D6.1: FITTEST Logging Approach

Project no. 257574

FITTEST
Future Internet Testing
Specific Targeted Research Project
Information and Communication Technologies

Due date of deliverable: 31th August 2011
Actual submission date: 21st Sept. 2011

Start date of project: 01-09-2010
Duration: 36 months

Organisation name of lead contractor for this deliverable: Utrecht University

Document Manager: Wishnu Prasetya

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU - Public</td>
<td>X</td>
</tr>
<tr>
<td>PP - Restricted to other programme participants</td>
<td></td>
</tr>
<tr>
<td>RE - Restricted to a group specified by the</td>
<td></td>
</tr>
<tr>
<td>consortium</td>
<td></td>
</tr>
<tr>
<td>CO - Confidential, only for members of the</td>
<td></td>
</tr>
<tr>
<td>consortium</td>
<td></td>
</tr>
</tbody>
</table>
FITTEST Consortium

This document is part of a research project partially funded by the ICT Programme of the Commission of the European Communities under FP7 as project number FP7-ICT-2009-5 257574.

**UNIVERSIDAD POLITÉCNICA DE VALENCIA**
Camino de Vera s/n
46026 Valencia, SPAIN
Tel: +34 690 917 971
Fax: +34 963 877 359
Contact person: Tanja E.J. Vos
E-mail: tanja@pros.upv.es

**BERNER & MATTNER SYSTEMTECHNIK GMBH**
Gutenbergstr. 15
10587 Berlin, GERMANY
Tel: +49 30 688385240
Fax: +49 30688385259
Contact person: Joachim Wegener
E-mail: joachim.wegener@berner-mattner.com

**UNIVERSITY COLLEGE LONDON**
Malet Place, London
WC1E 6BT, UK
Tel: +44 20 7679 1067
Contact Person: Mark Harman
E-mail: m.harman@cs.ucl.ac.uk

**IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD**
IBM Haifa Labs, Haifa University Campus
Mount Carmel
Haifa, ISRAEL
Tel. +97 2 48296154
Fax. +97 2 48296114
Contact person: Eitan Farchi
E-mail: farchi@il.ibm.com

**FONDAZIONE BRUNO KESSLER**
Via Sommarive, 18
38123 Povo - Trento, ITALY
Tel. +39-0461-314524
Fax +39-0461-302040
Contact person: Paolo Tonella
E-mail: tonella@fbk.eu

**SOFTEAM**
Avenue Victor Hugo, 21
75016 Paris, FRANCE
Tel. +33 1 30 12 18 57
Fax. +33 1 53 96 84 01
Contact person: Andrey Sadovykh
E-mail: andrey.sadovykh@softeam.fr

**UNIVERSITEIT UTRECHT**
PO Box 80.089
3508TB Utrecht, THE NETHERLANDS
Tel. +31 30 2534090
Fax +31 30 253 2804
Contact person: Wishnu Prasetya
E-mail: wishnu@cs.uu.nl

**SULAKE CORPORATION OY**
Korkeavuorenkatu, 35
00130 Helsinki, FINLAND
Tel. +358 10 656 7000
Fax. +358 10 656 7010
Contact person: Osma Ahvenlampi
E-mail: osma.ahvenlampi@sulake.com

© 2011 FITTEST Consortium
Abstract

To deal with the increased dynamics in future Internet application, FITTEST has chosen for a continuous testing approach. This implies that its main components will depend on feedback collected at run-time to adjust existing models and test-cases, and to infer new ones. Log files are an important source for this feedback. Many applications employ logging in order to provide tracing information about their executions. Indeed, the generated logs provide a lot of information that can be useful in many ways. However, to support the FITTEST approach, logs have to be generated systematically, and in a well defined format. Furthermore, in practice people will prefer to have non-invasive approaches that do not clutter their source code, nor distract their programmers. Unfortunately, existing solutions do not support all these capabilities in one single framework.

This report describes a new logging solution. It is non-invasive, and can log both high level and low level events. High level events are events that can be seen as produced by users, whereas low-level events are events that tell us what happens inside a function execution as part of the target program’s reaction to a high level event. These two classes of events require different logging approaches. To systematically log low level events, we statically inject logging statements into the target program. In contrast, systematic logging of high level events requires a dynamic approach. We also combine the static approach of low level logging with a dynamic approach to make it smarter, namely by allowing a piece of run-time logic to filter and compose the stream of entries generated by low level logging before they are put in the log file.

Our logging solution is flexible. It allows us to separately specify which events are to be logged, how to serialize them, and how we want to construct the abstraction of the target program’s actual state. The solution allows the logging level to be adjusted at run-time, e.g. to respond to increasing load. The logs are stored in a compressed format. However, it can be exported to XML; thus enabling post-processing by other tools.

We have implemented the solution as a prototype. It is part of the FITTEST Distributed Framework, which means that it can activated and deactivated remotely, as well as to upload log files to a remote FITTEST server.
Credits

This document is written by: Wishnu Prasetya, Arie Middelkoop, Alexander Elyasov, and Jurriaan Hage from Utrecht University. The ASIC compiler is the work of Arie Middelkoop and from Alexander Elyasov. The Logging framework is the work of Wishnu Prasetya, Alexander Elyasov, and Arie Middelkoop.

The Automation Framework is the work of Arthur Baars and Urko Molina from the Universidad Politecnica de Valencia; their work belongs to WP9.

Some texts here are based on the work by Alex Sturala from Utrecht University.

We thank all these people for their contribution.
# Contents

1 Introduction .......................................................... 8

2 State of The Art ........................................................ 10
   2.1 Introduction .................................................................. 10
   2.2 Logging Approach Overview ........................................... 10
      2.2.1 Type of Loggers ..................................................... 10
      2.2.2 Log Formats ......................................................... 11
   2.3 Applications of Logs .................................................... 11
      2.3.1 Logging Domains .................................................... 11
      2.3.2 Application in Testing ............................................ 12
   2.4 Alternative Approaches ................................................ 13
   2.5 Conclusion ................................................................ 14

3 Global Architecture of FITTEST Logging Framework ............... 15

4 Logging Application Events .............................................. 17
   4.1 High Level Logger, Pattern 1 .......................................... 17
   4.2 Making it non-Invasive, Pattern 2 .................................... 19
   4.3 Separating the Monitoring Infrastructure, Pattern 3 .......... 20

5 Attribute Grammars ....................................................... 21
   5.0.1 Syntax of Context-Free Grammars ............................... 21
   5.1 Syntax of Attribute Grammars ....................................... 22
   5.2 Dependency Graphs ..................................................... 24
   5.3 Tree-Walking Automaton .............................................. 24
   5.4 Demand-driven Attribute Evaluation ............................... 25
   5.5 Higher-Order Children ................................................ 25

6 Bytecode Instrumentation ................................................ 26
   6.1 Motivation ................................................................ 26
   6.2 Flash Structure .......................................................... 26
   6.3 Asic and Asil Overview ................................................. 29
   6.4 Parameter Analysis ..................................................... 31

7 Deep Logging ............................................................... 33
   7.1 Deep Events ............................................................. 33
   7.2 Instrumentation .......................................................... 34
   7.3 Automaton ............................................................... 35
   7.4 Automaton Implementation .......................................... 36

8 Serialization ................................................................ 38

9 Log Reduction ................................................................ 40
   9.1 Introduction .............................................................. 40
   9.2 Log Reduction Problem .............................................. 40
   9.3 Rewriting ................................................................. 41
   9.4 Learning rewrite patterns ............................................. 43
   9.5 More complex rewrite rules ......................................... 44

10 Conclusion and Future Work ............................................ 45
   10.1 Future Work ............................................................. 45

© 2011 FITTEST Consortium
A FITTEST Log Formats

A.1 Raw Log File .................................................. 48
    A.1.1 Compression ........................................... 49

A.2 Specialized Raw Format ......................................... 49
    A.2.1 Object Format ....................................... 49
    A.2.2 Event Format ....................................... 51

A.3 XML Log File .................................................. 53
List of Figures

1.1 Global architecture of the FITTEST tool set ............................................. 8
2.1 Logs Application Areas ........................................................................... 12
3.1 To enable logging, the application (APP) is first instrumented. .............. 15
3.2 Components of the logger. ...................................................................... 16
3.3 Logs are sent to and processed at FITTEST ITE. .................................. 16
4.1 A design pattern for application-events logging .................................... 18
4.2 Non-invasive application-events logging ................................................. 19
4.3 Implementation with Automation Framework ....................................... 20
5.1 Language to describe context-free grammars ....................................... 22
5.2 Minimalistic language for AGs. ............................................................... 24
5.3 Syntax of vertices in a PDG. ................................................................. 24
6.1 Overview of the Flash AST. ..................................................................... 27
6.2 Overview of the structure of ABC ASTs. .............................................. 28
6.3 The structure of ABC names. ................................................................. 29
6.4 The structure of traits. ......................................................................... 29
6.5 Example of control flow between blocks in the AST. ............................ 30
6.6 Definition of the argument analysis. ...................................................... 32
7.1 Sketch of deep logging. .......................................................................... 34
7.2 Instrumenter of the GcdRec function. ..................................................... 35
7.3 Example of a logging automaton. .......................................................... 35
7.4 Interfaces of the logging automaton. ...................................................... 37
A.1 An example of how a high-level event (clicking on a button) is serialized. 52
A.2 A log in the XML format. ...................................................................... 54
1. Introduction

Many applications employ logging in order to provide tracing information about their executions. The generated logs provide a lot of information that can be useful in many ways, e.g. for producing usage statistics, to be analyzed to help us diagnosing errors, or even to directly detect errors [Andrews, 1998b, Andrews and Zhang, 2000b]. Code coverage tools and profilers can use logs to determine which part of the code is executed and what the resource consumption is during an execution. Some also try to use them to infer malicious activities and security-policy violations [Kowalski and Beheshti, 2008].

Within the FITTEST project, logging is a necessity. FITTEST has chosen a continuous testing approach. The main components depend on feedback collected after the deployment of an application in order to adjust models and test-cases, or to infer new ones. Log files are an important source of feedback. Figure 1.1 shows the global architecture of our FITTEST tools set; we can see that the models inference, oracle inference, and diagnosis modules rely on logs as their data source.

Aside from logging, there is one other method to gather run time information, namely monitoring. With monitoring we run the analysis during the execution of the application. We can get the results instantaneously, but doing complex analysis live obviously degrades the application’s performance. It also only works on analyses that do not require a long term observation. On the other hand, with logging we perform the analysis offline and thus have the reverse of the advantages and disadvantages. With the logging framework that we build we can actually do both; at the moment we do not yet explore how to exploit this ability.

An application can register a wide range of events into a log. This can be anything from logging notable activities, irregular events, or logging unexpected errors. We categorize ‘events’ into two types. The first type is that of application events, also called high level events. These events are directly triggered by a user, e.g. when the user fills an input field or clicks on a button. Logging these events help us in understanding how a user used the application and how she has affected it. The second type is that of events at the lower level, e.g. when some functions are called or exited, when some exceptions are thrown, or when some branches are passed. Logging these lower level events give us deeper insight on what happens during an execution, but it is more expensive and hence should be done selectively. We call the logging of lower level events deep logging. Our logging framework can log both types of events, but as we explain later, the approaches have commonalities but also differences.

In practice logging is often done in an ad-hoc fashion. Implementations do not define an interoperable format. Since we put a lot of attention to interoperability in the FITTEST project, our logs must follow a strict syntax (Appendix A). Application-specific extensions to this format are still possible by tagging the contents with meta annotations, but they will not be interpreted. To log an event, we basically call a log function, passing to it relevant information about the event. The log function decides where to write the information and takes care the formatting. For example, below is how we can log a call like o.f(args) from the function callee:
We provide a set of such logging functions. We say that a call to a log function is a log statement (it is a statement that produces log entries). Programmers can insert them manually. However, our case study partners require a non-invasive approach. That is, their programmers should not be bothered by thinking about and writing any logging statement. So, in our approach we rely heavily on automatic injection of logging statements. This can be done in a systematic way and in large scale, without cluttering the source code. The logging functions are implemented in an OO language (Action Script), which means that a complex logging logic can be modularly injected simply by overriding the used logging functions (functions such as logFunCallEntry above).
2. State of The Art

2.1 Introduction

Starting from the mainframes’ era and then with the development of the Internet, logs have been used as the main tool for collecting information about system behavior. Different groups of people involved in the software development process, e.g. system administrators, developers, testers or even marketers, all want to know what has been going on with the system over its life time.

Considering all these groups with the potential purposes of using logs:

1. developers: debugging (to provide execution traces and actual failure), profiling;
2. testers: failure detection (to follow the program logic without having to run it);
3. performance analyst: profiling (to identify bottlenecks and increase performance);
4. support engineers: failure detection, to analyze what users did;
5. web masters and system administrators: configuring or tuning the system;
6. marketers: collecting different statistics.

Their purposes determine the systems’ behavior aspects which are to be reflected in the logs. Often, the same information can be expressed in different formats. The used format is often determined by the choice of automatic analysis tools. And of course, all these tools are developed to serve a concrete user group and their needs. People may also use different terminologies, e.g. developers use the term tracing instead of logging, when they are dealing with debugging issues.

Guided by the need to identify failures or to show their absence, which is almost impossible without involving formal methods instead of the testing, testing teams are using logs. It is such an application of logs which this report is mainly about. Below we will discuss the state of the arts in the field of logging and log-based testing.

2.2 Logging Approach Overview

From the most general perspective a log is just a plain text file, where every line record represents either abstraction of an event in the system, or the system’s internal state at a concrete moment, or both. Usually all records are tagged with time stamps and have types from a list of predefined types for a given application.

There exists two conceptually different categories of events: system-level events and application-level events. Such events as method calls, object creation, exception throwing and variable assignments are related to the first category. Whereas the second category includes high-level events, which are involved in the definition of the application business logic. Examples of such events can be buying items, user creation and the events work-flow associated with them as well as GUI interaction events.

2.2.1 Type of Loggers

There are a number ways to let an application to actually log its events. The easiest solution is to use a print-like function at the places where we want to log. The function can only write the text version of its arguments into the log file. Despite its simplicity and low-levelness, print proved to be a very useful technique in debugging and underlies many logging frameworks. Loggers based on this simple print approach are called in-code loggers. They are typically easy to understand and use. But they are suffering from many disadvantages, for instance, it requires lots of duplications and overwork when we need to log multiple function calls. This approach puts too much additional work on the developers, and therefore can be error prone because of the human factor.

Another well known solution is based on the idea of aspect-oriented programming (AOP). Logging statements are written as a separate concern, as an aspect, which are then weaved into the application’s code. Loggers built like this are called aspect-oriented loggers. The separation of the logging concern makes this solution modular,
and avoids multiple insertions of logging statements. Such an approach is however often hard to implement and is not easy to use. This may be the reason why it is not widely used. In addition, the number of operations supported by the join-point model of the used AOP framework may be limited.

