Log-Based Reduction by Rewriting

A. Elyasov

I.S.W.B. Prasetya

J. Hage

Technical Report UU-CS-2012-013
Sept. 2012

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Log-Based Reduction by Rewriting

A. Elyasov, I.S.W.B. Prasetya, J. Hage
Dep. of Inf. and Computing Sciences, Utrecht University, Utrecht, The Netherlands
{*A.Elyasov, S.W.B.Prasetya, J.Hage*}*@uu.nl*

***Abstract*—Software systems often produce logs which contain information about the execution of the systems. When an error occurs, the log file with the error is reported for subsequent analysis. The longer the log file, the harder to identify the cause of the observed error. This problem can be considerably simplified, if we reduce the log length, e.g., by removing events which do not contribute towards finding the error. This paper addresses the problem of log reduction by rewriting the reported log in such a way that it preserves the ability to reproduce the same error. The approach exploits rewrite rules inferred from a set of predefined algebraic rewrite rule patterns, capturing such properties as commutativity and identity. The paper presents an algorithm for inferencing the rewrite rules from logs and a terminating reduction strategy based on these rules. Being log-based the inference algorithm is inherently imprecise. So the inferred rules need to be inspected by an expert before actually being used for rewriting. The approach is language independent and highly flexible. The paper formally defines all used concepts and discusses a prototype implementation of a log reduction framework. The prototype was empirically validated on a web shop application.**

***Keywords*-logging; fault localisation; log reduction; log rewriting; property mining;**

I. Introduction

It is hardly possible to imagine a large modern software system that does not produce any logs during its operation. In some application areas, such as distributed computing, real-time critical embedded applications and operating systems, logging has already become standard practice [1]. For these systems the possibility of failure can not be completely eliminated, because they are often not fully verified or sufficiently tested due to their complexity and size. Therefore, writing logs has been considered as a simple and natural way of providing information about the system behaviour at run-time. This information, for instance, is used for recognition of potential failures, identification of their causes [2], [3], and building system models [4], [5]. Due to the continuous execution process and desire to log as much information about system behaviour as possible, the volume of logs can be excessive, which makes the analysis of logs an even more challenging task [6].

Recently, several tools have been suggested to improve the quality of log messages [7], [8], and this may implicitly simplify the diagnosis of failures from logs. Furthermore, some clustering based tools have been implemented to undertake the failure analysis problem by log reduction [9], [10].

Let us consider the diagnosis process of a typical computer application. The application is constantly writing messages into the log during execution. A log is a sequence of application events being triggered during an execution, together with some auxiliary information. The application may not have been exhibiting any invalid behaviour for a long time, when it suddenly crashes, or reports an error into the log file. This log should be passed for subsequent analysis to identify the root cause of the error so that it can be fixed. If the error cause is not manifested in an obvious way in the log, a person carrying out the failure analysis has to come up with a short reproducible test case that exhibits the same failure. Assuming event reproducibility, the sequence of all events contained in the log, in principle, can serve as such a test case. However, this sequence may consist of thousands of events, and most events may not contribute to the error.

This paper addresses the problem of *log reduction*. We propose a formal approach to this problem based on rewriting. The approach consists of two phases: 1) inferring rewrite rules from already collected logs based on a set of predefined *rewrite rule patterns*; and 2) applying these rules as a rewriting system on logs with the purpose to reduce the original log to a smaller one. The predefined patterns represent common algebraic properties between the application events, such as commutativity and identity. Despite their apparent simplicity, these properties can be successfully used for log reduction. It has been verified on a model web store application example.

The main contributions of this paper are:

- It formally states the log reduction problem.
- A solution is proposed, which exploits the equivalence of event traces with respect to the final states in which the execution of these events can result.
- A terminating and non-increasing reduction strategy is presented.
- Empirical validation is carried out on the Flexstore application example[1].

This paper has the following structure. Section II introduces a motivating example, the GCD application, and shows how the reduction approach works. In Section III-A the formal definitions of log and event trace equivalence are given. The event-state instrumentation model is presented

[1]http://examples.adobe.com/flex2/inproduct/sdk/flexstore/flexstore.html

in Section III-B. The algebraic rewrite rule patterns and the reduction strategy with its properties are discussed in Section III-C and III-D respectively. We talk about the implementation of the log reduction framework in Section IV. Results of the empirical validation are presented in Section V. Related work is discussed in Section VI, and future work in Section VI. Section VIII concludes the paper.

II. Motivating Example

In this section, we introduce an example to illustrate the log reduction approach. The example is a GUI application for calculating the great common divider (GCD) of two natural numbers. The application has three text fields X (the first number), Y (the second number), and R (the result), and two buttons $calc$ and $clear$. When the application is initialised, all fields are empty by default.

An essential part of our approach is to use an *event-state logging model*. This model is discussed in detail in Section III. For now, we only need to know that a log is composed of the alternation of events and states, where the events are the application events and the states are the abstract states of the application. That is, when an event is triggered, we log (serialise) the abstract application state at the beginning and at the end of the event, as well as a description of the event itself, including the values of its parameters. The state is logged after the event is executed. An event starts its execution in the state in which the previous event has finished (except for the first event, which starts in some initial state).

We distinguish the following application events in our GCD example:

- $setX(x)$ — the user assigns the value x to the field X;
- $setY(y)$ — the user assigns the value y to the field Y;
- $calc$ — the user clicks the button $calc$, which changes the value of R;
- $clear$ — the user clicks the button $clear$, which erases all fields.

As an abstract state of the GCD application, let's take the triple (x, y, r), where x, y and r are the values of the corresponding fields X, Y and R. The semantics of our events is described by the finite state automaton (FSA) in Figure 1. Our reduction technique does not require the presence such an FSA. Stronger yet, it may even be that the FSA is not known. Here, the FSA introduced only for presentation purposes to clarify what the events are, and how they affect the abstract state. In the state Q_0 all variables have undefined values; it is the initial state of any execution. If there is a call to the $calc$ event when one of the GCD parameters is undefined, we simply return to the same state, that is, this event does not produce any error.

It may turn out that somewhere in the application there is an error. To work out this case in our model, we pessimistically assume that an exception can be thrown during any execution step. Thus, we draw dashed transitions from

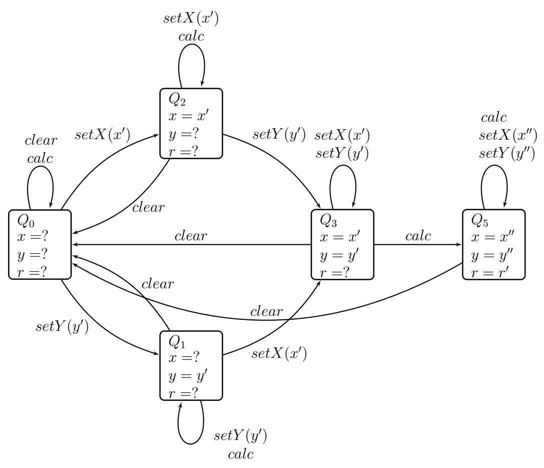

Figure 1. The GCD event execution model.

all states of the automaton to the special "error" state Q_{err} and label such transitions by the virtual event err, as shown in Figure 2. We add inverse transitions from the error state to each normal state to be able to recover from an error execution. In Figure 2 such transitions are labelled by the virtual event err^{-1}.

