause the system to react in certain ways. They can range from method calls, to message sends, to spawning of sub-components in the system.

Environment Interactions: These are computation steps involving interactions between the system and the environment, that are *initiated by the system*. These include synchronous interactions such as method calls (on the environment) and returns, as well as asynchronous interactions such as communication sessions and instructions sent to the environment.

System Actions: These are system computation steps that *do not involve the environment*.

In a limited number of cases (*e.g.*, when the environment is resettable, deterministic and not affected by side effects) it suffices to mock only the environment stimuli, allowing the environment interactions to still occur with the actual environment; see Figure 2, Option B. However, environments rarely have these prerequisite characteristics, and the only other alternative is to mock *both* the environment stimuli and

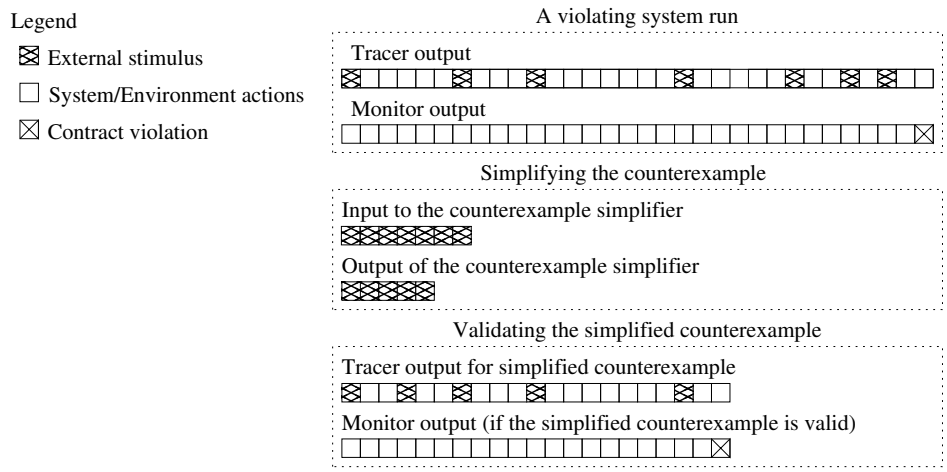

Figure 3: Correct shrinking in terms of generated traces

interactions; see Figure 2, Option A. This option carries disadvantages of its own, in that the replay may be unfaithful to the original violation execution because the environment may be *stateful*: in such cases, the actual environment would have yielded a different system interaction with a simplified trace than the mock extracted from the original violation trace. Notwithstanding this limitation, our methodology favours Option A, because it offers better guarantees *wrt.* confining the side-effects of a simulated system rerun, *i.e.*, the third complication in the list regarding post-deployment trace simplification of Section 1.

2.2 Trace Simplification

Trace simplification assumes the following interpretation for the *simpler-than* trace relation, relying only on the structure of the trace and the contract automata: a violation trace is considered to be simpler than another violation trace whenever:

1. The violation is caused by reaching the same bad state in the contract automaton.[3]
2. It requires less external stimuli, or the same stimuli but with simpler parameters.[4]

The underlying assumption justifying the utility of such a definition for our methodology is that traces reaching the same bad state typically relate to the same system error source. From an operational perspective, this definition also enjoys pleasing properties such as *transitivity*, which adhere to intuitive notions of simplification and facilitates iterative-refinement search techniques. Moreover, the definition integrates well with the methodology mechanism of using the captured system itself as a lightweight *simpler-than* predicate: if the captured system is replayed using a *subsequence* of the external stimuli of the original violation trace, and it still violates the contract by reaching the *same* bad state, then the trace generated is considered simpler. Figure 3 depicts this process, where the top part represents the original trace and monitor output, the middle part represents the extraction of the stimuli and the subsequent stimuli simplification (according to some criteria) while the bottom part shows the outcome of the simplification process of Figure 1 produced from a simplified list of stimuli.

There are other possible definitions for this relation, such as requiring that the trace is shorter in length or else that the simpler trace reaches *any* bad state in the contract automata. Despite their respective

[3] A contract automaton may have more than one bad state, each describing different ways how a contract may be violated.

[4] This assumes some form of ordering over the data domains used.

advantages, these alternatives proved not to be as effective for our methodology. Using the length of the trace as a measure is not compatible with iterative-refinement searching because it yields intermediate results that vary substantially between one another; this happens because we do not have total control over the system execution, even under capture, and decreasing stimuli may actually result in longer traces. On the other hand, using any bad state as a related notion of violation yielded traces that tended to describe violations caused by different sources in the system; our methodology aims to simplify the debugging for violations caused by the same source.