More generally, the aspect-oriented approach is an injection approach, where we a set of separately defined logging statements are injected into the application’s code. Within this class we can distinguish between source code and byte-code injection approach. Loggers built with the latter approach are called byte-code level loggers. Binary instrumentation is used to inject log statements to an already compiled application. We can distinguish between static and dynamic injection. The first means that the injection is done at the compile time, before the application is deployed. The second means that the injection is decided and done while the application is running.

In all approaches, it is actually not easy to provide a guarantee that the added logging statements do not change the intended behavior of the original program. It may seem obvious that a print statement will not change the original behavior, but note that e.g. it changes the timing behavior. It may also throw an exception.

2.2.2 Log Formats

The choice of the output format of the logs may facilitate or impede the subsequent analysis phase. It should not be too verbose and while still containing enough information for the analysis. Here are some examples of existing log formats, which are commonly used in real systems:

- The Syslog was developed as a part of the Sendmail project. It soon became the standard logging solution [Gerhards, 2009] on Unix-like systems and other operating systems. It allows separation of the software that generates messages from the system that stores them, and the software that analyzes them. It also provides a feature to notify administrators of problems or performance. Syslog log entries represent information about activities of different system components (demons, kernel, ftp, user and so on) with assigned priority/levels (Alert, Critical, Warning, Error, Message and so on).

- The Common Log Format (CLF) is the standard text format adapted by many web servers for the purpose of logging. Each entry contains information about the user, host, date, status and so on.

- The Extended Log Format (ELF) was suggested as an alternative for CLF, extending it and providing more flexibility.

2.3 Applications of Logs

2.3.1 Application Areas

There are several areas of applications where logs are used widely: data bases, operating systems, web applications and servers, distributed systems, and supercomputing.

- Logs produced by a database server are called transaction logs or database logs. Such a log represents a history of updates done to the database. The log can be used to recover the database state in the case of crashes or hardware failures, thus guaranteeing transactions reliability. These logs are binary (not human-readable).

- Logs produced by operating systems are called system log files. They typically record device changes, device operations, and security relevant events. These can be processed by other tools for various purposes. E.g. Windows Event Viewer [Schuster, 2007] and Unix System Activity Reporter use them for performance monitoring and debugging.

- Web servers produce the most widespread type of logs, called web-logs. They can store information about a server and its clients. Typically web logs are used to get an overview of requested pages history, including client IP address, request date/time, HTTP response code and so on. Lots of useful statistical parameters might be collected from such logs and used by system administrators, performance engineers and even marketers.

- Distributed systems are usually complicated due their synchronization and nondeterminism issues. Logging has been applied to facilitate debugging. A framework proposed in [Boroday et al., 2003] applies instrumentation, and uses communicating automata for modeling the generated event traces.

- In the field of supercomputing, systems are often a complex composition of different software and hardware. A good technique for failure prediction is highly desirable. Logs can be a good source of information to do it. The work by [Liang et al., 2006] studies BlueGene/L execution logs to infer some laws on system observations; these are exploited to do failure prediction. Currently, the massive size, complexity, and lack of standard formats in supercomputer’s logs make them difficult to analyze. The work done by [Jain et al., 2009] proposes a solution by using textual clustering to build the syntactic structures of log messages and then classify messages into semantic groups, taking advantage of online clustering algorithms.
2.3.2 Application in Testing

Below we summarize testing-related applications of logging.

Failure Analysis and Detection

In [Andrews, 1998a, Andrews and Na, 1998] Andrews gave a log analyzer specification language (LFAL), which is a formal way to represent a set of state machines whose transitions are labelled by entries from a log file. So, such a machine specifies how to analyze the file, and is used to decide whether the log is accepted or to reveal an error. Also with the framework two case studies were included: unit testing and system testing. Later on [Andrews and Zhang, 2000a, Yantzi and Andrews, 2007, Tu et al., 2009] the approach was extended by requirements clarification and random test-cases generation. In [Andrews, 1998a, Andrews and Na, 1998] Andrews also gave an extended review on log-based analyses and introduced a set of terminologies for the field.

Even if we know that a log contains evidence of a failure at some point, it can still be very hard for the testing team to identify the cause of the failure, especially if the log contains a lot of events. This can happen when an application has been running for quite some time when the failure happens. This problem was studied in [Cotroneo et al., 2007]. The solution proposed is based on comparing the log that exhibits the failure and the application’s behavior model inferred from previous log files. Further work on the same approach can be found in [Mariani et al., 2008, Mariani and Pastore, 2008].

Requirements Monitoring

Requirements monitoring is another research direction where analysis of log files was considered useful. Such an approach was first introduced by Feather [Feather, 1998, Feather and Smith, 1999]. As an analysis engine he exploited databases. The analysis process consisted of the following steps: prepare a database to able to store logging information, load the information in the database, convert requirements to DB queries, and then interpret the results. This approach and Andrews’ work on LFA [Andrews, 1998a] were mentioned in [Baresi and Young, 2001] as prospective future approaches to the oracle problem; the approaches are general in the sense that they require neither precomputed input/output pairs nor a previous version of the system under test.

Further development in requirements monitoring was done by Robinson in his work [Robinson, 2002], which was also based on the application of logs but in a completely different way. The proposed framework can be used for continuous requirements analysis during the runtime. It combines assertions checking and model checking to inform the monitor. The target program is instrumented in order to produce a stream of events, which can be translated to a sequence of method calls and used to generate a state-based model. The resulting model is checked for requirements violations. When one is found, the corresponding path is reported.

Ducasse in [Ducasse et al., 2006] have proposed the TestLog framework, which uses execution traces as a basis for expressing test-cases. The traces are represented in the form of Prolog logic facts and a test-case is expressed in the form of a Prolog logic query. Thereby it eliminates the need to programmatically bring the system to a particular state, and provides the test-writer an abstract mechanism to query the traces.

Security Testing

For security, in particular Internet security, we may want to continuously monitor and log an appilcation’s activities. This provides a full history of the application usage. The logs can be subjected to security analysis. E.g. Kowalski [Kowalski and Beheshti, 2006] did this to detect intrusion. To make the analysis more practical he considered the intersection of the web server’s firewalls’ log files and the server’s access logs. The approach can identify potential attackers’ IP addresses.

Invariant/Specification Inference

Log files have also been used for inferring program invariants. Daikon [Ernst et al., 2001] is a tool for discovering invariants from execution traces. In order to do this, the program is instrumented to trace relevant variables and then it is run through a (large) set of test-cases. Candidates invariants (in terms of both instrumented and derived variables) are systematically built, pruned, and checked against the traced values. Those which passed the tests, and with a sufficient number of positive witnesses, are reported to the programmer.

Dynamic approaches like Daikon can potentially be strengthened by exploiting information provided by static analysis, e.g. as in [Nimmer and Ernst, 2002].

Recent work [Barringer et al., 2010] in the field of runtime verification has shown that formal analysis of log files can be especially useful for real-time critical embedded applications, where memory and CPU resources are
limited. Barringer uses temporal logic for specifying properties of logs. The specification language supports data-parameterization, which is essential for analyzing logs (events typically carry data). Specified properties, called patterns, are translated to data-parameterized automata, expressed in a subset of the textual RULER language. Users can mix patterns with more expressive automata in a specification. Parameterized automata are visualized using the GRAPHVIZ tool. The system additionally offers preliminary support for automatically learning specifications from example logs. The log analysis framework, LOGSCOPE, is developed to support engineers testing the flight software for NASA’s next Mars rover mission, the Mars Science Laboratory (MSL), developed at the Jet Propulsion Laboratory (JPL). In a sense the work represents an instance of the often sought marriage between theory and practice in formal methods.

2.4 Alternative Approaches

Aside from logging, there is basically only one other method used to analyze program execution and that is dynamic observation of a running system. Dynamic observation is a common approach applied, for example, during software development. Developers use tools that enable them to run, pause, resume and trace execution step by step and examine states of the execution instantly online. The main advantage over a log is the ability to examine instantly any variable and object instance at each point of the execution. Dynamic observation of running system is however not always possible because we are not always supported with the necessary tools. Often it is not beneficial to monitor running system online and retrieve immediate results, either because we want to defer analysis to avoid performance degradation, or to perform analysis repeatedly. Dynamic observation is suitable only in some cases and does not work for long-term observation therefore often generating a log is more convenient or it is the only option we have.

2.5 Conclusion

Thirteen years have passed since the pioneering Andrews’s research in the field of Log-Based Analysis (LBA). State of the art in LBA at the beginning of 2000’s was reflected by Valdman in [Valdman, 2001]. How much have the situation changed since that time? Various researches have suggested many ways to exploit information in log files. Generally people agree that the potential is vast. However people are still searching and trying to mould all those ideas into a framework with the right balance of being sufficiently powerful, universal, and practical. We have not seen such a solution yet.
3. Global Architecture of FITTEST
Logging Framework

To enable logging we first *instrument* the application. The purpose of this is to identify *logging points*. These are the places in the application where we want to log some information; so some logging statements must be inserted there. Depending on the context of a given logging point we also need to decide what kind of logging statement is making sense to be placed there. Our instrumentation will either directly inject the log statement at that point, or else provide a way to later on execute the right logging statement at the right moment—we will explain this later.

An Internet application typically consists of a server and clients. Server-side logging is more common. However, modern clients have become rich and complex. Client-side logging is thus useful, but available (client-side) logging facilities are unfortunately rather inferior. As it is now, our logging framework is mainly tailored for client-side logging. The clients are written in the popular language ActionScript, which run on the Flash player, which in turn runs on users’ browsers.

We employ two kinds of instrumentation. The first one is static instrumentation. It statically parses the application, and directly injects logging statements. It generates a new application; it is this one which has to be deployed for production instead the original one. Figure 3.1 illustrates this. This work is done by our tool ASIC (ActionScript Instrumentation Compiler). ASIC does bytecode instrumentation, as opposed to source code instrumentation. This will be discussed in more details in Chapter 6.

![Figure 3.1: To enable logging, the application (APP) is first instrumented.](image)

Static byte-code instrumentation works well for facilitating the logging of low level events, but it does not work well for the logging of application events. Quite often the GUI of an application is dynamically constructed, which means that it is in principle not possible to predict statically which events should be considered as application events. So, we do the needed instrumentation dynamically. This means that at the run-time we scan the application GUI tree, and decide for each GUI component in the tree whether we want to log the component, and if so what kind of GUI events on the component should be logged. This work is done by another FITTEST component called Automation Framework module (see Figure 3.1) from WP9.

Figure 3.2 shows the top-level components of the logger. The logger itself only contains application-specific configuration code, such as setting which components it wants to use, the default logging level, and so on. The logging logic itself resides in the components. The role of Automation Framework has been explained. LoggingLib provides a set of logging functions, and the underlying capability to write and format log entries to some destination.

The Automation Framework can be thought to be parameterized by a set of *automation delegates*. They specify which application events are to be logged. The logger can be thought to be parameterized by a set of *serialization delegates*. They specify how objects of various classes should be represented in the log. E.g. they control which fields should be logged. Effectively, these two sets specify which (high level) events to log, and which part of them should be logged.

The *abstraction function* in Figure 3.2 is the serialization delegate for the application itself. So, it constructs an abstraction of the application’s actual concrete state, and when the logger need to log the application’s state, then this abstraction is used instead. Note that the logger does not construct this abstraction function by itself. Someone or something else must provide it.
Finally, Figure 3.3 shows where the logger sends the log information to. The FITTEST Integrated Testing Environment (ITE) is the hub of all FITTEST tools. It has the ability to control those other tools. When a user agrees that she wants to be logged, she launches the FITTEST agent on her machine. The logger will recognize the agent, but more importantly it can only write to this agent. So, without the agent, and thus the user’s consent, no information will flow out. The agent is also checked for its certificate to prevent faking. The logger writes log entries to the FITTEST agent, and when ITE requests it the agent will send the collected information to be saved at the ITE side.

Logs are saved in a compressed format; but to some extend they are still searchable without having to fully decompress them. This format is not suitable for consumption by other FITTEST tools, such as model inference. The Haslog component can filter compressed log files, and export them to XML. In this format they can be consumed.
4. Logging Application Events

Abstractly we can view an execution of a program as a sequence of events; each may change the state of the program. Some of these events may be interesting to be logged. Furthermore, we can define the concept of events at different levels of granularity. E.g. we can consider each machine instruction as an event; this would be very low level. Or, we can choose function calls to be our events. It also makes sense to log events of different granularity levels.

The highest level of granularity is at the program’s interactions with users (or other external entities that actively use the application), for example when a user fills in a text field, or when she clicks on a button. We call these interactions application events. Logging these events help us in understanding how a user have used the application and how she has affected it, e.g. when an error or something atypical is observed.

From the testing perspective, it would be nice if application events log can reconstructed to replay the original execution. Unfortunately this is in principle not possible if one of these turn out to be a property of our program and logging solution:

1. The target program is non-deterministic.
2. The target program has a persistent state that is too large to be fully serialized in the log.
3. The logging itself is incomplete, e.g. if we do not log all application events.

Definition 4.0.1. The \textit{REPLAY} assumption assumes that none of the above situations are in effect. \hfill $\Box$

Still, sometimes we can circumvent these obstacles, or devise some strategy to approximate the original execution. So, we believe it is still useful to give a logging solution which under the \textit{REPLAY} assumption will indeed enable replay.

In our Habbo case study, the application has a client-server setup. The logging will be done at the client side—the user interacts with the application through the client-side GUI. The client is written in ActionScript, and runs on the Adobe Flash player that in turn runs inside the user’s browser. The client has a rich GUI and is complex. From the logging perspective, this client will take the role of the ’target program’.

ActionScript offers a number of GUI frameworks (mx and spark) which in turn offer a powerful event handling mechanism. Our application events logging solution relies on this mechanism. However the idea behind the solution is generic. If people can duplicate enough aspects of this event handling mechanism on their GUI framework, the solution can also be duplicated. This is in fact is already the case in our Habbo example, since it actually uses a custom GUI framework.

Below we will describe a design pattern that generically describes how our logging solution works, as well as the underlying capabilities that must be present to make it works. Note that to actually apply the pattern in a particular application one needs to provide a concrete implementation of it.

4.1 High Level Logger, Pattern 1

Figure 4.1 shows how we model an application. Elements of this model are explained below. We primarily want to explain the concepts; so we ignore efficiency issue. An implementation should solve the latter.

Display Object
A display object $o$ represents a GUI element that a user can interact with. Examples of display objects are text fields and buttons. The application $A$ maintains a so called display list, represented by $A.disps$. This list contains all alive display objects that currently constitute the application. This list can change at the run-time.

Event Handling
A user interact with a display object $o$ by 'dispatching' an event $e$, possibly with parameters e.g. $x$. Logically we can treat this as sending a message $e(x)$ to $o$. We will treat events as objects of type \textit{Event}. So an event may have fields. The field $\text{args}$ will hold its parameters. So $e(x)$ is actually an object $e$ of type \textit{Event} such that $e.\text{args} = [x]$. 

© 2011 FITTEST Consortium
The application itself is assumed to be a display object as well. So, we can dispatch events to it.

We also categorize events in different types, e.g. Click (to represent clicking on something) or Type (to represent typing on an input field). We will denote the type of e with e.type.