In the GCD example the transition to the error state can actually happen only if the $calc$ event is executed when one of the GCD parameters has zero value.

Now, let us consider the following execution sequence:

$$\begin{aligned} \tau = setX(3) \rightarrow setY(5) \rightarrow calc \quad & \rightarrow clear \rightarrow setY(1) \rightarrow \\ setX(7) \rightarrow setX(0) \rightarrow setY(2) \rightarrow calc \quad & \rightarrow \mathbf{err} \end{aligned}$$

This sequence leads to an error due to the attempt to compute the GCD when $x = 0$. If we carefully look at the application model in Figure 1, it is easy to notice that there are some equivalences between different execution sequences. For instance, the following equivalences hold for all states of our model:

$$\begin{aligned} \forall e \in Event \; [e \quad ; clear \quad] &= [clear \quad] \\ [setX(x); setY(y) \;] &= [setY(y); setX(x)] \\ [setX(x); setY(x')] &= [setX(x') \quad] \\ [setY(y); setY(y')] &= [setY(y') \quad] \end{aligned}$$

The second rule, for instance, says that the order of assigning values to the fields X and Y does not matter with respect to the final state they produce. Applying these rules step by step to the original sequence τ, we can reduce it from ten entries to only five, such that the reduced sequence preserves the ability to reproduce exactly the same error:

$$\begin{aligned} & setX(3) \rightarrow setY(5) \rightarrow calc \quad \rightarrow clear \rightarrow setY(1) \\ \rightarrow \; & setX(7) \rightarrow setX(0) \rightarrow setY(2) \rightarrow calc \quad \rightarrow \mathbf{err} \\ \equiv \; & \quad \{ \text{ apply [e;clear]=[clear] three times } \} \\ & clear \rightarrow setY(1) \rightarrow setX(7) \rightarrow setX(0) \rightarrow setY(2) \\ \rightarrow \; & calc \rightarrow \mathbf{err} \end{aligned}$$

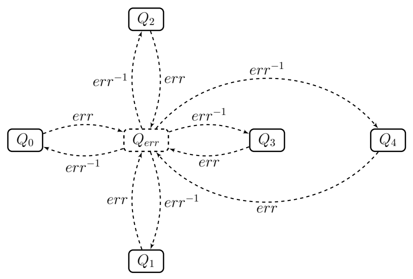

Figure 2. The err and err^{-1} events that complement the GCD model in Figure 1.

$\equiv$ { apply [setX(x);setX(x')]=[setX(x')] }
$clear \to setY(1) \to setX(0) \to setY(2) \to calc \to$ **err**
$\equiv$ { apply [setY(y);setX(x)]=[setX(x);setY(y)] }
$clear \to setX(0) \to setY(1) \to setY(2) \to calc \to$ **err**
$\equiv$ { apply [setY(y);setY(y')]=[setY(y')] }
$clear \to setX(0) \to setY(2) \to calc \to$ **err**

III. Formal Reduction Theory

In this section, we formally describe the reduction problem and present our solution. We start with the description of a logging approach and then define an equivalence relation on logs. This relation lies at the basis of the log reduction, the purpose of which is to decrease the length of a log by rewriting it to a smaller but equivalent one. We propose to learn special rewrite rules from logs that express simple and ubiquitous properties among the events such as commutativity and idempotence. The learning is possible, even if the logs are relatively small (hundreds of entries). And the application of the rules can result in a significant log reduction.

A. Log and Event Trace Equivalence

As we already mentioned, we use the event-state logging model to produce application logs. That is, together with an event we log the preceding and the following application state.

Given an application App, we can think of it as a black box that accepts some input values and returns some output values. At any time, App is in a certain *concrete state*, which completely determines its possible response (output) on a certain input. A concrete state is generally much too large, so we need to abstract away from details to obtain the *abstract state* that can then be serialised into a log.

Definition 1 (Event): Let Evt be the set of events that an application App can dispatch. There is a distinguished event ϵ that does not have any effect on the concrete application state. Each event has a fixed number of parameters.

Definition 2 (Log): A log generated by the execution of the application App is the following sequence:

$$\Sigma = [(\epsilon, s_0), (ev_1, s_1), \ldots, (ev_l, s_l)],$$

where $ev_i \in \texttt{Evt}$, s_i is an application abstract state during the execution, and s_0 is the initial state of the execution.

Definition 3 (Event Trace): The sequence of events $\tau = [ev_1, \ldots, ev_l]$ extracted from the log Σ in Definition 2 is called the event trace of this log.

In some sense, a log is the footprint of an event trace started from a certain state. This is denoted as:

$$\boxed{\Sigma = s_0 \Rightarrow \tau}$$

Let $\texttt{Evt}^*$ be the set of all possible event traces in Evt. Then $\Sigma^* = \Sigma(\texttt{Evt}^*)$ denotes the set of all logs based on the event traces from $\texttt{Evt}^*$.

Definition 4 (State δ-Equivalence): Let δ be a function that maps a state to a set D on which the equality relation $(=)$ is defined. Two abstract states s_1 and s_2 are δ-equivalent ($s_1 \overset{\delta}{=} s_2$) if $\delta(s_1) = \delta(s_2)$. A special case of δ-equivalence is when an abstract state consists of the set of application variables Var. In this case it is called a *state projection*, which is defined as:

$$proj_A(s_1, s_2) = \bigwedge_{x \in A} (proj_x(s_1) = proj_x(s_2)),$$

where $A \subseteq \texttt{Var}$.
Using δ allows us to impose another layer of abstraction over states, that is, to hide or simplify superfluous logging information, which is known to be irrelevant with respect to a particular log analysis.

Definition 5 (Log δ-Equivalence): Two logs Σ_1 and Σ_2 are δ-equivalent ($\Sigma_1 \overset{\delta}{=} \Sigma_2$) if (1) $\texttt{fst}(\Sigma_1) \overset{\delta}{=} \texttt{fst}(\Sigma_2)$; and (2) $\texttt{lst}(\Sigma_1) \overset{\delta}{=} \texttt{lst}(\Sigma_2)$; where the functions fst and lst return the first and the last state of the log respectively.