The methodology uses delta-debugging techniques [17] to iteratively refine its search towards an improved solution: the captured system (which includes the synthesised monitor) is replayed under minimised stimuli and parameters as in Figure 3. If the execution yields a violation and the violating bad state is the same as that for the original violation trace, then the trace constitutes a simplification (as defined in Section 2.2); thus the process is repeated using the minimised stimuli and parameters of the simplified trace as the new approximates for our solution. If not, a different minimised set of stimuli and parameters are chosen and the captured system is replayed with the new attempt. When all minimising alternatives are exhausted without yielding a simplified trace, the current simplified approximation is returned as the output of the minimisation process. To limit the search space of simplifying traces and the computational complexity of checking whether a trace is minimal, our notion of a minimal trace is based on *one-minimality* [17]: a trace is minimal if replaying the captured system after removing *any one* of its stimuli does reproduce the bug in the original trace.

There are however complications associated with the plain vanilla adaptation of delta debugging to our methodology. For instance, the system itself may be non-deterministic and may yield different outcomes for the same set of stimuli and parameters — complication no. 2 in the enumerated list of Section 1; in the absence of mechanisms forcing system replays to choose certain execution paths at non-deterministic points of execution, this can affect the precision of delta-debugging such as one-minimality guarantees [17]. Our methodology tries to mitigate this imprecision by performing the same replay a number of times, using a threshold for system replays at which point the search is terminated.

There are however other problems. In particular, the system may print to I/O terminals during replay — complication no. 3 of Section 1 while elements of the system, such as an internal database, may be stateful and not resettable to the state that lead to the original trace violation — complication no. 4 of Section 1. Moreover, the computational cost associated with repeating complex system computations may make iterative replays infeasible — complication no. 5 of Section 1. The solution chosen by our methodology to handle these problems is to *shift* the system-environment boundary we started off with in Figure 1 (Right). More concretely, elements of the system which produce side-effects, or are non-deterministic, can be identified and isolated (maintaining the violation), they can be considered to form part of the environment. A similar procedure can be applied to non-resettable stateful system components and computationally expensive components. This symbolic boundary shift implies that we also *mock* these components with our system capture, thus providing a system-independent, standard way of making the iterative-refinement search more precise and efficient.

3 Trace simplification for the ELARVA Monitoring Framework

We have implemented an instantiation of our methodology[5] as an extension to ELARVA [6], an asynchronous monitoring tool for Erlang [1, 4] (an actor-based programming language). Given the complex-

[5]The tool is freely available from `http://www.cs.um.edu.mt/svrg/Tools/ELARVAplus`. The distribution also includes the case study given in the paper.

ity of distributed industrial systems for which Erlang is usually used, contracts are a natural way how to specify what supplemented forms of behaviour the system parties are expected to adhere to. Concurrency and distribution, inherent to Erlang programs, may yield different thread interleavings each time a system is executed, potentially resulting in non-deterministic behaviour. It is also common for Erlang systems to be programmed to execute without terminating, as in the case of controllers for network switches or elevator systems. These characteristics are conducive to systems with large state spaces, making exhaustive methods of analysis infeasible. As a result, the post-deployment setup outlined in Figure 1, instrumented through ELARVA, constitutes an attractive proposition for ensuring contract adherence.

In ELARVA, contracts are specified as DATE automata [7] where transitions are triples of the form

```
event \ condition \ action
```

with the following semantics: whenever an `event` occurs and, at that instant, the `condition` is satisfied, the automata transitions to the new state and the `action` is performed. ELARVA also supports on-the-fly replication of automata through the `Foreach` construct. More precisely, it specifies a type of contract whereby, whenever an Erlang process executing a particular function is spawned, a corresponding monitor executing a replica automata is launched, typically to monitor activities associated with that process. `Foreach` constructs are particularly useful for keeping contract descriptions compact when monitoring systems with numerous replicated processes; this is often the case for most Erlang systems where process spawning is relatively cheap [4].

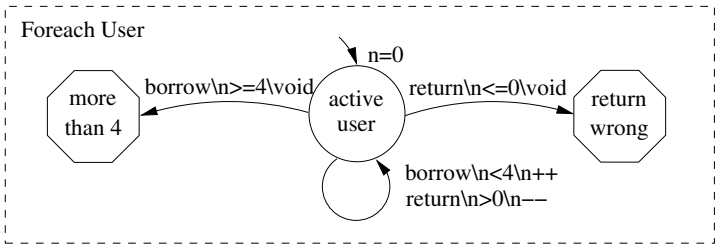

Figure 4: Library System `Foreach` User contract

Figure 4 depicts an example `Foreach` contract specifying that every (process representing a) library user can borrow a maximum of four books and, at the same time, cannot return a book when no books have been borrowed. `Foreach` contracts are violated if *any* replicated automaton that is launched reaches the corresponding bad state (represented by octagons).