Each display object o has a dictionary o.handlers that maps event-types to a list of functions. When it receives an event e(x) it will then execute all the functions in o.handlers[e.type], passing to them e(x). These are the 'business domain' behavior triggered by the event. These functions are called event handlers. The list o.handlers can change at the run-time.

If o.handlers[type] is either undefined or empty then o will do nothing when an event of that type occurs; so o.handlers also specify which events o will react on.

Abstraction Function
The application is assumed to provide an abstraction function denoted by A.absfun(). When called, it will construct a single object that abstractly represents the state of A at that moment.

Display Objects Monitoring
An automation delegate is the term we use for an object that has an ability to 'monitor' a display object. An automation delegate d will be associated to exactly one display object, which we will denote by d.subject. We do not have to monitor all display objects. But when we do want to monitor one, then we need to somehow attach a delegate to it. To do this, whenever a display object o is added to A, we assume A will also call added(o). This function decides whether or not to monitor o. If o is to be monitored, the function decides what kind of automation delegate it wants to attach to o. It creates it, then attaches it. Furthermore d.register will be called to activate the monitoring. So:

```java
class Application {
    ...
    function added(o:DisplayObject) : void {
        if (o is to be monitored) {
            var d: AutomationDelegate = create one of the right type ;
            d.subject = o ;
            delegates.add(d) ; // attach delegate
            d.register(o) ; // activate monitoring
        }
    }
}
```

An automation delegate d comes with a list d.types of event-types. It specifies the types of events on d.subject that will be monitored. When an event of type T has to be monitored, the function register above adds a new handler h to o. So, when an event e(x) of type T is dispatched to target o, when it arrives at o then h will also be executed. When called this new handler will dispatch a new event r of type RECORD (we assume that this does not clash with existing types), passing to it e(x) and o. So:

```java
class AutomationDelegate {
    var types ; // list of event-types to be monitored
```
The function \texttt{dispatchEvent}(p,e(x)) dispatches the event \(e(x)\), targeting the display object \(p\).

\section*{Logging}

The logger in Figure 4.1 provides a function \texttt{logAppEvent}(u) that will then serialize \(u\) to a log file, and marking it as an application event. When an event \(e(x)\) occurs and we want to log it, we can just pass it to the function \texttt{logAppEvent}. One possibility is to systematically and statically (at the compile time) inject the calls to the log function into the code of the application. But they have to be injected in the right places to make sure that the calls will happen at the right time. Unfortunately, this is very difficult because the application’s display list may change at the run-time. We do not take this approach, but instead we will do it by exploiting the monitoring functionality provided by automation delegates.

Suppose the event \(e(x)\) targets the display object \(o\). If events like this are to be logged, then we must make it so that \(o\) is monitored by an automation delegate. Let \(d\) be this delegate. We must furthermore make it so that \(e\text{.type} \) belongs to the monitored event-types of \(d\) (it occurs in \(d\text{.types}\)). When \(e(x)\) arrives at \(o\), recall that \(d\) will then dispatch an event \(r(e(x), o)\) of type \texttt{RECORD}. To log this automatically we need one more step, which actually must be done earlier.

All we need to do is to add a handler \(g\) to \(A\) to react on \texttt{RECORD} events, and make it so that this handler passes the events to the logger. This is done by the function \texttt{attachLogger()} which should be called once when the application is created:

```java
class Application extends DisplayObject {
  ...
  function attachLogger(L : Logger) : void {
    this.handlers[RECORD].add(function g(recordEvent) { L.log(recordEvent) })
  }
}
```

\subsection*{4.2 Making it non-Invasive, Pattern 2}

The previous solution is invasive, namely because the programmer has to explicitly write the abstraction function \texttt{absfun} and the function \texttt{attachLogger} in the application; these are logging specific functions. This can be solved easily by refactoring; namely by moving these functions from the application to elsewhere. A sensical place to put them is in the logger itself. If we add to the logger a pointer to the application, these functions will still be able to
navigate over it to get to the application. The new model is shown in Figure 4.2. It also seems nice to introduce a separate class to hold the abstraction function, and move the function there.

4.3 Separating the Monitoring Infrastructure, Pattern 3

Monitoring by automation delegates is a separate feature. It can be used for logging, but it can also be used for other purposes, e.g. for enforcing a certain business domain invariant. Still, it is not a feature we can assume to be present in every application. And if it is not present, then it has to be built. In this situation Pattern 2 is still invasive, because it forces the programmer to implement the function `added` and maintain the delegates list inside his application. To avoid this we again refactor it by moving the them away from the application. We will introduce a new class called `Automation` to hold them. The new model is shown in Figure 4.3.

Our actual implementation is an instance of Pattern 3. The class Automation and the automation delegates are provided by the FITTEST Automation Framework made by WP9. The framework also provides a replay functionality (the function `invoke` in the model).

Display Object’s ID and Replay

Even if the events can be fully reconstructed from the log, and we assume `REPLAY`, replay is still not possible. To replay an event `e(x)` on a display object `o` the run time system will need to find `o`, for which it normally needs to have the physical memory pointer to `o`. Whereas pure data can usually be serialized and deserialized, memory pointer can’t.

To get around this we will require that every display object `o` in `A` must have a unique ID, denoted by `o.id`. Recall that we log an event `e(x)` on `o` as `r(e(x), o)` where `r` is an event of type `RECORD`. If we make sure that the serialization of `r` serializes at least these: `e.type`, the parameter `x`, and `o.id`; then we will be able to find `o`. So, replay can be done like this:

```javascript
class AutomationFramework {

    function invoke(targetID, eventType, args) : void {
        for (var o in disp) {
            if (o.id == targetID) {
                var event = new Event(eventType);
                event.args = args;
                dispatchEvent(event, o);
            }
        }
    }...
}
```

![Figure 4.3: Implementation with Automation Framework.](image)
5. Attribute Grammars

An Attribute Grammar (AG) is a context-free grammar extended with attributes and rules with the purpose of specifying the semantics of a language as attributes on nonterminal symbols in the grammar of the language. Given a value for each attribute associated with the nonterminal symbols of a production, the rules specify whether these values are correct. On the other hand, the rules can also be used to compute such attributions for a given AST, which is how we make use of AGs within this project.

Compilation typically involves tree traversing. AGs abstract from tree traversals, and offer composable compiler descriptions and abstraction of common patterns. These properties make AGs convenient for the implementation of a compiler.

We make extensive use of UUAG, our attribute grammar system, for the implementation of the instrumentation compiler. Therefore, we use this chapter to explain AGs and their terminology. We use the term host language to refer to the language in which the algorithm or compiler is written which we generate from the attribute grammar description. As we see later, functions of the host language may appear in grammar descriptions.

5.1 Syntax of Context-Free Grammars

Definition (Context-free grammar). A context-free grammar is a tuple \((V,N,S,P)\) where \(V\) is a set of terminal symbols (the alphabet), \(N\) is a set of nonterminal symbols, \(S\) is the start symbol with \(S \in N\), and \(P\) is a set of productions (defined below). The set \(V\) and \(N\) must be disjoint.

Definition (Production). A (context-free) production \(p = n \rightarrow \bar{m}\) is a rewrite rule with nonterminal symbol \(n \in N\), and a sequence of symbols \(\bar{m} \in V \cup N\). The sequence \(\bar{m}\) may be empty. The application of \(p\) to a sequence of symbols \(\pi \in V \cup N\) constitutes to the rewriting of one occurrence of \(n\) in \(\pi\) to \(\bar{m}\). In production \(p\), \(n\) forms the left-hand side of the production and \(\bar{m}\) the right-hand side.

To summarize, a grammar is a rewriting system where productions specify a rewrite step from a sequence of symbols to sequence of symbols. In a context-free grammar, a production specifies how to rewrite a single nonterminal symbol to a sequence of symbols. The rewriting terminates when only terminal symbols are left: when successive applications of productions to some singleton sequence \(n\) (with \(n \in N\)) results in a sequence of symbols, then this sequence is derived from \(n\).

Definition (Sentence). A string is a sequence of symbols. A sentential form is a string derivable from the start symbol of the grammar. A sentence is a sentential form consisting only of terminal symbols.

Definition (Derivation tree). A derivation tree is a tree \(t\) that represents how a sentence \(\pi\) is derived from a symbol \(m\), and is inductively defined as follows:

- A leaf \(t\) represents either the derivation of the empty string from a nonterminal symbol \(n\) if there exists a production \(p = n \rightarrow \epsilon\), or the trivial derivation of the singleton string \(\pi = v\) from a terminal symbol \(v\). (Only) in the former case, we say that the leaf is associated with the nonterminal \(n\) and the production \(p\). In the latter case, the leaf is only associated with the terminal \(v\).

- If trees \(t_1, \ldots, t_k\) represent the respective derivations of sentences \(\pi_1, \ldots, \pi_k\) from symbols \(m_1, \ldots, m_k\) then the tree \(t\), formed by taking \(t_1, \ldots, t_k\) as the respective children of the root, represents the derivation of the sentence \(\pi_1 \ldots \pi_k\) from symbol \(n\) if there exists a production \(p = n \rightarrow m_1 \ldots m_k\). We say that the root of \(t\) is associated with the nonterminal \(n\) and the production \(p\).

Definition (Syntax tree). A syntax tree (or parse tree) is a derivation tree that is associated with the nonterminal \(n\). This definition purposefully excludes singleton trees denoting a terminal symbol.

Definition (Abstract syntax tree). An abstract syntax tree is the result of applying a projection to some syntax tree. Usually, the resulting tree has less branches (e.g. due to omission of layout) or branches replaced with a more general representation (e.g. desugared).
Definition (Language). The language $L_G$ specified by a grammar $G$ is the set of sentences that can be derived from the start symbol of $G$.

If $L_G$ is a language of algebraic data types, then nonterminals describe type constructors, terminals describe primitive types, and productions describe data constructors. An AST represents a data structure, and the bit sequence in memory can be regarded as the sentence.

Context-free grammar notation. In Figure 5.1 we introduce a language for the description of context-free grammars: i.e. terms in this language can be interpreted as a grammar as defined above. We later extend the language to describe attribute grammars and some AG extensions. The language that we present here is closely related to UUAG [Universiteit Utrecht, 1998], which we used for the actual implementation of the instrumentation compiler, but is more suitable for presentation. We explain some aspects of the notation below.

In the notation, a grammar is the composition of grammars for individual nonterminals. The set of terminals $V$ and nonterminals $N$ are left implicit, and productions $P$ are grouped per nonterminal. We reuse these letters for other purposes, such as $N$ as an identifier for a nonterminal, and $P$ as an identifier for a production.

The grammar is abstract: instead of terminal symbols, only the type of a terminal symbol is given. To stress the difference between terminals and nonterminals, we use a double colon to specify the type of a terminal and a single colon to specify the name of a nonterminal.

Definition (Children). Each symbol in the right-hand side of a production has an explicit name, which will be useful later. We call such named symbols the children of the production, which stresses the correspondence to children in the AST of nodes to which the production is associated.

In the notation, the nonterminal symbol of the left-hand side of a production is implicit, since we only describe the right-hand sides of productions. We give the symbol on the left-hand side of a production the fixed name lhs.

### 5.2 Syntax of Attribute Grammars

Attribute grammars. We now extend the language of Section 5.0.1 to denote context-free grammars with notation for attributes and their associated functions.

Definition (Attribute grammar). An attribute grammar is a tuple $(T,N,S,A,I,O,P,F)$, where the set of terminals $T$, set of nonterminals $N$, and start symbol $S$ are defined as for a context-free grammar. The set $A$ consists of attribute names. The map $I$ associates a set of names $I_n \subseteq A$ with each nonterminal $n$ in $N$, which make up the inherited attributes of $n$. Similarly, the map $O$ associates a set $O_n \subseteq A$ with each nonterminal $n$ in $N$, which makes up the synthesized attributes of $n$. For each $n$, the sets $I_n$ and $O_n$ must be disjoined. The productions $p \in P$ are redefined below. The set $F$ consists of computable functions $f$ which we call semantic functions.

Definition (Production). An attribute-grammar production $p = u \rightarrow v : \tau \cdot X$ consists of annotated nonterminal symbol $u$, annotated symbols $\overline{w}$, rules $\tau$, and a set of symbol names $X$. Production $p$ is associated with nonterminal $u$.

Definition (Annotated symbol). An annotated symbol is either an annotated terminal or nonterminal symbol. An annotated nonterminal symbol $u = x : n \cdot \overline{w}$ is a combination of a distinct symbol name $x \in X$, a nonterminal symbol $n \in N$, and a collection of attribute names $\overline{w} \subseteq A$ so that $a$ is either in $I_n$ or $O_n$. We say that $n$ is associated to $x$. An annotated terminal symbol $x : v$ is a combination of a distinct symbol name $x \in X$ and a symbol $v \in V$.

Definition (Attribute occurrence). A reference to an attribute $x.a$ is a combination of a symbol name $x \in X$ (associated to some nonterminal $n$) with an attribute name $a \in A$ so that either $a \in I_n$ or $a \in O_n$. A reference to a terminal $x$ is a symbol name $x \in X$ which is associated to some terminal $v$. An attribute occurrence $o$ is either a reference to an attribute or a reference to a terminal.
We call an occurrence $x.a$ also an attribute $a$ of $x$. Attribute occurrences can be found in rules, which are defined below.

**Definition (Rule).** A rule $\sigma_1 = f \sigma_2$ of some production $u \to \overline{w} \cdot \overline{r} \cdot X$ consists of a semantic function $f \in F$ and attribute occurrences $\sigma_1$ and $\sigma_2$.

The occurrences $\sigma_1$ represent the attributes defined by the rule, which are synthesized attributes of $u$ or inherited attributes of the children $\overline{w}$. The occurrences $\sigma_2$ represent the attributes used by the rule, which are the inherited attributes of $u$ or synthesized attributes of the children $\overline{r}$. In addition, occurrences in $\sigma_2$ may also refer to terminals.

Due to the restrictions on occurrences it is always clear whether an occurrence references an inherited or a synthesized attribute. In our notation for attribute grammars (further below) we allow the same name to be used for an inherited and a synthesized attribute.

**Decorated trees.** To give a semantics to rules, we consider syntax trees annotated with attributes, and define how attributes of the tree are related to attribute occurrences, which are mentioned in the rules of productions.

**Definition (Attributes of syntax trees).** An *annotated syntax tree* or *semantic tree* is a syntax tree $T$ where in addition subtrees are associated with the smallest set $Q$ defined as follows. Let $t$ be a subtree and $n$ be the nonterminal that is associated to $t$. For each attribute $a$ in $I_a \cup O_n$, let there be a distinct attribute symbol $q_i \in Q$. The set $Q$ represents the attributes of $T$.

The symbols $q$ can be seen as instances of the attributes, or as occurrences of the attributes in the tree. This definition states that many trees may be associated with the same nonterminal yet have different instances of the attributes. Moreover, if $t$ is an annotated syntax tree, then the attributes of each annotated direct subtree of $t$ are a distinct subset of the attributes of $t$.