Definition 6 (Event Trace δ-Equivalence): Two event traces τ_1 and τ_2 are δ-equivalent ($\tau_1 \overset{\delta}{=} \tau_2$) if for all s_1, s_2 such that $s_1 \overset{\delta}{=} s_2$ implies $(s_1 \Rightarrow \tau_1) \overset{\delta}{=} (s_2 \Rightarrow \tau_2)$.
If the function δ is just an ordinary componentwise equality, we denote equivalence of states, logs and traces as $s_1 \equiv s_2$, $\Sigma_1 \equiv \Sigma_2$ and $\tau_1 \equiv \tau_2$ respectively. When it is clear which δ is used, then states, logs, and traces are simply called *equivalent*. For the sake of simplicity, in the rest of this section, we only deal with $\equiv$-equivalence, but the theory also works for an arbitrary chosen δ-equivalence function.

Definition 7 (Log Reduction System): A log reduction system defined by a set of event trace equivalences R on $\texttt{Evt}^*$ is a pair $\Re = (\Sigma^*, \to_R)$.
A reduction step $\Sigma_1 \to_R \Sigma_2$ entails the replacement of an occurrence Φ in Σ_1 ($\Sigma_1 = V\Phi U$) by an equivalent log Ψ, i.e. $(\Phi \equiv \Psi) \in R$. The result of this reduction is a new log $\Sigma_2 = V\Psi U$ that is equivalent to Σ_1 with respect to Definition 5.

In a similar way, we can define *an event trace reduction system* if we were interested only in the reduction of corresponding event traces.

Because of the log reduction, we inevitably lose some concrete state information. This may be the consequence of removing events, but also the result of swapping two adjacent events. For instance, given the log $\Sigma = s_0 \xrightarrow{e} s_1 \xrightarrow{d} s_2$ and the rule $[e;d] \equiv [d;e]$, Σ can be reduced to $\Sigma_1 = s_0 \xrightarrow{d} * \xrightarrow{e} s_2$, where s_1 is replaced by star $(*)$ because the rule does not tell us what the new intermediate state is.

Definition 8 (Maximal Log-Reduction): Given $\Re = (\Sigma^*, \rightarrow_R)$ and $\Sigma \in \Sigma^*$, the log Σ^M is a maximal reduction of Σ in $\Re$ if for all $\Sigma \rightarrow_R^* \Sigma_1^M$ implies that $|\Sigma^M| \leqslant |\Sigma_1^M|$.
There might be possible several different reductions that are all maximal for a given log. They are all equivalent up to the length isomorphism.

B. Event-State Instrumentation Model

To extend an application with logging, a formal *application model* is required. *A control flow graph (CFG)* gives us the right level of abstraction for such a model, as it allows access to every *program point* and instrument it with a *logging statement*.

Definition 9 (Application Model): A model of an application $\mathtt{App}$ is a pair $(\mathtt{CFG}, \mathtt{CState})$, where:

- $\mathtt{CFG} = (\mathtt{N}, \mathtt{E})$ is the application's overall inter-procedural control flow graph. If there is an edge between nodes n_i and n_j, it is denoted as $n_i \rightarrow n_j$. The relation $\overset{*}{\rightarrow}$ is the reflexive and transitive closure of $\rightarrow$ on E. A path from n_i to n_j is written as $n_i \overset{*}{\rightarrow} n_j$.
- $\mathtt{CState}$ is a set of concrete application states.

The formal concept of the application model does not include a mechanism of *error (failure) propagation*. The next few definitions introduce it to our "universe".

Definition 10 (Error): Let $\mathbb{E}$ denote the set of errors an application can throw.

Definition 11 (Application Model with Errors): An application model $\mathtt{App}$ can throw errors from $\mathbb{E}$, which is denoted as $\mathtt{App}_{\mathbb{E}}$, if the application state s is either $s \in \mathtt{CState}$ or $s \in \mathbb{E}$, i.e., $\mathtt{App}_{\mathbb{E}} = (\mathtt{CFG}, \mathtt{CState} \cup \mathbb{E})$.

An application model with errors can also be extended to describe the recovery from an error like in the GCD example. For the sake of simplicity, the recovery model is not formally considered in this paper.

Definition 12 (Partial Execution): A partial execution π of the application $\mathtt{App}$ is an arbitrary path in $\mathtt{App}$'s CFG. We mean partial execution everywhere the term execution is used.

Definition 13 (Execution Trace): An execution trace produced by the execution $\pi = [n_1, \ldots, n_k]$ starting from the state s_0 is the following sequence:

$$s_0 \rightarrow \pi = [(n_1, s_1), \ldots, (n_{k'}, s_{k'})],$$

where k' is an index defined as:

$$k' = \max_{0 \leqslant i \leqslant k} \{i | \; \forall j \leqslant i \colon s_j \notin \mathbb{E}\}.$$

That is, in the case of an error, the execution trace is truncated to include exclusively the part preceding the error. If s_i is always defined, π is called a *feasible execution*, and the corresponding trace is also called feasible; otherwise, they both are *infeasible*. In this paper, whenever execution traces are used, we only mean the feasible ones.

Definition 14 (Application Logging Model): A logging model of an application is a 3-tuple $\mathtt{App}_{\mathtt{LM}} = (\mathtt{App}_{\mathbb{E}}, \mathtt{AState}, \alpha)$, where:

- $\mathtt{App}_{\mathbb{E}}$ is an application model with errors.
- $\mathtt{AState}$ is a set of abstract application states.
- $\alpha \colon \mathtt{CState} \rightarrow \mathtt{AState}$ is an abstraction function.

An application can be instrumented with events. We consider only static unconditional instrumentation.

Definition 15 (Event Instrumentation Model): An event instrumentation model is a 3-tuple $(\mathtt{App}_{\mathtt{LM}}, \mathtt{Evt}, \mathcal{I})$, where:

- $\mathtt{App}_{\mathtt{LM}}$ is an application logging model.
- $\mathtt{Evt}$ is a set of application events.
- $\mathcal{I} \colon \mathtt{Evt} \rightarrow 2^{\mathtt{N} \times \mathtt{N}}$ is an event instrumentation function. For each event $ev \in \mathtt{Evt}$, the function $\mathcal{I}$ returns the set (*instrumentation set*) of begin-end pairs (*instrumentation points*) of ev in terms of nodes. The following conditions must hold for the function $\mathcal{I}$:
  - $(n^b, n^e) \in \mathcal{I}(ev) \Rightarrow n^b \overset{*}{\rightarrow} n^e$;
  - $(n^b, n^e), (n^b, n^{e'}) \in \mathcal{I}(ev) \wedge ev \neq \epsilon \Rightarrow n^e = n^{e'}$;
  - $\mathcal{I}(\epsilon) = \bigcup_{n \in N} (n, n)$.