In practice, ELARVA monitors Erlang programs by traversing the DATE automata in correspondence to the events read from to the program execution trace, generated by the Erlang Virtual Machine (EVM). Erlang traces record events such as methods calls, communication messages and process spawning: together with the type of the event and the values associated with it, Erlang traces also record the entities producing these events, *i.e.*, the unique ID of the process producing that event. In an ELARVA monitoring setup similar to that in Figure 1 (Right), the EVM, acting as the Events Tracer, communicates events to the Contract Monitor while the system is executing and, as soon as a monitor automaton reaches a bad state, it flags the violation together with the trace justifying the violation detected.

As in Figure 1 (Left), we extend the ELARVA system with a *Simplifier* component which takes the violation trace and the contract as inputs and produces a simplified trace as output; implicitly, the Simplifier also takes the system being monitored as input so as to carry out capture and replay. The default ELARVA setting assumes that the environment consists solely of the user and, as a result, it only mocks the user input and output interactions recorded in the trace. This system-environment boundary may

however be shifted by manually specifying the process IDs recorded in the violation trace that are to be mocked. Boundary delineation is usually a trial-and-error process; at best it can fine tune the mechanism, making the trace simplification processes more efficient and effective; at worst, no simplification is carried out [6] and the original violation trace is returned.

The Simplifier uses an adapted version of a standard algorithm called *ddmin* [17]; this algorithm attempts to incrementally discard parts of the trace stimuli until discarding any more stimuli would result in a non-violating trace.[7] However, in a trace with multiple stimuli, the range of possibilities can be prohibitive. Our adaptation of the algorithm uses the DATE automata to guide this search for stimuli minimisation. In particular, our heuristic is based on the assumption that a process violating a sub-property inside a `Foreach` specification would still violate it if unrelated processes executing in parallel are somehow suppressed, either through removed stimuli, or else through blocking as a result of missing environment interactions from the mocking side. Thus, whenever the Simplifier realizes that the type of contract violated is a `Foreach` specification, it applies two passes of *ddmin* trace reductions. In the first pass, it attempts to identify which processes correspond to different replicated instances of the same replicated automata and, subsequently attempts to incrementally suppress different groups of processes until the minimum set of groups of processes is reached that can still produce the contract violation. In the second pass, the Simplifier applies *ddmin* again, this time on the whole trace of the remaining processes so as to further prune any stimuli which are superfluous for violating the contract. Thus, for the example `Foreach` contract depicted in Figure 4, the Simplifier first attempts find the minimum number of *users* that can contribute to a violation and it afterwards tries to find the minimum number of *stimuli* required by this number of users leading to a violation; see Figure 5.

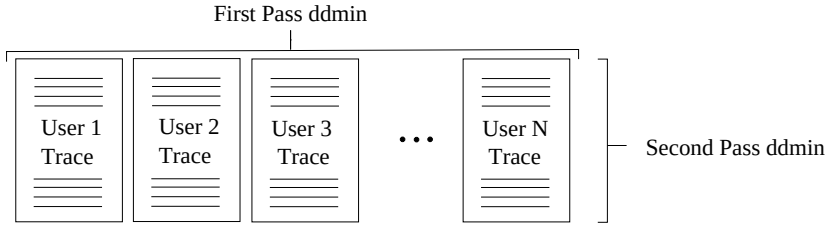

Figure 5: Two pass *ddmin* minimisation for `Foreach` Constructs

4 Case Study

To demonstrate the effectiveness of using contract information for the violation trace simplification, we used the library case study, mentioned briefly in Section 3, which allows users to register, browse through the available books, borrow books, and eventually return the books. The library system should adhere to four contracts, named as follows: (i) *same book twice*: no client can borrow two books with the same name; (ii) *more than four*: no client can borrow more than four books; (iii) *different client*: no client can borrow/return a book using the name of another client; and (iv) *return wrong*: no client can return a book if currently it is not borrowing any. Encoding such contract in terms of DATEs for ELARVA monitoring would result into automata such as Figure 4, which describes contracts (ii) and (iv) together.

[6] Recall that due to non-determinism the violation might not be reproduced during simplification.