**Definition (Attribute association).** Given some annotated syntax tree $t$ with associated production $p$, an *attribute association* $\alpha$ is a mapping so that for each attribute occurrence $o \in p$, either $\alpha o = q$ for some $q \in Q$ when $o$ is a reference to an attribute (either of $t$ or of a direct subtree of $t$), or $\alpha o = v$ when $o$ is a reference to a terminal child $v$ of $t$.

For each (node of an) annotated syntax tree, there exists such an attribute association. We leave open how this straightforward connection between attribute occurrences and attributes of the syntax tree is constructed.

**Definition (Valuation).** A *valuation* $M$ is a mapping that associates with each $q \in Q$ and each $v \in V$ a value in the host language, which is denoted as $M q$ or $M v$. Furthermore, $M v = \lfloor v \rfloor$ for $v \in V$ where $\lfloor v \rfloor$ is some encoding of $v$ as a value in the host language. These values are called *decorations*.

**Definition (Attributed syntax tree).** An *attributed (or decorated) syntax tree* is an annotated syntax tree combined with a valuation $M$.

A rule $\sigma_1 = f \sigma_2$ encodes the condition $M (\alpha o_1) = f M (\alpha o_2)$ for each node the rule is associated with. Alternatively, we may say that $f$ functionally defines occurrences $o_1$ in terms of occurrences $o_2$, and thus that inherited attributes of children are defined by the parent, whereas synthesized attributes of the children are defined by the children and may be used by the parent.

**Definition (Correctly attributed syntax tree).** A syntax tree is correctly attributed when the conditions imposed by the rules are satisfied.

**Notation for attribute grammars.** The above definitions introduce concepts that underly attribute grammar languages and implementations. Figure 5.2 gives a minimalistic language for the description of AGs, which is an extension of the language for context-free grammars in Figure 5.1. We explain some of its aspects below.

The notation in Figure 5.2 consists of a collection of nonterminal declarations $g$, attribute declarations $I$, and semantics blocks $S$. Attributes are declared separately for each nonterminal and have a type associated with them. The right-hand side of a rule is an expression $f \lfloor \sigma \rfloor$ in some formal language $H$ with embedded references to attributes at identifier positions of $H$ via attribute occurrences $\sigma$. The left-hand side of a rule is also an expression, but limited to patterns such as tuples. The use of expressions is slightly more flexible than just a function symbol.

For attribute occurrences $t.x.y$, we take the following notational conventions. The attribute kind $t$ distinguishes inherited and synthesized attributes. This kind is always clear from the context thus we usually leave it unspecified in examples, unless we want to stress the difference. To refer to an inherited attribute $y$ of a symbol named $x$, we use $\text{inh}$.x.y or simply $x.y$. To refer to a synthesized attribute $y$ of a symbol named $x$, we use $\text{syn}$.x.y or simply $x.y$. To refer to a terminal named $x$ we use the attribute occurrence $\text{loc}$.x.self or simply x.self.
A rule

Graphs of a production.

We also use the notation

Local attributes.

Definition (Tree dependency graph)

child on inherited attributes of the child.

and then add the graph for each child of the root, which describe the dependencies of synthesized attributes of the

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Graph of a tree.

These graphs can be projected on each node of a tree and then combined to form a dependency

Figure 5.2: Minimalistic language for AGs.

Figure 5.3: Syntax of vertices in a PDG.

Local attributes. We also use the notation loc.loc.x to refer to a local attribute with the name x, which is an attribute defined by a rule of a production and is only in scope of that production. The name must be distinct from the name of a terminal symbol. Local attributes are typically used to represent common subexpressions.

5.3 Dependency Graphs

To describe how the values of attributes are actually computed, we consider the data dependencies induced by rules. The trees that we consider in this section are derivation trees generated by some AG \((T, N, S, A, I, O, P, F)\).

Graphs of a production. A rule \(\sigma_i = f \sigma_o\) represents a data dependency of occurrences \(\sigma_i\) on occurrences \(\sigma_o\), or equivalently, a flow of data from \(\sigma_o\) via \(f\) into \(\sigma_i\). These dependencies form a graph.

Definition (Production dependency graph). A Production Dependency Graph (PDG) is a directed graph \((V, E)\) associated with some production \(p\). There is a one-on-one mapping between vertices \(d \in V\) (Figure 5.3), and the nonterminal children of \(p\), the rules of \(p\) and attribute occurrences in rules of \(p\). The edges \(E\) consists of:

- For each vertex \(syn.x.y\) an edge to a vertex \(child.x\).
- For each rule \([\sigma_i] = f [\sigma_o]\) (represented as vertex \(rule.r\)) an edge from \(rule.r\) to vertex \(o\) for each \(o \in \sigma_o\), and an edge from vertex \(o\) to \(rule.r\) for each \(o \in \sigma_i\).

Similarly, a production data-flow graph is a production dependency graph with the edges reversed.

Graph of a tree. These graphs can be projected on each node of a tree and then combined to form a dependency graph for a tree, or a data-flow graph for a tree. The general idea is that we take the PDG of the root of the tree and then add the graph for each child of the root, which describe the dependencies of synthesized attributes of the child on inherited attributes of the child.

Definition (Tree dependency graph). For some annotated syntax tree \(t\) with associated production \(p\), attribute association \(\alpha\), and annotated subtrees \(t_1, \ldots, t_k\) with corresponding nonterminal children \(c_1, \ldots, c_k\) of \(p\), the tree dependency graph is inductively defined as the union of the PDGs of \(t_1, \ldots, t_k\) and the instantiation of the PDG of \(p\) by transforming (with preservation of edges) rule and child vertices to fresh vertices and each occurrence vertex \(o\) to vertex \(\alpha.o\). Similarly, a tree data-flow graph is a tree dependency graph with the edges reversed.

If the tree dependency graph for a given tree is acyclic, the graph specifies in what partial order the rules are to be applied in order to compute attributes.
5.4 Tree-Walking Automaton

Tree-walking automata arose from tree language theory and were introduced by Aho and Ullman [1969]. A tree-walking automaton (TWA) is a device that walks over a tree in a contiguous manner and is accompanied by a state machine that describes how the nodes of the tree change their state upon each visit and whether the device goes up to the parent or goes down to one of the children as the next step. Such an automaton can be used to specify the evaluation algorithm for attribute grammars.

**Definition** (Tree-walking automaton). A TWA for some AST is a tuple \((V, Q, I, F, \delta)\), with an alphabet \(V\) of node labels, a finite set of states \(Q\), an initial state \(I \in Q\), a set of final states \(F\), and a transition relation \(\delta \subseteq (Q \times V \times C \times Q)\), where \(C = \{\text{up}, \text{down}_0, \ldots, \text{down}_k\}\) is a set of commands and \(k\) is the maximum branching factor of nodes in the AST. There exists a one-to-one relation between productions and symbols in \(V\) so that each node is labelled with a \(v \in V\) depending on the production associated with the node.

**Acceptance.** Initially each node in the AST is associated with the initial state \(I\). The automaton starts at the root of the tree and stops if no step can be taken anymore. The tree is accepted if the automaton ends with the root having an associated state in \(F\). With each step, the automaton visits a node. If the automaton is at a node with label \(v\) and associated state \(q\), then the automaton chooses a step \(c\) and new state \(q'\) so that \((q, v, c, q') \in \delta\), or the automaton stops if no such step exists. In the former case, the automaton updates the state of the node to \(q'\) and visits the parent if \(c = \text{up}\) or visits child \(i\) if \(c = \text{down}_i\).

**Evaluation of rules.** An actual AG evaluation algorithm does not only traverse the tree but also needs to apply rules to compute attributes. Thus, in an actual implementation, the automaton also applies a subset \(\gamma(v, q)\) of the production’s rules upon visiting a node with label \(v\) and associated state \(q\).

5.5 Demand-driven Attribute Evaluation

**Definition** (Attribute evaluation). Given a tree, attribute evaluation of an AG is a traversal over the tree dependency graph (Section 5.1) so that it is expressible as tree traversals over the tree as described by a tree-walking automaton (Section 5.3).

**Definition** (Demand-driven attribute evaluation). With on-demand attribute evaluation, the dependency graph is traversed based on which attributes are actually needed by a rule.

UUAG supports demand-driven evaluation by mapping an AG description directly to a lazy Haskell program.

5.6 Higher-Order Children

The semantics of a child is a combination of a (partially decorated) tree and an evaluation algorithm that decorates the tree when provided with decorations related to the inherited attributes of the root.

Higher-order children are children with a semantics determined by the value of an attribute. Higher-Order AGs (HOAGs) [Vogt et al., 1989] support higher-order children, a feature that is also supported by UUAG.

**Children defined by rules.** In a conventional AG, the semantics of a child of a production is determined prior to attribute evaluation. In a Higher-Order AG (HOAG), additional children may be declared for a production. Their semantics is the value of an attribute, or as we encode it: the outcome of evaluating a rule:

\[
\text{child } x : N = f[\pi] \quad -- \text{rule that introduces a child } x
\]

The expression \(f[\pi]\) evaluates to the semantics of child \(x\). Such children are known as higher-order children.

**Application.** We use higher-order children to represent bytecode transformations, and common transformation patterns, as we discuss in a later chapter.
6. Bytecode Instrumentation

For a log-based testing approach, we need to keep additional bookkeeping during the program’s execution, which either requires changes to the machine that executes the program, or requires changes to the program. For dynamic internet applications it is difficult to change the machine (e.g. the web browser or the Flash runtime) because of security policies. Hence, we change the program for which we have the option to change the source code, or change the machine code (e.g. the Flash bytecode). Since we consider semi-automatic transformations of possible precompiled Flash applications and libraries, our approach performs instrumentation of the bytecode of Flash programs.

In this chapter, we give an overview of the Flash bytecode and the infrastructure Asil that we developed to instrument this bytecode. Our infrastructure supports bytecode transformations of methods in Flash files, the injection of Flash libraries in Flash files, and the generation of various tables from Flash files.

6.1 Motivation

There exist many technologies for client-side web applications. In the 90s, Java applets posed an alternative to various COM-components to provide the capabilities of stand-alone applications through a web browser. Adobe Flash took over with a collection of features that allowed rich user interfaces with an easy integration with multimedia. In recent years, Microsoft posed Silverlight with .Net as alternative which provides a tighter integration with technology on Windows platforms. Recent advancements in JavaScript and HTML, however, provide an alternative to both Flash and Silverlight and the use of JavaScript and HTML is increasingly gaining popularity.

Motivation for using Flash. Many web applications are still written in Flash, including our use cases, hence our focus on developing infrastructure for the analysis and transformation of Flash programs. However, it can be expected that in the near future the focus shifts to JavaScript. Our infrastructure is based on the bytecode structure of Flash programs, which is executed by a virtual machine for dynamic web languages. We expect that such a machine is representative for a wider range of platforms: both ActionScript and JavaScript are compatible with some ECMAScript standard.

Motivation for Asil. Many instrumentation frameworks for Flash were based on reflection. With the release of ActionScript version 3, the programming language behind Flash, such approaches were not possible anymore due to the introduction of objects described by classes that cannot be changed at runtime. To instrument Flash programs, a program transformation is necessary for which not many frameworks existed.

Apparat is one of the few frameworks for bytecode transformations of Flash that is still being maintained. It started its development slightly before we started the development of our infrastructure. It provides a similar range of features. A key difference is that we focus on instrumentation and analysis, using attribute grammars (Chapter 5) as powerful abstraction and composition mechanism to compute the necessary context information, whereas Apparat provides a small DSL in Scala for transforming linear sequences of bytecode with limited access to context.

In particular, we use attribute grammars to provide information about where parameters of instructions are defined as context information (Section 6.4) and to discover how a method is statically nested. Some of these analyses are costly on the large ASTs that our example programs have. By using attribute grammars in combination with a pure functional language, our programs are ready to exploit many-core architectures. Since many of attributes are local to methods, the automatic dependency analysis in UUAG can discover that many computations per method can be run in parallel.

6.2 Flash Structure

Our infrastructure for bytecode implementation works on AST representations of flash programs. This section gives an overview of the structure of flash programs. Our infrastructure can handle the whole spectrum of Flash
programs as available today (Flash movies, flash components, and ABC files). However, we give a simplified presentation here.

**Definition** (ABC). ActionScript (version 3) is the programming language used in combination with Flash, which can be compiled to ActionScript bytecode (ABC). ABC may be present in Flash programs.

**Definition** (AVM). The *Actionscript Virtual Machine* (AVM, version 2) is a standardized specification for the interpretation of ABC. The Flash runtime of Adobe implements this specification, as well as several open source initiatives.

**Definition** (Tag). A *tag* is a named component, such as various forms of media and also ActionScript bytecode. With our instrumentation framework we can transform and create ABC tags. Other tags are opaque to our framework.

**Definition** (Flash program). A *Flash program* is an association list of tags. Figure 6.1 sketches the structure of a Flash program.

Each ABC tag contains an ABC module *AbcFile*. Figure 6.2 shows the structure of such modules. In general, a module consists of a constant pool, a class table and a method table. The constant pool provides short integer references to strings, various kinds of numbers, namespaces and names. These references are only visible inside one module. The classes describe describe the structure of objects. Since ActionScript is an object-oriented language, methods may be called on objects. The methods are represented in a separate table along with machine instructions. We focus below on some particular ingredients.

**Definition** (Name). A *name* is a combination of a namespace and an identifier. Some ABC instructions support late binding. A name or namespace with a *Late* reference refers to a name or namespace on the stack. Names are represented as first class objects of type *MultiName* and can be manipulated programmatically. Figure 6.3 shows the relatively complex structure of names.

Values in the AVM have a type. These types are dynamically checked, in particular prior to invoking a method and when coercing a value. Types in ActionScript are represented as names. A name with index 0 in the name pool represents the any type `*`.

**Definition** (Trait). A *trait* is a property of an object. Two of the common traits are fields and methods. A field-trait describes a named value of an object. A method-trait describes a method that can be called on the object, and refers to a method in the method table. Figure 6.4 sketches the structure of traits.
**Definition (Block).** A *block* is a group of instructions with a consecutive static control flow. The last instruction of a block may be a static branching instruction to the beginning of another block or a dynamic branching instruction in the form of method exit or exception throwing. The static control flow represents a control flow graph which is sketched in Figure 6.5.

ABC Instructions consists of various conventional instructions to invoke methods on objects, read and write properties, manipulate the stack, and change the control flow. To instrument the program we need to identify places where to inject code. We can do so by identifying particular blocks and instructions.

**Definition (Virtual instruction).** A *virtual instruction* is a placeholder for a sequence of instructions. The original program is obtained by replacing each virtual instruction with an empty sequence of instructions. If this sequence has a consecutive control flow, the original control flow of the program is not affected.

We initially automatically inject virtual instructions to mark coarse-grained locations where we can inject the instrumentation. In particular, we identify jump targets (labels) and the beginning and end of methods and blocks. Moreover, a range of high-level virtual instructions are available to specify the injection of reads and writes to additional local variables. These instructions can be used to memorize important intermediate results in additional variables. We perform a liveness analysis on these virtual instructions, which we use to ensure that we only inject code that memorizes intermediate results when these results are needed by logging code.