Definition 16 (Log'): Given an event instrumentation model $(\mathtt{App}_{\mathtt{LM}}, \mathtt{Evt}, \mathcal{I})$, the log produced by an execution $\pi = [n_1, \ldots, n_k]$ of $\mathtt{App}$ from the starting state s_0 is the sequence:

$$\Sigma = [(t_1^b, ev_1, t_1^e), \ldots, (t_l^b, ev_l, t_l^e)],$$

for which there exists a function $\mathcal{J} \colon \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ such that the following conditions hold for the execution trace:

$$s_0 \rightarrow \pi = [(n_1, s_1), \ldots, (n_k, s_k)]$$

- For all $i \in [1, l]$ it holds that:
  (1) $\mathcal{J}(i) = (n_p, n_q)$, where $1 \leqslant p \leqslant q \leqslant k$;
  (2) $\mathcal{J}(i) \in \mathcal{I}(ev_i)$;
  (3) $t_i^b = \alpha(s_p)$ and $t_i^e = \alpha(s_q)$.
- For all $i \in [1, l-1]$ it holds that:

$$\mathcal{J}(i) = (n_{b1}, n_{e1}) \wedge \mathcal{J}(i+1) = (n_{b2}, n_{e2}) \Rightarrow b1 \leqslant b2.$$

- For all $i, j \in [1, k]$ it holds that:

$$(n_i, n_j) \in \bigcup_{ev \in \mathtt{Evt}} \mathcal{I}(ev) \Rightarrow \exists k \in [1, l] \colon \mathcal{J}(k) = (n_i, n_j).$$

Logging separately the beginning and end of the events, we can always reorder the log generated by π according to Definition 16. Even though we did not attach any formal semantics to the events, they are assumed to be deterministic. To carry this property along with the instrumentation, the next few restrictions have to be imposed on the log structure:

- $\Sigma_1 = [(t_1^b, ev, t_1^e)] \wedge \Sigma_2 = [(t_1^b, ev, t_1^{e'})] \Rightarrow t_1^e = t_1^{e'}$, for all $ev \in \texttt{Evt}$.
- $\Sigma = [(t_1^b, ev_1, t_1^e), \ldots, (t_l^b, ev_l, t_l^e)] \Rightarrow \forall 1 \leqslant i < n\colon t_{i+1}^b = t_i^e$, for all Σ.

The implication of the second property is that the log Σ can be represented as a list of pairs — the event and the final state — instead of triples. Namely, it can be rewritten to:

$$\Sigma = [(\epsilon, t_1^b), (ev_1, t_1^e), \ldots, (ev_l, t_l^e)].$$

The given expression for Σ coincides with Definition 2, which means that the logging model defined in this subsection can be used to implement event-state logging model required for log reduction.

C. Rewriting Patterns

As we have already seen in the GCD example, there exist some equivalences between the GCD event traces. This is essentially due to the nature of algorithmic languages that execute constructions such as loops, branches and recursion. Some of the trace equivalences are application specific, but there are also equivalences that commonly occur in many applications. Below we propose three *algebraic rewrite rule patterns* that express equivalences between segments of event traces. These algebraic patterns form the basis for the log reduction.

Skip: $[e(p)] \equiv \epsilon$
Obviously, not all events have an effect on the application. Those events that do not interfere with the abstract state at all fall into the category of this rewrite rule pattern. The GCD application does not have any skip-like events. However, if we project the abstract state only on x and y, that is, apply a δ-function, the $calc$ event becomes skip-like with respect to the state projection $proj_A$, where $A = \{x, y\}$.

Zero: $[e(p); d(q)] \equiv [d(q)]$
Some events may completely overwrite the effect of the preceding events. We call such events zero-like with respect to the preceding events. In the GCD-application, the event $clear$ annuls the effect of any preceding event ($setX$, $setY$ or $calc$). A particular case of this pattern is when $d = e$, that is, whatever the application state is, an execution of $e(q)$ always overwrites the effect $e(p)$. The $setX$ and $setY$ are example events of this particular pattern.

Com: $[e(p); d(q)] \equiv [d(p); e(q)]$
The last pattern asserts the property of two events being commutative. For example, the fields X and Y can be updated in any order, and therefore the corresponding events $setX$ and $setY$ are commutative, in other words they are independent.

In the sequel, when we talk about reduction, we only consider log reduction systems formed by these three patterns: $Skip$, $Zero$ and Com.

Definition 17 (Algebraic Log Reduction System): We call the log reduction system $\Re_A = (\Sigma^*, \rightarrow_{R_A})$ algebraic if for all $r \in R_A$, r is of one of the types $Skip$, $Zero$ or Com.

D. Reduction Strategy

The following auxiliary functions are used to retrieve the rules corresponding to a given rewrite rule pattern:

$$\begin{aligned} R_s &= \{r \in R \mid isSkip(r) \text{ — is } r \text{ a } Skip \text{ rule?}\} \\ R_z &= \{r \in R \mid isZero(r) \text{ — is } r \text{ a } Zero \text{ rule?}\} \\ R_c &= \{r \in R \mid isCom(r) \text{ — is } r \text{ a } Com \text{ rule?}\} \end{aligned}$$

Algorithm 1 A reduction strategy for an algebraic event trace reduction system

```
procedure ReduceLog($R_s, R_z, R_c, \tau$)
    $(R_s, R_z, R_{mz}, R_c) \leftarrow InferRules(R_s, R_z, R_c)$
    $R_z/_\sim \leftarrow splitZeroCls(R_z)$
    $R_{mz}/_\sim \leftarrow splitZeroCls(R_{mz})$
        ▷ $splitZeroCls(R) = \{r_x | r_x \subset R, \forall r \in r_x : r = (yx \equiv x)\}$
    $\tau \leftarrow applySkip(R_s', \tau)$
    repeat
        $\tau \leftarrow ZeroReduce(R_z/_\sim, R_c, \epsilon, \tau)$
        $\tau \leftarrow reverse(\tau)$
        $\tau \leftarrow ZeroReduce(R_{mz}/_\sim, R_c, \epsilon, \tau)$
        $\tau \leftarrow reverse(\tau)$
    until $FIX(\tau)$
    return $\tau$
end procedure

procedure InferRules($R_s, R_z, R_c$)
    $R_s' \leftarrow inferSkip(R_s, R_z)$        ▷ $ab \equiv b \wedge b \equiv \epsilon \Rightarrow a \equiv \epsilon$
    $R_z' \leftarrow inferZero(R_z)$        ▷ $ab \equiv b \wedge ca \equiv a \Rightarrow cb \equiv b$
    $R_{mz} \leftarrow inferMZero(R_z \cup R_z', R_c)$
                                        ▷ $ab \equiv ba \wedge ab \equiv b \Rightarrow ba \equiv b$
    $R_{mz}' \leftarrow inferZero(R_{mz})$
    return $(R_s \cup R_s', R_z \cup R_z', R_{mz} \cup R_{mz}', R_c)$
end procedure

procedure ZeroReduce($R_z/_\sim, R_c, u, v$)
    $(r_x, (w, x', v)) \leftarrow findFirstOccur(R_z/_\sim, v)$
    if ($x'$ = Just $x$) then
        $u \leftarrow uw$
        repeat
            repeat
                $u \leftarrow applyZero(u, r_x)$
                                ▷ $r = (yx \equiv x) \wedge u = u'y \rightarrow u = u'$
            until $FIX(u)$
            $u \leftarrow applyZeroWithCom(u, r_x, R_c)$
                                                                        ▷
  $r = (yx \equiv x) \wedge u = u'yu'' \wedge (\forall e \in u'' : com(e, y)) \rightarrow u = u'u''$
        until $FIX(u)$
        $ZeroReduce(R_z/_\sim, R_c, ux, v)$
    else
        return $uv$
    end if
end procedure
```