[7] In this preliminary implementation, we do not attempt to simplify traces on the basis of simplified parameters. These techniques, used already in popular test-minimisation tools such as [2, 15] are often data-dependent and should be complementary to ours.

Property Violated	Original number of Stimuli	DDMIN		Foreach DDMIN	
		Stimuli	Steps	Stimuli	Steps
same book	23	7	49	4	34
twice	58	4	90	4	35
more than	20	15	85	10	34
four	73	15	279	14	70
different	9	2	18	2	9
client	23	2	28	2	12
return	11	2	15	2	16
wrong	60	2	35	2	23

Table 1: Shrinking performance in different scenarios

To highlight our approach we focus on contract (iv) and give an example of a violation trace and how it is simplified. Consider the scenario where, upon starting the library system, ELARVA reaches the *return wrong* bad state, returning the following trace documenting the violation:

```
[{trace_ts,<0.35.0>,'receive',{newClient,bob},{1339,842747,273000}},
{trace_ts,<0.35.0>,spawn,<0.38.0>,{client,newClient,[bob]},{1339,842747,273001}},
{trace_ts,<0.35.0>,link,<0.38.0>,{1339,842747,273002}},
{trace_ts,<0.38.0>,register,bob,{1339,842747,273003}},
{trace_ts,<0.35.0>,send,{confirm_reg,bob},<0.38.0>,{1339,842747,273004}},
{trace_ts,<0.38.0>,'receive',{confirm_reg,bob},{1339,842747,273005}},
{trace_ts,<0.38.0>,send,{code_call,<0.38.0>,{ensure_loaded,client}},code_server,{1339,842747,273006}},
{trace_ts,<0.38.0>,'receive',{code_server,{module,client}},{1339,842747,273007}},
{trace_ts,<0.38.0>,send,{io_request,<0.38.0>,<0.23.0>,{put_chars,unicode,io_lib,format,[[126,110,42,
45,45,45,32,67,108,105,101,110,116,32,126,112,32,114,101,103,105,115,116,101,114,101,100,32,115,
117,99,99,101,115,115,102,117,108,108,121,32,126,110],[bob]]}},<0.23.0>,{1339,842747,273008}}, ...
```

Apart from stimuli from the environment, the trace also contains all the messaging between the various system processes. For example while the first line is a receipt from the user to add a new client *bob*, the second line is the spawn of the process which will handle *bob*'s requests. Recall that since the system's internal behaviour cannot be steered, our replay and minimisation mechanisms focus on the environment stimuli. Once the trace is filtered from internal system events, the stimuli are the following:

```
[{library,{newClient,ian}},{library,{addBook,fable}}, {library,{addBook,story}},
 {library,{addBook,wish}}, {library,{newClient,bob}}, {library,{addBook,hobby}},
 {ian,{borrowBook,story}}, {ian,{borrowBook, wish}},  {bob,{borrowBook,fable}},
 {ian,{returnBook,story}}, {ian,{returnBook,fable}}]
```

While the above list of stimuli is useful for reproducing the contract violation, it contains entries that do not contribute towards the violation, thus complicating debugging. The ELARVA extension can apply the simplification techniques described above and reduce the list to just two steps:

```
[{library,{newClient,bob}}, {bob,{returnBook,magic}}].
```

Note that the simplified trace obtained is in fact minimal *wrt.* one-minimality, meaning that removing the first element would not have triggered the monitor to start checking `[bob]` while removing the second element would not violate the contract. The simplified trace generated by the above two stimuli is what is outputted by the extended ELARVA, allowing the debugger to pinpoint the source of violation more easily.

To evaluate the simplification capabilities of ELARVA we carried out preliminary tests on the the library system described above. In particular, through these tests we wanted to substantiate our hypothesis

that using contract information, *i.e.*, the *foreach* structure, improves the performance of the simplifying algorithm. Thus for each contract, we identified two violating traces and each trace was first simplified using the plain *ddmin* algorithm and then we simplified the trace using contract information as explained in the previous section. The original number of stimuli, their resulting length and the number of steps required for simplification[8] are shown in Table 1. Using contract information has consistently produced simpler traces and with the exception of one case, it has also taken fewer number of steps to reach the simplified trace.