---

**Figure 6.2:** Overview of the structure of ABC ASTs.
A VM2 Virtual Machine. The A VM2 is a virtual machine for object-oriented dynamic web applications. It supports both class-based inheritance and prototype-based inheritance. The use of prototype-based inheritance is discouraged in ActionScript version 3.

A VM2 differentiates itself from a virtual machine for statically typed languages, such as the JVM, by providing various constructs for late binding. Names computed on the fly may be used as identifiers of method calls and field lookups, although this particular feature is not used often. However, A VM2 provides dynamic scoping, and the lookup of identifiers largely takes place though the scoping mechanism. This particular feature complicates abstract interpretations severely: to identify to which declarations an identifier may refer requires an approximation of the dynamic scopes, which by itself requires a full-fledged points-to analysis. Thus, when determining contextual information for instrumentation, we need to carefully choose which analyses are done statically and which analyses at runtime.

6.3 Asic and Asil Overview

Our framework is called Asil, and the tool that uses this framework is called Asic. The Asil framework consists of several extensible components:

**Parsers.** Asil contains parsers for Flash programs (.swf files), Flash components (.swc files) and bytecode libraries (.abc files). The parsers are tested against a regression suite of Mozilla’s Tamarin project, and all components of the Flex framework. We discovered that the specification of the ABC contains a small number of mistakes and omissions.

We noticed that several of our example programs used obfuscation techniques to make transformation more complicated. In AVM, instructions may be of variable length and branching is expressed as byte offset. In obfuscated code, we found unreachable code with branches that do not point to the start of an instruction. Moreover, the unreachable code was type-incorrect and exhibited incorrect stack behavior. After parsing, we eliminate unreachable code to prevent these obfuscations from destroying analysis results. The unreachable code cannot be executed, thus does not need to be instrumented.

Also, we replace branching with byte offsets to branching to explicit label instructions. This way, we can shift instructions around without invalidating branching instructions.

**Pretty printers.** Asil contains binary pretty printers to each of the formats listed above. The pretty printers are robust to changes of the bytecode structure. Branching to labels is automatically converted to branching to the appropriate bytecode offsets. Method declarations administer various resource boundaries such as the maximum number of local variables and stack entries needed by the method to hold intermediate results.

**Symbol tables.** Asil contains various components for the computation of symbol tables. For example, we provide facilities to import and export the structure of classes. For many instrumentations, we need to know which identifiers and types are in the environment, and which methods and fields objects of these types contain.
Among these symbol tables, which we can use in analyses to determine by which classes certain methods are implemented, and how many possible subclasses a given class has.

Another commonly used symbol table is the nesting environment of methods and classes. In AVM2, methods and classes may be nested, which corresponds largely with the static scoping of methods and classes. This information is important in instrumentations to apply certain instrumentations only in the context of certain methods and classes.

Analyses. Internally, we transform the parse trees to the AST of a flash program (Figure 6.1 and Figure 6.2). Most of the analyses work with that representation. Attribute grammars provide us with the infrastructure to process such trees. For abstract interpretations, we further work with the control-flow graph representation of methods as shown in Figure 6.5. We implemented an efficient worklist algorithm to perform fixpoint computations over such graphs.

We consider one example of an analysis in Section 6.4.

Asic is the tool that uses Asil to implement a number of operations, including:

Transform code. To inject instrumentations into bytecode, Asic is compiled together with an injection description which consists of attribute grammar code in combination with an injection DSL. The attribute grammar system and the Haskell compiler statically check the transformation description so that the injected code is type correct. In general, it cannot be guaranteed that the instrumentation does not change the original semantics of the program. Section 7.2 gives an example of the DSL, which is presented by Middelkoop et al. [2011] in more detail.

Generate type environment. The core DSL is untyped, and can inject arbitrary code. On top of this layer, we provide combinators that use the Haskell type system to verify that the injected code is guaranteed to be well-formed and type correct. For this purpose, Asic can generate a Haskell module from a type environment, which contains typed Haskell identifiers to various methods and classes.
**Inject Flash component.** The instrumented program uses functionality from a support library (Chapter 8) to produce a log. This functionality may not be part of the original program, thus in addition to instrumenting the original programs with calls to the support library, we also need to embed the support library into the program. Asic provides an operation to inject code from Flash components in a Flash file as additional ABC Tags. The code can be added before the first ABC tag or after the last ABC tag, depending on which initializers should run first.

**Generate Flash component.** It is often necessary that the support component can call functionality of the original Flash program. However, to compile the support component against the original program, we need the program to masquerade as a Flash component, which is not possible. However, Asic can transform any Flash program into a Flash component. This component does not contain the actual code, and can only be used as a dynamically linked library, which is sufficient for the support library.

### 6.4 Parameter Analysis

In AVM, intermediate values are stored on the stack. For various instrumentations we are interested in saving these intermediate values. For example, prior to a method call, the top segment of the stack consists of the object to call and arguments for each parameter of the method, in that order so that the last argument is on the top of the stack. When a method has more than one argument and we are interested in the first argument, we wish to take this argument from the stack and store it in a local variable so that we can access it later. However, it is only possible to access the top of the stack. To minimize overhead, we want to store the argument in the local variable just after it was pushed on the stack. For that purpose, we implemented an abstract interpretation that approximates the top of the stack prior and after each instruction.

Figure 6.6 gives the definition of the parameter analysis, which is a conventional abstract interpretation on control flow graphs. Our definition does not include the construction of the graph: this is done by the framework.

As abstract stack entry, we collect the labels of the last instructions that may have produced that value on the stack. Note that this is an approximation when multiple paths in the control-flow graph come together. The joinStackDefs function defines how such abstracts are joined.

With each node in the graph, we associate a small processor loc.nodesem. These processors are repeatedly applied by the solve function. This processor gets the approximations from its predecessors in the control flow graph, combines these approximations with mut and then provides this result to each successor in the control flow graph or Nothing if the result did not change with respect to the previous application. The processor function is additionally parameterized with the previous result (if any) and produces as second result the value to use for the next application.
-- type of the stack approximation (the abstract stack)

`type StackDefInfo = [DefInfo]`  -- the elements visible from the top of the stack

`type DefInfo = Label`  -- the instruction labels where this value was produced

`joinStackDefs :: StackDefInfo -> StackDefInfo -> StackDefInfo`

`joinStackDefs [] ys = ys`

`joinStackDefs xs [] = xs`

`joinStackDefs (x:xs) (y:ys) = x ∪ y; joinStackDefs xs ys`

`sem Instruction prod _`  -- for each instruction

-- processor code for each node, applies stack mutator \( f \) to the abstract stack

`loc.nodeSem ::= \((\text{DefInfo}, \text{MaybeDefInfo}) \rightarrow (\text{DefInfo}, \text{DefInfo})\)`

`loc.nodeSem = \( f \) (inps, mbPrev) \rightarrow \{(\text{DefInfo}, \text{DefInfo})\}`

`let inpStack = case inps of`  

`[] \rightarrow []`  -- no input stacks on incoming edges

`[inp] \rightarrow inp`  -- one input stack on incoming edge

`→ foldl1 joinStackDefs inps`  -- multiple input stacks on incoming edges

`outStack = \text{mut} loc.label loc.inputs loc.outputs inpStack`  -- apply the stack transformation

`mbOutStack = \text{case} mbPrev of`  -- check if the output changed

`Just prevStack \mid prevStack ≡ outStack \rightarrow Nothing`  -- output has not changed

`\rightarrow Just outStack`  -- output changed (into outStack)

`in (Just outStack, mbOutStack)`  -- produces the stack for each output edge and the result for the node

-- collects the processors for each node

`attr Instruction syn gathNodes ::= Nodes StackDefInfo StackDefInfo`  -- collects the processors of each node

`sem Instruction prod _`  -- for each instruction

`lhs.gathNodes = singleNode loc.label loc.nodesem`  -- produce processor for current node

-- the input and output behavior for each instruction

`sem Instruction prod Add`  

`loc.inputs = 2`  -- number of arguments consumed by the Add instruction

`loc.outputs = 1`  -- number of results produced by the Add instruction

`sem Instruction prod CallProp`  

`loc.inputs = 1 + loc.argCount`  -- number of arguments consumed by the CallProp instruction

`loc.outputs = 1`  -- number of results produced by the CallProp instruction

-- applies the stack mutation to the input stack

`mut :: Label \rightarrow Int \rightarrow Int \rightarrow StackDefInfo \rightarrow StackDefInfo`

`mut label nInputs nOutputs = (replicate nOutputs (singleton label) ++).drop nInputs`  

-- putting it together (gathEdges and solve provided by framework)

`sem MethodBody prod Body`  

`loc.outcome = solve instrs.gathEdges instrs.gathNodes`  -- gives the resulting abstract stack for each node

---

**Figure 6.6:** Definition of the argument analysis.
7. Deep Logging

With deep logging, we intend to log all the state transitions that the machine goes through during the execution of the program under test with the purpose of later finding common patterns in the logs of erroneous executions. However, the full sequence of state transitions is too verbose to log, hence we log small abstractions of particular subsequences of the states. The programmer determines which subsequences of states, and which portions of these states to log, which thus needs to be specified in some formalism.

**Approach.** To obtain the desired logs, we instrument the program to produce deep events, which are labelled subsets of the state at particular program points. These deep events are consumed by an automaton that outputs a trace of abstractions of these events. Figure 7.1 gives a sketch of this approach. A stream of deep events \( e_1, \ldots, e_k \) is processed by an automaton to output a more abstract trace \( l_1, \ldots, l_n \) with \( n \leq k \). The machine throws some of the events away (e.g. \( e_1, e_4, \) and \( e_6 \)), and glues some events together as a single log entry (e.g. \( e_2 \) and \( e_3 \) as \( l_1 \)).

**Rationale.** Adding logging instructions to a program has an impact on the program’s runtime and code size. Thus, we restrict the granularity of logging to method and property calls instead of each and every instruction. The kind of patterns that we want to log is for example the observation that a method \( m \) is called after calls to methods \( m_1, \ldots, m_k \). As part of such an observation we also wish to log some of the arguments to these methods.

In case of dynamic languages such as ActionScript it is hard to statically recognize such patterns in the bytecode. An abstract interpretation of an ActionScript program to approximate an inter-procedural control-flow graph is made difficult by an abundance of method lookups on the (dynamic) scope stack.

Hence our motivation for splitting up the generation of deep events (which can be identified statically) from the runtime recognition of the patterns and the associated logging.

7.1 Deep Events

**Definition (Machine state).** A *machine state* \( S \) is a map from locations to values which represents a snapshot of the memory of the machine. Some of these locations correspond to the local stack and to registers.

**Definition (Deep event).** Given a set of event labels \( L \), a sequence of field labels \( P \) for each event label, and a machine state \( S \), a *deep event* is a tuple \( (L, S_L) \) with \( L \in L \) and record \( S = \{s_1, \ldots, s_{n_L}\} \) with \( s_i = \pi_1^{-1}(S) \) given some projection functions \( \pi_1, \ldots, \pi_{n_L} \) and the sequence of field labels \( P_L = p_1, \ldots, p_{n_L} \).

The set of event labels \( L \) is chosen by the programmer, and can be seen as the type of the event. The record \( S_L \) represents some of the values that variables of the program have in state \( S \).

**Implementation.** A deep event \( (L, S_L) \) can be represented in ActionScript as on object of class \( L \):

```actionscript
class L {
    var p_0 ;
    ...
    var p_k ;
}
```

There is a one-on-one mapping between properties of such objects and field labels of \( S_L \).

We shall use the gcd-ucase as running example. For this use case, an example of a deep event is a registration of a call to the recursive gcd-function. The event is represented as an object of class \( GcdRecCalled \):

```actionscript
class GcdRecCalled {
    var x ;
    var y ;
}
```

Another name for deep event is *low-level event*. Chapter 8 gives concrete examples of deep events.
7.2 Instrumentation

To yield deep events, we instrument the program to construct deep events at various locations in the program and deliver the events to the automaton for processing. Figure 7.2 shows an instrumenter of the GcdRec function which describes what code to inject at the method entry and (non-exceptional) method exit of GcdRec. We explain some aspects of this example below.

Higher-order child. The higher-order children beginRec and endRec represent injectors. The inherited attribute code contains the injector description, which is an expression in the injection monad. The code is injected just after the virtual BeginBody and EndBody instructions.

Pattern matching. The instrumentation is conditional. The injector beginRec tests if the current method is the recGcd method using a regular expression match against the method name. The identifier ps is bound to an array of references to parameters of the method. With ps • 0 we obtain a reference to the first parameter, and with ps • 1 a reference to the next parameter. When the match failed, the instrumentor aborts, and the remainder of the code is not executed.

Timestamp. We later want to identify which events originated from the same method. The trick that we use is that we obtain some global unique identifier (the timestamp t) at the beginning of the method and use this identifier as parameters to events.

The newTimestamp command declares a new local variable and returns a reference t to it. It also injects code that assigns a unique value from a global counter to this local variable:

```
newTimestamp = do
  t ← declLocal
  t := callStatic next Dispenser \0
  return t
```

Via the successors command, we deliver t to all successors in the control-flow graph under key kTimestamp. When we obtain the reference to t via a fetch kTimestamp we thus know that the current method is recGhc, otherwise the key kTimestamp would not be defined, and the fetch command would cause the instrumentor endRec to abort.

The raiseEvent command then finally injects the code that delivers the event e to the automaton:

```
raiseEvent e = callStatic accept_Automaton (e ⊕ \0)
```

Remarks. The instrumentation has a runtime penalty: the event objects are constructed at runtime and analyzed by the automaton. Which locations to instrument and what values to store in the events is a balance between the
7.3 Automaton

Definition (Logging action). A logging action is a function that takes a string of events as input and returns a string of events.

Definition (Trace). The string of events that is logged.

Definition (Logging automaton). A logging automaton is an extension of an nondeterministic state machine where:

- The symbols represent events and are attributed with the fields of an event.
• The transition relation $\delta$ is additionally parameterized with the symbols recognized so far.

• A logging action is associated with each end state. When the machine enters an end state, the logging action is applied to the recognized input symbols, and the resulting symbols are added to the trace.

Since the machine is nondeterministic, the automaton may enter multiple end states when given a string of symbols.

Figure 7.3 shows an example of a logging automaton, depicted as a graph. The vertices represent the states. The initial state is $A$, the two end states are $D$ and $F$. The end states are labelled with logging actions. The edges describe the transition relation $\delta$, and are labelled with a pattern. The pattern $\varepsilon$ represents an epsilon transition. The pattern $?$ matches any symbol. All other patterns must unify with a symbol which causes identifiers to be bound.

When an identifier is mentioned more than once, such as $i$ in the picture, each occurrence must unify. Thus, the $GcdRecExit$ transition can only be made when its timestamp is equivalent to the timestamp introduced by the first $GcdRecCalled$ symbol.