There are some properties that we would often like to have such as termination and confluence [11]. Algorithm 1 presents a reduction strategy for an algebraic event trace reduction system that can be proved to be terminating and non-increasing. It terminates in polynomial time, but does not necessarily produce the maximal possible reduction that is reachable with a given set of rewrite rules. The reduction is carried out by the procedure `ReduceLog`, which consists of the following key steps:

1) Enriching the original set of rules is implemented by the `InferRules` procedure, which reminds of an analogous step of the Knuth-Bendix algorithm [11]. The procedure considers overlaps of $Skip$ and $Zero$ rules (`InferSkip`), $Zero$ and $Zero$ rules (`InferZero`), and $Zero$ and Com rules (`InferMZero`). For example, taking the overlap of the rules $ab \to b$ and $bc \to c$ and applying them in a different order to the sequence abc, we get $ac \to c$ as a new possible reduction rule.
2) The function `splitZeroCls` groups $Zero$ rules with equal right hand sides in the same equivalence class.
3) The function `applySkip` removes all occurrences of $Skip$ events.
4) The last reduction step is the recursive application of the `ZeroReduce` procedure to the event trace in both directions (from left to right and right to left) until a fixed point is reached. To express the fixed point computation, Algorithm 1 uses the special form of repeat-until loop (the top line of the equivalence), which is only a syntactic sugar for the function $Fix(F, \tau)$ (the bottom line of the equivalence), defined as follows:

$$\begin{aligned} &\textbf{repeat } F \textbf{ until } Fix(\tau) \\ Fix(F,\tau) := &\Updownarrow \\ &\textbf{if } F(\tau) \equiv \tau \textbf{ then } \tau \textbf{ else } Fix(F, F(\tau)) \end{aligned}$$

The procedure `ZeroReduce` looks for the first occurrence of x in v, where r_x is a class in $R_z/_\sim$ (`findFirstOccur`), and then it exhaustively applies all rules from r_x to the left part of v that starts from x (`applyZero`), and it also tries to combine zero with commutative rules (`applyZeroWithCom`). The later combination allows to discover zero reductions that are not enabled by default. Crossing zero and commutative rules gives us the *mirror zero rules* (line 18), which are like zero with respect to the reversed event trace. These rules are used at line 10 to get a reduction by means of the application of the `ZeroReduce` procedure to the reversed trace.

Theorem 1 (Termination): For the algebraic log (event trace) reduction system $\Re_A = (\Sigma^*, \to_{R_A})$, the reduction relation $\to_{R_A}$ implemented by Algorithm 1 is terminating.

Note, that during reduction Algorithm 1 effectively applies only $Skip$ and $Zero$ rules in the decreasing direction, that is, they form a non-cyclic reduction system. In order to prove termination, we need to show that the fix points are guaranteed to be reachable in all three cases (lines 28, 27 and 7). But this obviously follows from the facts that at each time the log is either reduced or not, and the number of rules is finite. If the log can not be reduced anymore, we have reached the corresponding fixed point.

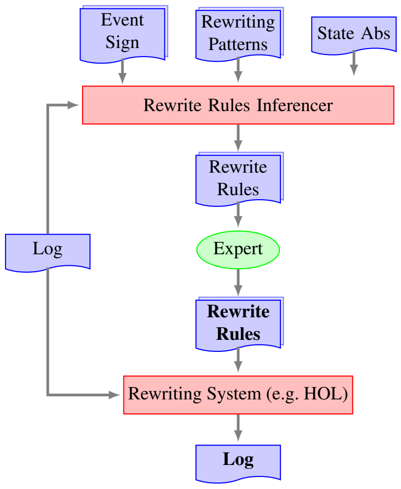

Figure 3. Log Reduction Framework

Theorem 2 (Length Reduction): For an algebraic log (event trace) system $\Re_A = (\Sigma^*, \to_{R_A})$, the reduction relation $\to_{R_A}$ implemented by Algorithm 1 is non-increasing. This property follows from the shape of reduction being applied. The algorithm only uses non-increasing reductions.

Lemma 1 (Zero-Based Log Reduction System): For a given $\Sigma \in \Sigma^*$, the maximal reduction of Σ in $\Re_z = (\Sigma^*, \to_{R_z})$ is equal to the maximal reduction of Σ in $\Re_z^{\gtrless} = (\Sigma^*, \to_{R_z^{\gtrless}})$ and the reduction relation $\to_{R_z^{\gtrless}}$ is confluent, where $R_z^{\gtrless} = R_z \cup \{ac \to c \mid (ab \to b) \in R_z \wedge (bc \to c) \in R_z\}$.

IV. Implementation

Our log reduction framework, including the inference and reduction part, have been implemented in Haskell. The complete structure of the framework is depicted in Figure 3. As an input, it accepts a log for reduction, a list of predefined rewrite rule patterns (for example, the ones presented in Section III-C), the event signatures (event names and arities) and a state abstraction function. The Rewrite Rule Inference block analyses the initial log and produces the concrete rewrite rules, which are not guaranteed to be either sound or complete. Therefore, user intervention is required to remove the false positives (Expert block). Finally, the initial log can be reduced using the filtered set of rewrite rules. In our earliest experiments, we tried to adopt the rewriting capabilities of HOL [12] to carry out log reduction. To do this, we literally translated our rewrite rules into the HOL format, and used HOL's built-in automatic rewriting tactics to reduce the log. But it led to either an infinite rewriting loop or a weak reduction if we had chosen and fixed the direction of rule application in order to handle termination. That is, if we had had a rule $ab \equiv ba$, we should have decided whether apply it as $ab \to ba$ or $ba \to ab$. As a result, Algorithm 1 has been implemented, which ensures termination and provides a better reduction then the one in HOL. The algorithm is implemented in Haskell, but we could integrate it into HOL, and thereby being able to use the whole bunch of HOL tactics to prove some additional facts about our reduction rules.