5 Related Work

Two Erlang software testing tools offering trace simplification to facilitate debugging are PropEr [15] and QuickCheck [2]. Both tools are property-based and perform testing by randomly generating values within the given range and running the functions being tested with the generated values. Should one of the generated test cases fail, the simplification process mutates the failing test case and re-runs the result in order to check if a violation occurs. A simplified test case is considered valid if it produces an error breaking the system's specification and it is simpler than the original failing test case. These tools differ from the extended ELARVA because they are used pre-deployment; since they already drive the environment, they do not need capture mechanisms. Furthermore, PropEr and QuickCheck do not differentiate amongst violations and a successful counterexample simplification is one which generates *any* violation; by contrast, our trace simplification requires violations to match *wrt.* the same bad states.

In principle, our capture and replay approach is similar to SCARPE [10, 14]: as we discussed in Section 2, the tool leaves it up to the user to delineate the system of interest while capturing the interaction of the selected subsystem with the rest of the environment. Although SCARPE is used for debugging purposes, it does not perform any trace shrinking. The target language is also different from ours in that they focus on Java-based systems.

The body of work on JINSI [3, 13] is perhaps the closest to ours. The tool is more mature than our ELARVA extension and is able to use advanced replay mechanisms and techniques such as event and dynamic slicing to considerably improve the efficiency of simplifying counterexamples. However, their approach can only handle *crashing bugs* and *terminating* program executions resulting in "infected" program state.[9] Since our work assumes the notion of a contract, it can handle non-terminating computation as well as a notion of violations that is far richer than crashing bugs. We also employ contract information as a form of contract-guided event slicing, which appears to be novel.

6 Conclusion

In this paper, we have studied post-deployment debugging techniques for contract violations. Our contributions are:

- A methodology that uses contracts to simplify violation traces that are obtained post-deployment, thus facilitating the pinpointing of the defects causing the violation.
- A prototype tool implementation instantiating the methodology, that simplifies Erlang violation traces using contract-based heuristics.

[8] Note that some steps were unsuccessful and did not contribute towards a simpler trace.

[9] In [3] computation has to terminate before the state can be analysed.

When compared to state-of-the-art post-deployment debugging tools such as [3, 13], our methodology also makes fundamental use of contracts to extend debugging beyond traces relating to crashing bugs or terminating programs.

Future Work: We plan to assess better and improve the simplification algorithms used by our prototype tool by exploiting more contract information such as iterating event sequences and dependencies between the events leading up to the violation. We also plan to extend our tool with parameter shrinking techniques used frequently in property based testing [2, 15]. Finally, we plan to integrate thread-level scheduler tools for Erlang such as PULSE [5] which would enable the controlled replay of specific concurrency interleavings mirroring those of the violating trace. This approach should significantly fine-tune our capabilities of reproducing concurrency bugs.

References

[1] Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)

[2] Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with quviq quickcheck. In: ACM SIGPLAN workshop on Erlang. pp. 2–10. ACM (2006)

[3] Burger, M., Zeller, A.: Minimizing reproduction of software failures. In: ISSTA. pp. 221–231. ACM (2011)

[4] Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly (2009)

[5] Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., Wiger, U.: Finding race conditions in erlang with quickcheck and pulse. In: ICFP. pp. 149–160. ACM, New York, NY, USA (2009)

[6] Colombo, C., Francalanza, A., Gatt, R.: Elarva: A monitoring tool for erlang. In: Runtime Verification. LNCS, vol. 7186, pp. 370–374. Springer (2012)

[7] Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: FMICS. LNCS, vol. 5596, pp. 135–149 (2008)

[8] Fenech, S.: Conflict Analysis of Deontic Conflicts. Master's thesis, Dept. of CS, University of Malta (2008)

[9] Gastin, P., Moro, P.: Minimal counterexample generation for SPIN. In: SPIN. pp. 24–38. Springer (2007)

[10] Joshi, S., Orso, A.: SCARPE: A technique and tool for selective capture and replay of program executions. In: ICSM. pp. 234 –243 (2007)

[11] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, NJ, USA (2004)

[12] Nopper, T., Scholl, C., Becker, B.: Computation of minimal counterexamples by using black box techniques and symbolic methods. In: ICCAD. pp. 273–280. IEEE (2007)

[13] Orso, A., Joshi, S., Burger, M., Zeller, A.: Isolating relevant component interactions with JINSI. In: WODA. pp. 3–10. ACM (2006)

[14] Orso, A., Kennedy, B.: Selective Capture and Replay of Program Executions. In: WODA. pp. 29–35 (2005)

[15] Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: SIGPLAN Erlang Workshop. pp. 39–50. ACM (2011)

[16] Pierce, B.C.: Types and programming languages. MIT Press, Cambridge, MA, USA (2002)

[17] Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28(2), 183–200 (2002)