Identifiers introduced with an existential represent fresh monotonically increasing indices. Thus, $i$ represents monotonically increasing subscripts of variables $x_i$ and $y_i$, each time a $GcdRecCalled$ symbol is recognized. Through indices we can collect lists of events. The any-pattern can be implemented with $\exists i.x_i$, which introduces a sequence $\pi$ of identifiers which contains an element for each matched symbol.

The automaton in Figure 7.3 describes the logging of events that arise from the execution of the $whileGcd$ and $recGcd$ methods.

**Acceptance.** The resulting trace is the concatenation of all traces obtained by feeding the automaton all substrings of the raised events:

**Definition** (Result trace). Given a string of raised events $e_1, \ldots, e_k$ and a logging automaton $A$, the result trace is a string $t_{1,1} + \ldots + t_{k,k}$ where $t_{i,j}$ with $i \leq j$ is the output trace when $A$ accepts the string $e_i, \ldots, e_j$. Otherwise, $t_{i,j}$ is the empty string.

The events in the resulting trace need to be serializable in order to store them in the log (Chapter 8).

### 7.4 Automaton Implementation

The automaton is integrated into the application in order to efficiently inspect the state of the virtual machine, hence the implementation of the automaton is in ActionScript. In the actual implementation, we permit a number of generalizations. Logging actions may be associated with each node instead of only a final node.

Figure 7.4 gives an overview of the logging API. The automaton is a singleton that implements the $Machine$ interface. Any object that is tagged with the $State$ interface can be used to identify a state. An $EnterActor$ can be associated with a state, which is the functionality that is executed when the machine enters that state. Similarly, a $LeaveActor$ can be associated with a state, which captures the behavior of outgoing edges. The actual implementation provides several ways to compose these actors sequentially and in parallel. For example, the two outgoing edges of $A$ in Figure 7.3 are represented as two $ExitActors$ composed in parallel.

The actors take an instance of the machine as parameter, which keeps track of the bound variables that arose during state transitions. A $LeaveActor$ may pattern match against an event and abort the instance if it does not match. If it accepts the event, it can bind new variables, update the current state, and indicate the event as consumed.

**Instances.** The automaton gives a means to determine how long to keep events in memory. Instead of keeping all events in memory and running the automaton on each substring, we instead run the automaton directly upon each raised event.

We keep track of the possible ways we can end up in final states for the events still to come from the current and previous events using instances:

**Definition** (Instance). An instance of some machine $m$ is a tuple $(I, H)$ where $I$ is the start state of the instance and $H$ represents an environment of bound identifiers resulting from state transitions from the start symbol of $m$ to $H$.

An instance represents a path from the start symbol to $I$.

With each raised event we create a new instance of the machine, and we fork (clone) each existing instance for each outgoing transition of their current states. We then run each instance against the corresponding $LeaveActor$ and event, and prune those instances that are aborted (failed state). The following theorem ensures that we can do this pruning:
public interface Machine {
    function addLeaveActor(state : State, listener : LeaveActor) : void;
    function addEnterActor(state : State, listener : EnterActor) : void;
    function raise(evt : Event) : void;
}

public interface State {}

public interface LeaveActor {
    function onLeave(state : State, evt : Event, instance : Instance) : void;
}

public interface EnterActor {
    function onEnter(state : State, txn : Instance) : void;
}

public interface Instance {
    function setAborted() : void;
    function setConsumedEvent() : void;
    function setIgnoredEvent() : void;
    function getIsAborted() : Boolean;
    function getValue(key : Object) : Object;
    function setValue(key : Object, value : Object) : void;
    function getCurrentState() : State;
    function setCurrentState(state : State) : void;
    function setEvtFilter(filter : Filter) : void;
    function setLifetimeFilter(filter : Filter) : void;
    function fork(evt : Event, actor : LeaveActor) : void;
}

public interface Filter {
    function process(evt : Event, txn : Instance) : void;
}

Figure 7.4: Interfaces of the logging automaton.

Theorem 7.4.1 (Doomed string). If for some machine m, a prefix $S'$ of a string $S$ of events ends in a failed state in m, then $S$ ends in a failed state in m.

Filters. A filter may be associated with an instance. If an event does not match the filter, the event bypasses the instance. Filters are a convenient way to abstract from self loops with the purpose of skipping unimportant intermediate events.

Dead Instances. When a program executes an unexpected path through the control flow graph (e.g. through exceptions), there may be instances that will never reach a final state but may take a while to end up in a failed state, and thus accumulate memory.

To garbage collect unsatisfiable instances, we associate lease-filters with instances. For example, a lease filter may abort an instance if it did not accept an event within a fixed number of raised events.

If we want to limit the life-time of an instance to the execution of a particular method, the following mechanism piggy-backs on the garbage collection mechanism. In ActionScript, we may store values in the activation record of a method. The activation record is garbage collected at some point after the method is finished. Now, if we keep a weak reference to a value in the activation record, which does not keep the value live, we may abort those instances that have a weak reference to a garbage collected value. This approach works well together with exceptional control flows.
8. Serialization

Suppose we want to be able to log the state of various objects in an application. Because an object can contain a complex and large structure we do not want to just blindly serialize the whole object. Imagine that we have a variable _delegates that holds a mapping between classes and serialization-delegate functions:

```
var _delegates : Dictionary
```

If C is a class, _delegates[C] gives us a function f that knows how to serialize any instance of C. The logger essentially takes such a map as a parameter (see Figure 3.2), and that is how it decides how to serialize objects. To allow serialization delegates to be written abstractly, we separate from it the underlying formatting functionality. So, a serialization-delegate function has this form:

```
function sdf(o: Object, s: Serializer) : void
```

Such a function only needs to specify which parts of o need to be serialized. The serializer s will do the actual formatting and writing to some destination. For example, the serialization-delegate of a class Person could look like this:

```
function sdfPerson(o: Object, s: Serializer) : void {
    s.beginObject(o, "Person")
    s.storeField("name", o.name)
    s.storeField("age", o.age)
    s.storeField("spouse", o.spouse)
    s.endObject()
}
```

So it says that only the fields name, age, and spouse will be serialized. The serializer s will go after the objects in these fields. It knows how to serialize primitives. If the object on the field is not primitive it will again consult the dictionary _delegates to figure out how to serialize it. It will make a decision if no delegate function can be found in the dictionary. It will also make a decision on how deep in the object structure it will go. And as said, it will do the formatting. So, by using a different serializer we can produce a different format.

The approach as described above is non-invasive. That is, no change is required on the application source code. Serialization delegates can be written and modified separately from this source code. However, a non-invasive serialization can in principle only look into the public part of a given object. To get access to its private part, the serialization function has to be defined as part of the object’s own class; thus it is invasive. We provide an interface:

```
interface Serializable {
    function serializeSelf(serializer : Serializer) : void ;
}
```

Below is how we can write the serialization of a class Employee. The function serializeSelf is a serialization function. As in the previous example it specifies which fields of an Employee are to be serialized. Notice that we serialize information derived from a private field, and therefore this function must be declared in the Employee itself.

```
class Employee implements Serializable {
    public var name : String ;
    private var salary : int ;
    public var boss : Employee ;
    ... // other things
    function serializeSelf(serializer : Serializer) : void {
```

When a serializer gets an object \( x \) to process (this happens in the call \( \text{storeField}(\text{name}, x) \)), it first checks if \( x \) is an instance of \( \text{Serializable} \). If so the its \( \text{serializeSelf} \) will be called. Else the serializer will look for a delegate in the \( \_\text{delegates} \) map as before.
9. Log Reduction

9.1 Introduction

Imagine an application that records user interactions in a log file. Let us assume that interactions of different users can be distinguished, and they are stored in separate log files. Each log file is organized as a sequence of log entries, ordered based on time. Suppose a user uses the application, then at some point the application throws an exception that indicates an error. We assume that this exception is also logged. The user then reports the error, sending along his log as evidence. However, he typically used the application for hours, and thus the log can be quite large. Large long-running logs pose a problem to the maintenance team who first need to figure out which of the interactions done by the user were actually contributing toward the error, before they can fix it. This chapter discusses an approach to reduce (simplify) the log, while preserving its semantics to some extent.

One can indeed come up with ad-hoc reduction strategies and implement them. However, this likely leads to a messy and tangled implementation. So, we will first formulate the problem formally and abstractly, so that we can express our solution at the same level. Then it can be converted to an implementation.

Generally, our reduction approach works by repeatedly replacing fragments of the target log with equivalent but simpler fragments. In other words, we do rewriting. Our idea is to use algebraic properties like idempotence and commutativity as rewrite rules. This in turn requires a procedure to identify which combinations of log entries satisfy those algebraic properties. For the latter we choose an approach which is also log-based. We assume a collection of previous logs is available as ‘training’ data. We use them to infer what kind of log entries satisfy those algebraic properties. Since logs are an abstraction of real executions, any log-based inference will be imprecise. On the other hand, such an approach is also very robust. It does not require complex source code or byte code analysis; and neither does it depend on the language or whatever middleware used to program the application.

As a proof of concept we implemented a prototype. The prototype logs user interactions of a Flash application, collect logs, export them to XML, analyze them for patterns, and apply the reduction approach on them.

9.2 Log Reduction Problem

We treat a log as a sequence of entries, each represents a user interaction with the logged application. From our perspective, not every actual interaction has to be logged — doing so is in practice unfeasible. Of course, the more information is available in the log, the more precise analyses can be.

We use the term event rather than ‘user interaction’ (shorter). A log is modeled as a sequence of entries, each is a pair \((e(p), s)\) where \(e\) the name of a logged event, \(p\) is the parameter given to \(e\), and \(s\) is an abstraction of the application’s actual state, sampled when \(e\) took place. For us it is irrelevant whether \(s\) is sampled e.g. just before \(e\), or just after it. But to fix the idea, let us suppose that the state is sampled after the event happens. We use the term event-sequence to mean a sequence of items of the form \(e(p)\). So, a log always induces an event-sequence. Where it does not lead to ambiguity, we omit the parameter.

We want to do log reduction that preserves the log’s semantics; at least, in the ideal situation. To have a semantics, we define an execution model that pretends that we can re-execute event-sequences in a log and compare the effects of their execution. It does not mean that logs should in reality be directly re-executable. It does mean that our log reduction approach is semantics preserving, modulo this execution model.

Let us first introduce these notations:

1. If \(\Sigma\) is a log, \(\text{evp}(\Sigma)\) denotes the event-sequence it induces, and \(\text{abstates}(\Sigma)\) is the corresponding sequence of abstract states.

2. \([\ ]\) or \(\epsilon\) denotes an empty sequence; \([x_1;\ldots; x_n]\) denotes a sequence of events of length \(n\. If \(\sigma\) and \(\tau\) are sequences, \(\sigma \tau\) denotes their concatenation. To improve readability sometimes we also write it as \(\sigma + + \tau\). \(\text{last}(\sigma)\) denotes the last element of \(\sigma\).

3. If \(\sigma\) is an event-sequence, \(a \rightarrow \sigma\) denotes the execution of \(\sigma\) on \(a\), with \(a\) being the initial state.

4. \(\text{final}(a \rightarrow \sigma)\) denotes the resulting final state (actual state). This final state is unique, implying that our execution model \((\rightarrow)\) is deterministic.
5. \( \log(a \rightarrow \sigma) \) denotes the log that this execution generates. \( \text{abstates}(a \rightarrow \sigma) \) denotes the list of abstract states that would be logged during the execution. Finally, \( \text{abfinal}(a \rightarrow \sigma) \) denotes the last element of \( \text{abstates}(a \rightarrow \sigma) \).

Our execution model is as such that it has these properties:

**Reproducibility (REP):** On that model, logs are fully reproducible. That is, if \( \Sigma \) is the log obtained from the application with the initial state \( a \), then re-executing its event-sequence will yield exactly the same log due to determinacy of the execution model:

\[
\log(a \rightarrow \text{evp}(\Sigma)) = \Sigma.
\]

**ABS** The abstract state is fully determined by the actual state. That is, there is a function \( \text{abs} \) so that for all \( a \) and \( \Sigma \):

\[
\text{abfinal}(a \rightarrow \Sigma) = \text{abs}(\text{final}(a \rightarrow \Sigma)).
\]

**TRANS:** Execution is transitive. That is, if \( \text{final}(a \rightarrow \sigma) = b \), then \( \text{final}(a \rightarrow \sigma \tau) = \text{final}(b \rightarrow \tau) \).

We assume that it is possible to distinguish if a given abstract state indicates an error or not, and that we can tell what kind of error it is. We model this with error-predicates over the abstract state. If \( E \) is such a predicate, we write \( \text{error}_E(s) \) to denote that the abstract state \( s \) satisfies the predicate \( E \), and thus indicates an error of type \( E \).

The log reduction problem can be formally stated as follows.

**Definition 9.2.1.** Let \( \Sigma \) be a log produced from a known initial state \( a \). Let \( (f(p), z) \) be its last entry, indicating an error of type \( E \); so: \( \text{error}_E(z) \). Find a solution \( \Sigma' \) such that:

1. \( \text{last}(\text{evp}(\Sigma')) = f(p) \)
2. \( \text{final}(a \rightarrow \text{evp}(\Sigma)) = \text{final}(a \rightarrow \text{evp}(\Sigma')). \)

\( \Box \)

So, under the \( \rightarrow \) execution model, \( \Sigma' \) produces exactly the same final state as \( \Sigma \), with the additional constraint that \( \Sigma' \) ends up with the same event \( f \). The ABS assumption implies that the same error \( E \) is observed at the end of \( \Sigma' \). The intention is that the solution \( \Sigma' \) is shorter than \( \Sigma \). Note that the definition above only requires us to reproduce the final situation. It is conceivable to require more, but reduction then also becomes more difficult.

### 9.3 Rewriting

Our reduction approach is based on rewriting. For this we need a notion of equivalence. We define semantical equivalence between event-sequences and logs in terms of their final states:

**Definition 9.3.1.**

\[
\sigma \equiv \tau \quad \equiv \quad (\forall a. \text{final}(a \rightarrow \sigma) = \text{final}(a \rightarrow \tau))
\]

\[
\Sigma \equiv \Sigma' \quad \equiv \quad \text{evp}(\Sigma) \equiv \text{evp}(\Sigma')
\]

\( \Box \)

It is straightforward to show that this relation is an equivalence relation (reflexive, symmetric, and transitive). The theorems below are also not difficult to prove. The first one allows us to convert the log reduction problem to the problem of finding an equivalent sequence:

**Theorem 9.3.2.** If \( \Sigma \equiv \Sigma' \) and \( \text{last}(\Sigma) = \text{last}(\Sigma') \) then \( \Sigma' \) is a solution of \( \Sigma \). \( \Box \)

The next one states that replacing a segment of a sequence with an equivalent one yields an overall equivalent sequence. This allows to rewrite a sequence:

**Theorem 9.3.3.** \( \tau \equiv \tau' \) implies \( \sigma \tau \mu \equiv \sigma \tau' \mu \). Similarly, this holds for logs. \( \Box \)

Since \( \Sigma \) is always a solution for itself, rewriting with \( \equiv \) always yields a solution. Below we give a set of \( \equiv \) properties that we use as the 'basic rewrite rules' for reductions. They are applied repeatedly, in the implementation one must of course provide a rewriting strategy that ensures that the rewriting terminates.