V. Empirical Validation

To evaluate our framework, we applied it to the web shop application flexstore. It is a standard example of an application for buying mobile phones, provided by adobe to demonstrate some features of the Flex SDK. The flexstore has the usual ingredients of a web shop, such as a catalog, various filters and a shopping cart. Therefore it is representative enough to be used for the evaluation of our reduction framework.

For the flexstore, we can define high level events such as "add phone to the cart", "compare several phones", "show all phones satisfying a certain criterion". These and many others flexstore events were instrumented by the use of the FITTEST Automation Framework (AF) [13], [14]. AF allows us to log application events and the application abstract states associated with them in the FITTEST Logging Format [13]. The user of AF provides an application abstract function, which is the collection of objects and fields associated with them. The resulted log fully conforms to the definition of log given in Section III-A.

All experiments presented in this section were carried out on an Intel i5 (2.4 GHz) machine with 6GB of RAM under Ubuntu 12.04 OS. We generated a log of length 11000 entries, randomly invoking different flexstore events out of the 23 possible events. We considered two different state projections: high abstraction (2 variables, **Abs** $= high$) and low abstraction (7 variables, **Abs** $= low$). False positives were filtered by the use of a confidence level (**Conf** $= yes$), i.e., all rules with a confidence level lower than 0.99 were not accepted. Without the confidence level (**Conf** $= no$) it was sufficient for a rule to have at least one positive witness and zero negatives to be accepted as a rewrite rule.

A. Inference Results

As we mentioned already, the inference algorithm can both report false positives and reject false negative rules. Therefore, an expert assessment is required to at least sift out the false positive ones, otherwise, we might get a wrong reduction sequence. We judged the results of the rule inference from the log of 11000 entries, and used them as a *template* to measure the number of erroneously accepted and rejected rules in all other measurements in the experiment.

Table I
Results of pattern inference for the flexstore. The events recognised as skip are excluded from the consideration for candidates to form $Zero$ or Com rule.

Patterns	**Abs**	**Conf**	5000			2500			1000			500			100		
			m	p	n	m	p	n	m	p	n	m	p	n	m	p	n
Skip	low	yes	7	0	0	7	0	0	7	0	0	6	0	1	2	0	5
	low	no	7	0	0	7	0	0	7	0	0	7	0	0	7	4	0
	high	yes	9	0	0	9	0	0	9	0	0	8	0	1	2	0	7
	high	no	9	0	0	9	0	0	9	0	0	9	0	0	9	5	0
Zero	low	yes	15	6	11	13	13	13	11	22	15	4	5	22	0	0	26
	low	no	25	39	1	23	42	3	17	42	9	12	37	14	0	4	26
	high	yes	1	9	8	1	17	8	0	31	9	0	7	9	0	0	9
	high	no	8	15	1	6	30	3	3	46	6	2	42	7	0	4	9
Com	low	yes	8	3	2	2	0	8	0	0	10	0	1	10	0	0	10
	low	no	10	5	0	6	8	4	2	3	8	1	1	9	0	0	10
	high	yes	6	4	1	1	1	6	0	0	7	0	1	7	0	0	7
	high	no	7	9	0	4	11	3	1	5	6	0	2	7	0	0	7

The aggregated data of the inference part are shown in Table I. We used the logs of different sizes (initial segments of length from 100 up to 5000 entries taken from the original log) to infer the rules, and then we compared the inference outcome with the template. As a result, we calculated the number of correctly identified rules (m column) as well as the number of false positives (p column) and negatives (n column). Because of the abstraction, we might get some new rules as well as lose some old ones. It is clear from Table I that to correctly identify all skip rules, it was already sufficient to have 1000 entries in the log. But for $Zero$ or Com rules we had missed or wrongly accepted some rules even for the log of 5000 entries. The choice of an appropriate confidence level is a trade-off between the number of false positive rules we want to avoid and the number of potential false negatives we might lose because of being too exact. As we can see, in case of the $Zero$ and Com rules, there are dozens of candidates, so it is wise to rely on the confidence level to decrease the number of false positives, especially if the amount of data in the logs is limited. For instance, we got rid of 33 false positive rules (reported 6 instead of 39 rules) of the type $Zero$, learnt from the log of 5000 entries.

B. Reduction Results

Table II presents the reduction results of the log of 5000 entries that were achieved by Algorithm 1 using the rules from Table I. In the best case, we managed to reduce the original log by 65% (1727 entries), and, in the worst case, we got only 35% reduction (3259 entries). The higher level of abstraction did not give us better reduction power, even though, in the abstracted log, we had more skip like events (9 vs. 7), but at the same time, we had less zero-like rules. The

Table II
Results of 5000 entries log reduction for flexstore

Abs	**Conf**	11000	5000	2500	1000	500	100
low	yes	1727	1846	1907	1963	2243	3259
low	no	1727	1739	1739	1882	2023	2327
high	yes	1954	2159	2159	2181	2393	3259
high	no	1954	1973	2023	2141	2150	2181

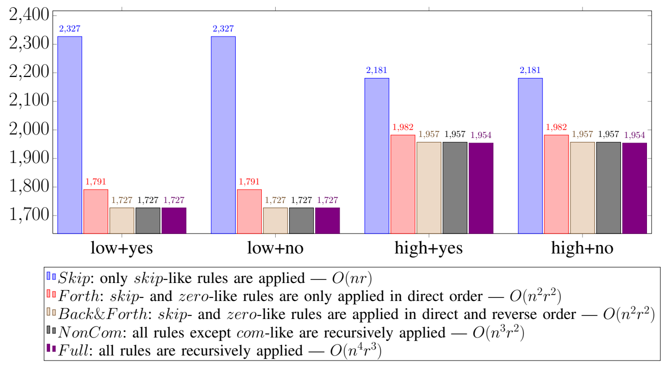

Figure 4. Reduction of the log l where $|l| = 5000$ by different strategies.

use of a confidence level allowed to reject some incorrect rules, but it also led to the rejection of some correct ones, which finally affected the reduction.

Figure 4 shows the comparison of the reduction power that was reached by the different variations of Algorithm 1 when applied to a log of length 5000. It turns out that we already obtanied a 50% reduction by only removing all $skip$ events. Additionally, reducing all immediate instances of zero rules after only a single traversal we obtained 4-11% extra reduction. In this experiment we did not observe any additional reduction due to the recursive application of $Back\&Forth$, i.e. $NonCom$. The reduction achieved by the application of Com together with $Zero$ allowed us to remove 3 additional events out of 1957. So there is only little contribution from the reduction by combination of $Zero$ and Com rules. This is a consequence of the randomness of event invocations during the generation of our experimental log data. We expect, the real user behaviour scenarios to exhibit more cases of the reduction by application of $Zero$ together with Com rules.