**SLE:** An event \( e \) behaves as a 'skip' (it does not change the application state) if:

\[
(\forall p. [e(p)] \equiv [])
\]

Such an action can thus be removed.
ZERO: An event $e$ behaves as zero with respect to $d$ if:

$$\forall p, q. \ [d(p); e(q)] \equiv [e(q)]$$

An example of such a pair is where $d(p)$ sets the user age to some value, and $e$ resets all user properties to their default values. So, any sequence $[d; e]$ can be reduced to just $[e]$

IDP: An event $e$ behaves idempotently if it behaves as zero with respect to itself. So:

$$\forall p, q. \ [e(p); e(q)] \equiv [e(q)]$$

An example of such an event is:

$$e(p)\{\text{App.user.age} := p\}.$$

IGP: An event $e$ ignores its parameter if:

$$\forall p, q. \ [e(p)] \equiv [e(q)]$$

An example of such an event is:

$$e(p)\{\text{App.user.saldo} := +\}. \text{ We cannot remove the event, but we can at least replace its parameters with some constant.}$$

COM: Two events $d$ and $e$ are commutable if:

$$\forall p, q. \ [d(p); e(q)] \equiv [e(q); d(p)]$$

Except for COM, it is quite obvious that the above rules will simplify a given target event-sequence. COM is an interesting property. If $d$ and $e$ are commutable, the definition implies that executing $[d; e]$ and $[e; d]$ on the same initial state results in the same final state, which implies that the effect of these two events must be independent from each other. Although at the first sight it does not seem to reduce the size of the log, in combination with the other rules it can. For example, if $d$ is idempotent, then we may apply the reduction:

$$[d; e; d] \equiv [e; d; d] \equiv [e; d]$$

A more interesting reduction is possible if we also know that the error itself is in a way commutable. Let us first define what it means:

**Definition 9.3.4.** Errors of type $E$ commutes with an event $d$ if:

$$\forall a, e, p, q. \ \text{error}_E(\text{abfina}(a \to [e(q); d(p)])) = \text{error}_E(\text{abfina}(a \to [e(q)]))$$

The above definition implies that $d$ is not the direct cause of $E$. So, if it is observed in the state after $d$, it must have been observed in the state before $d$ as well.

Suppose now we have $[d; c; e]$ as our log, such that error $E$ is observed in the last state. If $d$ and $c$ are commutable, we can rewrite it to $[c; d; e]$. $E$ is also observed in the last state of this new sequence. If $E$ is also commutable with $d$, then just $[c; e]$ will also produce a state satisfying $E$. Effectively, we just removed $d$. However, the way the log reduction problem is formulated does not actually allow this. It requires the sequences to end up in the same state, whereas the above reduction only guarantees that $E$ is reproducible. We expect that in many situations this is acceptable; so, below we give a weaker formulation of the reduction problem that permits the above reduction:

**Definition 9.3.5.** Weaker Reduction Problem

Let $\Sigma$ be a log produced from an initial state $a$, and has $(f(p), z)$ as the final state, such that error$_E(z)$. $\Sigma'$ is a solution if:

1. last(evp($\Sigma'$)) = $f(p)$
2. error$_E$(last(abfina($a \to evp(\Sigma')$)))

The previous example—removing an event that directly precedes an error, if the error commutes with the event—is now generalized to the following theorem. It allows us to remove an event which are further away if it can be bubbled up toward the error:
Theorem 9.3.6. Let $\tau$ be a non-empty sequence. If $d$ commutes with all events in $\tau$ and also with $E$, then:
\[
\text{error}_E(ab\text{final}(a \rightarrow [d(p)] + \tau)) = \text{error}_E(ab\text{final}(a \rightarrow \tau))
\]

For example, consider the sequence $[d,a,b,c,e]$ where error $E$ is observed at the end. If $d$ commutes with all the events that come after it, we can rewrite the sequence to $[a,b,c,e,d]$. This preserves the final state, and therefore $E$ is still be observed at this final state. If $d$ and $E$ commutes, it implies that $[a,b,c,e]$ also produces $E$.

Any solution of the original reduction problem is also a solution of the weaker reduction problem. This means that we can still do rewriting with $\equiv$, and furthermore we can use Theorem 9.3 to, in some situations, remove events that commute with the produced error.

9.4 Learning rewrite patterns

To apply the rewriting rules from the previous sections we first need to know which events behave as a skip, as a zero, are commutable, and so on. This information can be obtained from various sources. E.g. it can be extracted from annotations provided by programmers, or be inferred using some static analysis.

In this section, we describe we to use a log-based analysis to infer algebraic properties of events. We use previously collected logs as data, and from there we infer equivalence properties. Such an approach is highly portable and simple to setup. On the other hand, logs provide only partial information, and therefore the approach is neither sound nor complete.

We call the previously collected logs experience logs. In particular, they must not be confused with the target log that we want to reduce.

Recall that our rewrite rules are expressed in the form $\tau_1 \equiv \tau_2$. This equivalence is in turn expressed in terms of the equality of the actual final states of those sequences. This is problematical, because we cannot actually inspect the actual state of an application. We still can observe its abstract states, because these are logged. Therefore, we propose the following weaker concept. To check if two (actual) states $a$ and $b$ are equivalent we execute various event-sequences of length $k$ on both states. Then we observe the generated abstract states. If they are the same, we consider $a$ and $b$ as observation equivalent:

Definition 9.4.1.
\[
a \overset{k}{\sim} b = (\forall \mu. \#\mu = k \Rightarrow \text{abstates}(a \rightarrow \mu) = \text{abstates}(b \rightarrow \mu))
\]

It is an equivalence relation, but weaker than equality ($a = b$ implies $a \overset{k}{\sim} b$). However, since it is based on abstract states this relation can be determined by analyzing the experience logs.

Now, to check if $\tau_1 \equiv \tau_2$ by definition we have to check that for all $a$, $\text{final}(a \rightarrow \tau_1) = \text{final}(a \rightarrow \tau_2)$. We approximate this by checking:

\[
\text{final}(a \rightarrow \tau_1) \overset{k}{\sim} \text{final}(a \rightarrow \tau_2)
\]

The above property can be checked by looking for occurrences of $z_1 = \tau_1 \mu$ and $z_2 = \tau_2 \mu$ in the experience logs. We call these occurrences witnesses. For every such pair of witnesses, we check that the two $\mu$’s have exactly the same sequence of abstract states. However this is actually too pessimistic. The formula above only requires the final states to be observation equivalent if $\tau_1$ and $\tau_2$ are executed on the same state $a$. Some, or perhaps many, of the witnesses $(z_1, z_2)$ may actually be based on different starting states. We may thus wrongly reject $\tau_1 \equiv \tau_2$ because we also compare mismatched witnesses. Note that even if we do so, we are not being incorrect. We are just being overly discriminative. This can be mitigated by looking for witnesses of the form $z_1 = \sigma \tau_1 \mu$ and $z_2 = \sigma \tau_2 \mu$, where the $\sigma$ in both parts should also produce the same sequences of abstract states. The idea behind this is that imposing the requirement on the same prefix $\sigma$ makes it more likely that the actually states just before $\tau_1$ and $\tau_2$ will be the same. The above leads to the following definition of observation equivalence over log fragments:

Definition 9.4.2. Two event-sequences $\tau_1$ and $\tau_2$ are observation equivalent, denoted by $\tau_1 \overset{n,k}{\sim} \tau_2$, if for all sub-sequences $Z_1, Z_2$ of the experience logs such that:

1. $Z_1 = ST_1 U_1$, for some $S$ and $U_1$ of length $n$ respectively $k$
2. $Z_2 = ST_2 U_2$, for some $U_2$ of length $k$
3. $\text{evp}(T_1) = \tau_1$ and $\text{evp}(T_2) = \tau_2$

© 2011 FITTEST Consortium
4. $\text{evp}(U_1) = \text{evp}(U_2)$

we have $U_1 = U_2$.

Note that because we required $U_1$ and $U_2$ to be equal on their events and parameters, $U_1 = U_2$ effectively only requires that the sequence of the abstract states must be the same. The above definition basically describes how we look for rewrite patterns in the experience logs.

We can also have an indication of the confidence level of the inference by counting the number of witnesses we have (the number of pairs $(Z_1, Z_2)$ that satisfy conditions 1..4 above.

### 9.5 More complex rewrite rules

More reduction can be obtained if we have more rewrite rules. For example the SLE rule only recognizes when a single event behaves like a skip. Imagine events like $\text{openHelpWindow()}$ and $\text{closeHelpWindow()}$. Each does not behave like a skip, but the composite does:

$$\text{openHelpWindow()}; \text{closeHelpWindow()} = []$$

Another example: $a$ and $b$ may not be individually commutable with $c$. But $[a;b]$ together may be commutable with $c$.

Another example of a complex pattern is the following. Imagine an online shop with event $\text{sel}(i)$ selects the item $i$ and puts it in the shopping basket. There is an event $\text{clear()}$ that will clear the basket. We cannot just say that $[\text{sel}(i);\text{clear()}] = []$, because in the state where the basket is already filled, the effect of these two sequences will not be equal. What may still be true is:

$$\text{clear()} + +[\text{sel}(..);...;\text{sel}(..)] + +[\text{clear()}] = [\text{clear()}]$$

We can expect that patterns like these are more expensive to learn. Since we initially do not know which rewrite patterns are 'valid', the straight forward way to discover them is by quantifying over all possible patterns e.g. of maximum length $k$, and then we check them one by one. But the number of possibilities will explode as $k$ increases. So, for complex patterns some prioritizing strategy is needed. This is future work.
10. Conclusion and Future Work

Many applications employ logging in order to provide tracing information about the executions of such applications. The generated logs provide a lot of information that can be useful in many ways. Within the FITTEST approach, logging is even a necessity. The main FITTEST components depend on feedback collected after the application has been deployed. Log files are an important source for this feedback. In FITTEST, logs have to be generated systematically, in a well defined format, and in a non-invasive way. Existing solutions are unfortunately not good enough for this task.

We described a new logging solution. It is non-invasive, and can log both high level and low level events. To systematically log low level events we statically inject logging statements into the target program, and we combine this with a dynamic approach to make it smarter, namely by allowing a piece of run-time logic to filter and compose the stream of entries generated by low level logging before they are put in the log file. The high-level logging of events requires only a dynamic approach.

The solution is flexible; it allows the logging level to be adjusted at the run-time, e.g. to respond to increasing load. Currently it is still not possible to adjust the level per component (e.g. objects of a certain class in ActionScript); this is future work.

The logs are stored in a compressed binary format for performance reasons. However, we provide an export mechanism to XML. The format of this XML has been defined, which enables post-processing by other tools.

The solution has been implemented as a prototype. It is also part of the FITTEST Distributed Framework.

10.1 Future Work

Performance has so far not been an issue. However, it requires closer investigation. Our approach provides several dimensions for adjusting the load generated by the logger:

1. We change the logging level.

2. We let different classes to have different logging level.

3. We change the used serialization delegates (e.g. by marking some fields as unserialized).

The last two capabilities are not implemented yet. Once they are, we need to define an algorithm that determines at which places to make adjustments in order to respond to the application’s load.
Bibliography


S. Ducasse, T. Girba, and R. Wuysts. Object-oriented legacy system trace-based logic testing. 2006. ISSN 1052-8725.


A. FITTEST Log Formats

FITTEST logger produces logs in a structured XML-like format, but it is more compact than XML. In favor for interchangeability, these logs are written in UTF-8 rather than as binary. This XML-like logs are called raw logs. They can in principle be directly consumed.

In our own framework, we actually have an additional intermediate level. We store raw logs in a compressed form, but they can be queried. The result of a query is a list of log entries. In other words, it is another log. This is then exported to a full XML file to be consumed by other FITTEST tools.

A.1 Raw Log File

<table>
<thead>
<tr>
<th>Character set</th>
<th>UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension</td>
<td>.log</td>
</tr>
</tbody>
</table>

A raw log is just a sequence of log entries. Each entry is organized as a section, which can recursively be made of sub-sections. The lowest level section is called paragraph, which consists of sentences. A sentence is basically just a string. There is no format imposed, however a specific logger may impose a certain syntax on the sentences—we will return to this later. Sections are tagged, which can be used to associate some semantic to them. The syntax is below.

\[
\langle \text{log} \rangle ::= \langle \text{section} \rangle \langle \text{whites} \rangle +
\]
\[
\langle \text{section} \rangle ::= \langle \text{section start} \rangle \langle \text{whites} \rangle \langle \text{section part} \rangle \langle \text{whites} \rangle ^* \langle \text{end marker} \rangle
\]
\[
\langle \text{section start} \rangle ::= \%<S \langle \text{whites} \rangle \langle \text{time stamp} \rangle ? \langle \text{whites} \rangle "\langle \text{tag} \rangle"
\]

— note that a tag has to be quoted.
\[
\langle \text{end marker} \rangle ::= \%>
\]
\[
\langle \text{whites} \rangle ::= \langle \text{white} \rangle ^*
\]
\[
\langle \text{section part} \rangle ::= \langle \text{paragraph} \rangle \mid \langle \text{section} \rangle
\]
\[
\langle \text{paragraph} \rangle ::= \%<P \langle \text{whites} \rangle \langle \text{sentence} \rangle \langle \text{whites} \rangle ^* \langle \text{end marker} \rangle
\]
\[
\langle \text{sentence} \rangle ::= \%<\langle \text{sentence content} \rangle \%>
\]
\[
\langle \text{time stamp} \rangle ::= \langle \text{UTC offset} \rangle \% \langle \text{UTC time} \rangle
\]
\[
\langle \text{UTC offset} \rangle ::= (+1-2)? \langle \text{offset in minutes} \rangle
\]

Additional notes and constraints:

1. \langle \text{sentence content} \rangle is any sequence of character, but it should not contain the combination \%\%, which is used to identify the sentence’s end.

2. Time stamp is written as a pair \( o : t \) where \( t \) is UTC time, which is location independent, and \( o \) is the offset of the local time with respect to the UTC time. With this offset we can infer what the local time of \( t \) is.

The UTC-time \( t \) encoded as a single integer, which expresses the number of milli-seconds elapsing since midnight 1-st January 1970 and the actual UTC time when \( t \) is measured.

The offset \( o \) is also an integer, expressing the time difference between the location on which \( t \) is measured and the UTC time, expressed in minutes.

© 2011 FITTEST Consortium 47
A.1.1 Compression

Organizing logs in paragraphs and sentences enables compression. We can collect sentences that occur repeatedly in a log, and then index them in a dictionary. Then we can replace the occurrences of a sentence \( s \) with its index, and thus making the log more compact, while it is still efficiently searchable. We can do the same with long section-tags or even paragraphs that occur repeatedly.

Timestamps can be compressed as well. Only sections are timestamped. If \( S \) is a section with timestamp \( t \), we can instead represent it as the difference with the previous timestamp, which usually takes less space to store.

The current logging framework partially implements the the above compression approach. This is to be completed and improved in the future.