VI. Related Work

Since our approach essentially consists of two parts: learning specific rules from logs, and then applying the reduction procedure based on these rules, we accordingly split the related work section.

A. Mining Properties From Logs

There are various categories of properties that can be learnt from logs. But these properties come under different names: invariants [15], specifications [16] or oracles [17].

The Daikon tool [15] discovers *assertions* (invariants) that hold at certain program points, e.g., method entry and exit. The assertions are templated predicates over program variables. They express invariants such as constant equality and ordering. The inference of polynomial and array invariants, which Daikon is unable to discover, is presented in [18]. DySy obtains invariants that are specific for the observed program executions by using symbolic analysis [19]. The Daikon approach can be extended by the inference of behavioural models of an application [4], which are extended finite state machines that describe the interplay between data values and component interactions.

Temporal properties have also been thoroughly investigated. Therefore, several tools for learning them have been developed in recent years [20], [5], [21]. In contrast to Daikon's properties, which represent data-flow dependencies, the temporal properties describe control-flow relations, for instance, the precedence of function calls. The Perracotta tool [21] infers the strictest template, with respect to the hierarchy proposed for the *Response pattern* [22] (P must be followed by S), that any two events satisfy. A more general class of temporal properties can be inferred by the Ocd tool [20] — a tool for learning and enforcing temporal properties over function and method call sequences. The properties are predefined by templates, which are two-letters regular expressions (ab, ab^+, etc.). Synoptic [5] mines temporal properties, such as a always followed by b, a always precedes by b, from partially ordered logs, and then uses these properties to visualise the application model.

The properties proposed in this paper belong to the category of *algebraic properties* (specifications) [23]. Hankel et al. [16] suggest discovering algebraic specifications by exercising the terms (dynamic part) generated from the algebraic signatures of program classes (static part). Adiheu [24] improves this approach by using Adabu's sequences of legal operations [25] expressed as non-deterministic FSA .

B. Log Reduction

The reduction always leads to the loss of some information, in particular, we have to sacrifice the completeness of logged data. But there should exist an invariant that continues to hold during the reduction. For instance, someone might be interested in the presence of certain events in the reduced log. The invariant discussed in this paper has the ability to reproduce the same failure as the one contained in the original log. This section discusses different approaches to the log reduction irrespective of the chosen reduction invariant.

Wang and Parnas [26] suggest to use trace specifications of software modules as a basis for trace rewriting to simulate module behaviour. The trace specifications completely describe the effect of an event trace execution on the application. In [26] the authors present a smart trace rewriting, which is proved to be terminating and confluent. Due to these properties, the smart rewriting, moreover, gives the maximal possible reduction. But a formal trace specification is often absent, and the issue of its inference brings us back to the questions raised in Section VI-A, namely how to get the specification in the first place.

Clustering techniques are found to be quite useful to reduce the length of logs, in particular, if logs are unstructured. So several clustering algorithms and tools have been introduced [27], [28], [29], [30], [31]. Clustering assists reduction in two possible ways: 1) grouping similar events in clusters and use one event as a representative of the entire cluster; and 2) learning event correlations [10]. Moreover, outliers (those events that do not fall in any of the existing clusters) are potential candidates to be considered as anomalies.

Zawawy et al. [32] propose to filter logs with respect to the set of analysis goals and diagnostic hypotheses in order to assist root cause analysis. Two reduction strategies are suggested: 1) filtering events that are irrelevant to the failure by executing SQL queries generated from annotated goal models [33] capturing application requirements, and 2) applying Latent Semantic Indexing [34] to identify log entries connected with the query representing a particular aspect of the model. Kontogiannis et al. [9] suggest to reduce logs according to the chosen upfront sequence of *beacon* events. This approach exploits a collection of event dependency relations to construct the *Event Dependency Graph*, and then by clustering to extract all events correlated to the beacon sequence.

Delta Debugging [2] is able to reveal the *cause-effect chain* of a failure, isolating the relevant variables and values. This chain is essentially the reduced log we are looking for. The method compares the state difference between a passing and failing run. This requires the ability to replay the failing execution multiple times, which we do not need.

Lee et al. in [35] consider the reduction of log replaying, retaining the ability to reproduce the failure. The reduction is reached by reducing the amount of information that needs to be logged in order to replay the execution. The reduction is carried out at the unit level (loop iterations) by the offline analysis of the *enhanced log* — a log resulting from the execution of an instrumented program to collect some axillary runtime information.

BugRedux [36] synthesises and reduces in-house executions that could reproduce the failures observed in the field. BugRedux exploits additional information, for instance call sequences or complete traces, to produce an input that mimics the execution by means of symbolic analysis. Our reduction technique is purely based on the information presented in log files, and it does not require the application source code to be available.

An FSA is a common way to represent an application model. Such a model can be learnt from logs as in [3], [37] and used for root cause analysis. A failure is then recognised by observing that an execution trace is inconsistent with the FSA. The point of divergence indicates where the abnormal behaviour has started, and the application model can be used to find the shortest path to this point. This gives us a reduced failing execution trace. But, of course, building the precise application model from logs is an expensive task, and the lightweight approaches to reduction might be preferable.

VII. Discussion and Future Work

The reduction algorithm for algebraic log reduction systems presented in this paper, as we mentioned in Section III-D, does not guarantee reaching a maximal reduction. We have not yet investigated the decidability of this question. Even if the answer is positive, it is still desirable to provide an efficient algorithm for maximal reduction. These questions are more clearly formulated in the Appendix of this paper.

Apart from the three rewrite rule patterns discussed in Section III-C, there exist some other rule patterns such as $[e(p)] \equiv [e(q)]$ and $[e(p)] \equiv [d(p)]$. The former pattern suggests to ignore the concrete values of the parameters, when the latter one says that two events are simply equivalent if they are applied to the same parameters. In general, some practical issues appear if we want to extend our reduction framework with new rewrite rule patterns, namely:

- How to infer the concrete instances of the rewrite rules for this pattern? The inference might be very expensive, require a huge amount of logging data, or generate too many false positives.
- How to incorporate a new pattern into the existing reduction algorithm or to build a new one that is as effective and powerful as the former one?

The answers to these questions in many cases require a tradeoff between the pattern complexity, inference efficiency and reduction power. These problems are beyond the scope of in this paper, and we consider them future work.

The inferred rewrite rules, after they were inspected by an expert, become application invariants that need to hold over all executions. A violation of any rule is a potential signal of an error in the application. Therefore, the rules can be used as testing oracles. The strength of these oracles can be assessed by applying mutations to the source program and validating the oracles on the modified version of the program. Moreover, it is interesting to compare the number and types of the mutations caught by our rules and well-known tools like e.g. Daikon [15].