A.2 Specialized Raw Format

The basic raw-log format just defines the concept of sentence, paragraph, section. For most logging purpose we will want to have a richer semantic of the logs. E.g. if we serialize objects to logs, we want to have concepts like class name and fields. We also have to specify how the fields of an object will be mapped to sections and sentences. This can be done by further specifying the raw-log syntax. For example by defining a syntax for the sentences and the section tags. We will use the following meta notation for describing the syntaxes:

1. \( \text{Sen}(x) \) means that \( x \) will be formatted as a sentence, with \( x \) as the content.

2. \( \text{Par}(s_1...s_n) \) means that this will be formatted as a paragraph with each \( s_i \) forming a sentence of the paragraph, appearing in the order as specified.

3. \( \text{Sec}(T,S_1...S_n) \) means that this will be formatted as a section with tag \( T \), with the \( S_i \)’s concatenated to form the body of the section. This section has no timestamp.

4. \( \text{Sec}_{\text{timed}}(T,S_1...S_n) \) is as above, except that the section can be timestamped.

A.2.1 Object Format

Here we define how objects are printed/serialized in a raw log. Our concept of ‘object’ broadly represents a structure of some data. It can be a real object in the target program, or a fake object that we use to abstractly represent a complicated object.

We will have two kinds of objects: simple and nested. A simple object only has a single paragraph with a single sentence.

\[
\text{(object)} \quad ::= \quad \langle \text{simple object} \rangle \mid \langle \text{nested object} \rangle
\]

\[
\text{(simple object)} \quad ::= \quad \text{Par}(\text{Sen}(\langle \text{simple value} \rangle))
\]

\[
\text{(simple value)} \quad ::= \quad \langle \text{undefined} \rangle \mid \langle \text{null} \rangle \mid \langle \text{integer} \rangle \mid \langle \text{numeric} \rangle \mid \langle \text{boolean} \rangle \mid \langle \text{string} \rangle \mid \langle \text{unserializable} \rangle
\]

\[
\langle \text{undefined} \rangle \quad ::= \quad \text{undefined:void}
\]

\[
\langle \text{null} \rangle \quad ::= \quad \text{null:Null}
\]

\[
\langle \text{integer} \rangle \quad ::= \quad \langle \text{int value} \rangle:\text{int}
\]

\[
\langle \text{numeric} \rangle \quad ::= \quad \langle \text{numeric value} \rangle:\text{Number}
\]

\[
\langle \text{boolean} \rangle \quad ::= \quad \langle \text{boolean value} \rangle:\text{Boolean}
\]

\[
\langle \text{string} \rangle \quad ::= \quad \langle \text{string value} \rangle:\text{String}
\]

\[
\langle \text{unserializable} \rangle \quad ::= \quad ??:(\langle \text{class name} \rangle)
\]

Additional notes and constraints:

1. There is no special format for natural numbers (non-negative integers).

2. A numeric value is either an integer or a dot-separated float value like 1.05.

3. A boolean value is either \text{true} or \text{false}.

4. A string value has to be quoted, e.g. "hello world".

5. A class name can be simple of fully qualified. In ActionScript a fully qualified name of a class \text{Item} looks like this \text{eu.fittest.mypackage::Item}. 
Some examples of simple objects:

```plaintext
%<P %<| undefined : void |%> %>
%<P %<| null : Null |%> %>
%<P %<| 199 : int |%> %>
%<P %<| 0.00000123 : Number |%> %>
%<P %<| false : Boolean |%> %>
%<P %<| "hello world!" : String |%> %>
%<P %<| ??? : eu.fittest.MyPackage::Item |%> %>
```

### Nested Object

A nested object has multiple fields, some of these may contain subobjects. These fields are grouped in one or more paragraphs. This is to facilitate compression —see Subsection A.1.1. E.g. the logger may group the fields such that those that tend to vary together are put in the same paragraph. In any case, the logger is free to decide how to arrange the fields in paragraphs; but they must be packed in paragraphs.

There is however one restriction imposed by the syntax (below). Consider as an example the class below:

```plaintext
class Person {
    var name : String ;
    var age : int ;
    var spouse : Person ;
    var height : int ;
}
```

The `spouse` field of a person contains another person as a subobject. This cannot be packed in a paragraph, and has to be packed as a section instead. We will however put the field name (`spouse`) in the paragraph that precedes it. For example, here is how a person can be serialized:

```plaintext
<S "O: Person"
%<P %<| 1=0:ID |%>
%<| name="Sponge Bob" : String |%>
%<| age=4: int |%>
%<| spouse= |%> // the spouse-object can be found in the next section
%>
<S "O: Person"
... // here is the serialization of the spouse
%>
```

Above, `I` is by the way a fake field; this is explained later. Below is the syntax of nested objects:

```plaintext
(nested object) ::= Sec(object tag), (object body)
(object tag) ::= O:(class name)
    — Note that an object tag is a section tag, so it has to be quoted.
(object body) ::= (object part)*
(object part) ::= Part(simple field)1 ... (simple field)n — at least one field.
    | Part(simple field)1 ... (simple field)n (subobject marker) (nested object) — n can be 0.
(simple field) ::= Sen(field name) = (simple value)
    | Sen(field name) = (iref)
(subobject marker) ::= (field name) =>
```

### Object with Cyclic Structure

Suppose we have an object `o` to serialize into the log. When the subobjects structure beneath `o` contains a cycle, the logger cannot just recursively serialize `o`; it will not terminate. So, as it traverses `o`, it will keep track of the object references it saw, and number them. Let’s call these numbers are called local indices. When it comes to a field that points to an object `p` it has seen before, it will not serialize the object. Instead it will just write `p`’s local index. For `o` and each of its subobjects the logger creates a fake field called `I`; it contains the local index of the object, and will be marked as having the type `ID`.

For example, consider an object `p` of the class Person such that its spouse points to `x` itself. This could be serialized as follows:
The indices are called 'local' because they are only unique within the traversal of a given top-level object that we are serializing. They are not unique over the whole log. So, if after Patrick we write another person Bob to the log, Bob’s local indices will again start from 0.

We usually put $I$ as the first field; but it does not have to be. Furthermore, if an object is never referred back to in a cycle, then we do not need to know its $I$-field. In that case the field can be dropped.

Note: in the future the field will be renamed to prevent clash with normal field names.

Array, Collection, and Dictionary

An array or collection $a$ is treated as an ordinary object of type $Array$ respectively $Collection$. Each element $x$ of $a$ is treated as a field called $elem$ (so, we will have $N$ fields called $elem$). If $a$ is an array, the elements will be serialized the same order as in the array. Here is an example of an array of two elements (10 and 100):

```
<S "O: Array"
  %<P %<\ elem=10:int >%
  %<\ elem=100:int >%
%
```

A dictionary $d$ is treated as an object of the type $Dictionary$. Each entry $(key, value)$ in the dictionary will be treated as two consecutive fields called $key$ and respectively $val$. Here is an example of a dictionary containing two entries (0, 10) and (1, 100):

```
<S "O: Dictionary"
  %<P %<\ key=0:int >%
  %<\ val=10:int >%
  %<\ key=1:int >%
  %<\ val=100:int >%
%
```

A.2.2 Event Format

High level events

A high level event $e$ is described by an 'event object' and a state object. The first describes what kind of event $e$ is (e.g. a click on a button, or an update to a text-field), and the second abstractly describes the state of the target application 'when' the event occurs. We cannot impose here what the exact temporal relation between the sampled state and the event; this depends on the implementation of the logger. Typically, the state is sampled after the event occurs, and before the next event occurs.

$\newcommand{\Sec}{\text{Sec}}$

```
(high level event) ::= \Sec_{time}(E_1 (event object) (state object))
(event object) ::= (nested object)
(state object) ::= (object)
```

An event object has the type $RecordEvent$ and has the following fields:

1. type: $String$, specifies what kind of event that is. E.g. "$item\_click" means the event was a click on some display object.

2. targetID: $String$, is a string that identifies the display object that is targeted by the event. E.g. if the event was a click on a button $b$, this this would be the ID-name of $b$. Having this ID allows another FITTEST component, e.g. its test execution system, to find $b$. This presumes that all display objects in an application have unique IDs. In reality this may not be the case.

© 2011 FITTEST Consortium
3. \texttt{args: Array}, is an array containing the arguments passed to the event. Event like clicking on a button has no argument. On the other hand, an update to a text-field is an event that takes one argument, namely the new value in the text-field.

Figure A.1 shows an example of the serialization of high-level event.

\textbf{Low level events}

Below is the syntax of events related to entering and exiting a function. Suppose in in a function \( f \) we call \( x.g(y) \). Two events are provided to log this. The 'function call entry' event is used to log the state of the target of the call \((x)\) and the parameters passed just before the call. This event will be logged with the syntax as shown below, where \( f \) will be the caller and \( g \) the callee. The 'function call exit' is used to log the target object, the return value, and the exception thrown when the call returns.

The latter event implies that the logger is somehow able to catch and rethrow \( f \)'s exceptions. However, this is the logger's own decision. Here we simply describes the syntax of how those events will be logged.

\begin{verbatim}
(function call entry) ::= Secimed((function call entry tag), (callee) (target object) (args))
(function call entry tag) ::= FCE:(function name)
(callee) ::= Par((function name))
(function name) ::= (function name)\#(class name)
(target object) ::= (object)
(args) ::= Sec(args (object)*)

(function call exit) ::= Secimed((function call exit tag), (callee) (target object) (return object) (exception object))
(function call exit tag) ::= FX:(function name)
(return object) ::= (object)
(exception object) ::= (object)
\end{verbatim}

Rather than logging a call to \( x.f(y) \) from the caller-side, we can also log it from inside \( f \) itself. The event 'function entry' is used to log the state of the target \( x \) and the parameters passed to \( f \) when the function is just entered. The event 'function exit' is used to log the target \( x \) and the return value when \( f \) reaches its normal end point.

\begin{verbatim}
(function entry) ::= Secimed((function entry tag), (target object) (args))
(function entry tag) ::= FE:(function name)

(function exit) ::= Secimed((function exit tag), (target object) (return object))
(function exit tag) ::= FX:(function name)
\end{verbatim}
The program of a method can be divided into 'blocks'. A block is a maximal consecutive segment of instructions that does not contain a jump nor targeted by a jump. The events below can be dispatched to log when a block is entered, when an exception handler is entered, when a loop is entered, and when a loop is exited. The logger is responsible for figuring out how to properly keep track the execution flow through blocks.

\[
\begin{align*}
\text{(visit block event)} & \quad ::= \text{Sec\,timed}((\text{visit block tag}),) \\
\text{(visit block tag)} & \quad ::= \text{B:}(\text{block id}),\text{function name} \\
\text{(handling exception event)} & \quad ::= \text{Sec\,timed}((\text{handling exception tag}),\text{(exception object)}) \\
\text{(handling exception tag)} & \quad ::= \text{BEH:}(\text{block id}),\text{function name} \\
\text{(enter loop event)} & \quad ::= \text{Sec\,timed}((\text{enter loop tag}),) \\
\text{(enter loop tag)} & \quad ::= \text{BLE:}(\text{block id}),\text{function name} \\
\text{(exit loop event)} & \quad ::= \text{Sec\,timed}((\text{exit loop tag}),\text{Par}((\text{iteration count}))) \\
\text{(exit loop tag)} & \quad ::= \text{BLX:}(\text{block id}),\text{function name} \\
\text{(iteration count)} & \quad ::= \text{cnt:}(\text{integer})
\end{align*}
\]

A.3 XML Log File

| Character set | UTF-8 | Extension | .xml |

The previous section defines the syntax of raw logs. These are subsequently stored in a compressed form. We can query them, and then export the result in XML for consumption by other tools. Here we define our XML log format.

\[
\text{element body} = \text{content} \ (entry)^*
\]

\[
\langle entry \rangle = \langle\text{E}\rangle | \langle\text{low level event}\rangle
\]

\[
\text{element E} = \quad \text{— high level event}
\]

\[
\text{attrib optional} \quad \text{t} \quad \text{— time stamp}
\]

\[
\text{content} \quad \langle\text{event object}\rangle \langle\text{state object}\rangle
\]

\[
\langle\text{event object}\rangle = \langle\text{O}\rangle
\]

\[
\langle\text{state object}\rangle = \langle\text{object}\rangle
\]

\[
\langle\text{object}\rangle = \langle\text{OV}\rangle
\]

\[
\text{element O} = \quad \text{— nested object}
\]

\[
\text{attrib type} \quad \text{— the object’s type}
\]

\[
\text{content} \quad \langle\text{fd*}\rangle \quad \text{— fields}
\]

\[
\text{element fd} = \quad \text{— object’s field}
\]

\[
\text{attrib n} \quad \text{— field name}
\]

\[
\text{content} \quad \langle\text{O|V}\rangle \quad \text{— field’s content}
\]

\[
\text{element V} = \quad \text{— simple object}
\]

\[
\text{attrib v} \quad \text{— the object’s value}
\]

\[
\text{attrib ty} \quad \text{— the object’s type}
\]

Conform to what is said in Section A.2.2, an event object has the type `RecordEvent` and has these fields: `type:String`, `targetID:String`, and `args:Array`. Figure A.2 shows an example of a high level event (in XML).

The syntax for low level events is shown below.

\[
\langle\text{low level event}\rangle = \text{FCE} | \text{FCX} | \text{FE} | \text{FX} | \text{B} | \text{BEH} | \text{BLE} | \text{BLX}
\]

\[
\text{element FCE} = \quad \text{— function call entry event}
\]

\[
\text{attrib f} \quad \text{— caller function name}
\]

\[
\text{attrib ce} \quad \text{— callee name}
\]

\[
\text{content} \quad \langle\text{target object}\rangle \text{args}
\]
Figure A.2: A log in the XML format.

\[
\langle \text{target object} \rangle = \langle \text{object} \rangle
\]

\text{element} \text{args} = \text{content} \langle \text{object} \rangle^* \\

\text{element} \text{FCX} = \quad \text{— function call exit event} \\
\quad \text{attrib} \ f \quad \text{— caller function name} \\
\quad \text{attrib} \ ce \quad \text{— callee name} \\
\quad \text{content} \ (\langle \text{target object} \rangle \langle \text{return object} \rangle \langle \text{exception object} \rangle) \\

\langle \text{return object} \rangle = \langle \text{object} \rangle \\
\langle \text{exception object} \rangle = \langle \text{object} \rangle \\

\text{element} \text{FE} = \quad \text{— function entry event} \\
\quad \text{attrib} \ f \quad \text{— function name} \\
\quad \text{content} \ (\langle \text{target object} \rangle \text{args}) \\

\text{element} \text{FX} = \quad \text{— function exit event}
attrib  f — function name
content ⟨target object⟩ ⟨return object⟩

element B = — visit block event
attrib  f — function name
attrib  i — block ID

element BEH = — handling exception event
attrib  f — function name
attrib  i — block ID
content ⟨exception object⟩

element BLE = — enter loop event
attrib  f — function name
attrib  i — block ID

element BLX = — exit loop event
attrib  f — function name
attrib  i — block ID
attrib  cnt — iteration count