VIII. Conclusion

In order to discriminate failures that might happen during the execution of an application, the programmers are trying to provide as much informative logs as possible. But if a failure occurs, we are not interested in all this excessive information to carry out the root cause analysis of that particular failure.

This paper addresses the issue of log reduction that arises if we consider in-house or in the field debugging. We propose to infer algebraic properties among application events, and use them as the basis for our reduction system. We built a prototype of the log reduction system based on these rules and applied it to the flexstore application. As a result, we managed to get 65% reduction of the original logs.

Acknowledgements

This work is funded by the EU FITTEST project No. 257574. We would like to thank Arie Middelkoop for his contribution to the efficient implementation of the search procedure in the pattern inference algorithm. And special thank to Jan Rochel for our discussion on rewriting.

References

[1] Y. Liang, A. Sivasubramaniam, and J. Moreira, "Filtering failure logs for a BlueGene/l prototype," in *DSN*, 2005, pp. 476–485.

[2] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.

[3] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *ISSRE*, 2008, pp. 117–126.

[4] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE*, 2008, pp. 501–510.

[5] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *ESEC/FSE*, 2011, pp. 267–277.

[6] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, pp. 55–61, 2012.

[7] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *SIGARCH Comput. Archit. News*, vol. 39, pp. 3–14, 2011.

[8] D. Yuan, S. Park, and Y. Zhou, "Characterising logging practices in Open-Source software," in *ICSE*, 2012.

[9] K. Kontogiannis, A. Wasfy, and S. Mankovskii, "Event clustering for log reduction and run time system understanding," in *SAC*, 2011, pp. 191–192.

[10] W. Zhou, J. Zhan, D. Meng, D. Xu, and Z. Zhang, "LogMaster: Mining event correlations in logs of large scale cluster systems," *CoRR*, 2010.

[11] R. V. Book and F. Otto, *String-rewriting systems*, 1993.

[12] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: a theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press, 1993.

[13] I. S. W. B. Prasetya, A. Middelkoop, A. Elyasov, and J. Hage, "D6.1: Fittest logging approach," 2011.

[14] A. Middelkoop, A. Elyasov, and W. Prasetya, "Functional instrumentation of actionscript programs with asil," in *IFL*, ser. LNCS, 2012, vol. 7257, accepted for publication.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, pp. 35–45, 2007.

[16] J. Henkel and A. Diwan, "Discovering algebraic specifications from java classes," in *ECOOP*, 2003, pp. 431–456.

[17] S. R. Shahamiri, W. M. N. Wan-Kadir, S. Ibrahim, and S. MohdHashim, "An automated framework for software test oracle," *Information and Software Technology*, 2011.

[18] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *ICSE*, 2012, pp. 683–693.

[19] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: dynamic symbolic execution for invariant inference," in *ICSE*, 2008, pp. 281–290.

[20] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *ICSE*, 2010, pp. 15–24.

[21] J. Yang and D. Evans, "Dynamically inferring temporal properties," in *PASTE*, 2004, pp. 23–28.

[22] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999, pp. 411–420.

[23] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica*, vol. 10, pp. 27–52, 1978.

[24] C. Ghezzi, A. Mocci, and M. Monga, "Efficient recovery of algebraic specifications for stateful components," in *IWPSE*, 2007, pp. 98–105.

[25] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *WODA*, 2006, pp. 17–24.

[26] Y. Wang and D. L. Parnas, "Simulating the behavior of software modules by trace rewriting," *Software Engineering, IEEE Transactions on*, pp. 750–759, 1994.

[27] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IPOM*, 2003, pp. 119–126.

[28] A. Makanju, S. Brooks, A. Zincir-Heywood, and E. Milios, "LogView: Visualizing event log clusters," in *PST*, 2008, pp. 99 –108.

[29] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *KDD*, 2009, pp. 1255–1264.

[30] N. Taerat, J. Brandt, A. Gentile, M. Wong, and C. Leangsuksun, "Baler: deterministic, lossless log message clustering tool," *Comput. Sci.*, pp. 285–295, 2011.

[31] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *J. Softw. Maint. Evol.*, pp. 249–267, 2008.

[32] H. Zawawy, K. Kontogiannis, and J. Mylopoulos, "Log filtering and interpretation for root cause analysis," *ICSM*, pp. 1–5, 2010.

[33] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite, "Reverse engineering goal models from legacy code," in *RE*, 2005, pp. 363–372.

[34] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, 1990.

[35] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang, "Toward generating reducible replay logs," in *PLDI*, 2011, pp. 246–257.

[36] W. Jin and A. Orso, "BugRedux: Reproducing Field Failures for In-house Debugging," in *ICSE*, 2012.

[37] Q. Fu, J. G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *ICDM*, 2009, pp. 149–158.

Appendix: Unresolved Problems

The appendix lists the unresolved problems arisen in this paper. In addition, some basic concepts of the reduction theory are given. The theory is presented in a more general way than in the main content of this paper. That is, the interpretation of alphabet symbols is neither events nor log entries.

Let E be an event alphabet; $\tau = [e_1, \ldots, e_n]$ is a event sequence in E, i.e. $\forall i \in [1, n] : e_i \in E$; $\Re = (E^*, \to_R)$ is a reduction system based on R, where R is a set of rules, i.e. $\forall r \in R : r = (\tau \to \sigma)$.

Definition 1 (Maximal Reduction): Given $\Re = (E^*, \to_R)$ and $\tau \in E^*$, the reduction τ^M of τ ($\tau \to_R^* \tau^M$) in $\Re$ is called maximal if $\forall \tau_1 : \tau \to_R^* \tau_1$ implies $|\tau^M| \leqslant |\tau_1|$.

There might be more than one the maximal reductions for a given τ.

Consider three special classes of rules:

$$R_s \subseteq R : r \in R_s \Rightarrow r = (a \leftrightarrow \epsilon)$$
$$R_z \subseteq R : r \in R_z \Rightarrow r = (ab \leftrightarrow b)$$
$$R_c \subseteq R : r \in R_c \Rightarrow r = (ab \leftrightarrow ba)$$

R with a subscript consisting of a combination of letters s, z or c defines which types of rules R_s, R_z or R_c are included in R.

Problem 1: Given $\Re = (E^*, \rightarrow_{R_{zc}})$ and $\tau \in E^*$. Does there exist a maximal reduction τ^M for τ in $\Re$?

We are not only interested in decidability of Problem 1, but also want to get an efficient algorithm computing τ^M.

Problem 2: Given $\Re = (E^*, \rightarrow_{R_c})$ and $\tau = a\sigma b \in E^*$. Does there exist $\tau_1 = \sigma' ab\sigma''$ such that $\tau \rightarrow^*_{R_c} \tau_1$?

For both problems, in case of positive answer, the question of decision procedure complexity will finally appear.