ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

**Project number:** 258109  
**Project Acronym:** FastFix  
**Project Title:** Monitoring Control for Remote Software Maintenance  
**Call identifier:** FP7-ICT-2009-5  
**Instrument:** STREP  
**Thematic Priority:** Internet of Services, Software and Virtualisation  
**Start date of the project:** June 1\(^{st}\), 2010  
**Duration:** 30 months

**D5.1: State of the art in fault replication and test automation**

**Lead contractor:** INESC ID  
**Author(s):** Paolo Romano, Benoit Gaudin, João Garcia, Luís Rodrigues, António Rito Silva  
**Submission date:** December 2010  
**Dissemination level:** Public

Project co-founded by the European Commission under the Seventh Framework Programme

© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document examines and discusses the state of the art regarding fault replication and test automation and relates them to the goals of the FastFix project. First, some central concepts regarding the foundations of state replication are presented, such as deterministic re-execution and the problem of trust in software, as well as the main guiding ideas in test automation. Then, the most relevant academic and commercial fault replication and test automation system are presented and their relation to the context of FastFix is explored. Finally, we place FastFix’s goals in the context of the state of the art, describe the contributions the project will make and conclude.
Contents

1 Introduction .............................................................................................................. 6

2 Foundations .......................................................................................................... 8
   2.1 Fault Replication............................................................................................... 8
      2.1.1 Sources of Non-Determinism ................................................................. 8
      2.1.2 Deterministic Replay .............................................................................. 9
      2.1.3 Privacy .................................................................................................... 10
      2.1.4 Trust in Software Maintenance .............................................................. 11
      2.1.5 Statistical Debugging ............................................................................. 11
   2.2 Test Automation ............................................................................................... 12

3 Existing Systems and Applications for Fault Replication ................................. 14
   3.1 Approaches to Fault Replication ................................................................. 14
   3.2 Hardware-assisted Solutions ........................................................................ 14
      3.2.1 Bacon et al. ........................................................................................... 15
      3.2.2 Flight Data Recorder ........................................................................... 15
      3.2.3 BugNet ................................................................................................. 16
      3.2.4 DeLorean ............................................................................................... 17
      3.2.5 Conclusions .......................................................................................... 18
   3.3 Software-based Solutions ............................................................................... 18
      3.3.1 Error Reporting Tools ............................................................................ 18
      3.3.2 Checkpointing ........................................................................................ 19
      3.3.3 Igor ........................................................................................................ 19
      3.3.4 Flashback ............................................................................................... 20
      3.3.5 Jockey and liblog ...................................................................................... 21
      3.3.6 DejaVu .................................................................................................. 22
      3.3.7 Bressoud et al. ..................................................................................... 22
      3.3.8 InstantReplay ......................................................................................... 22
      3.3.9 SMP-Revirt .......................................................................................... 23
      3.3.10 R2 .......................................................................................................... 24
      3.3.11 PRES ................................................................................................. 24
      3.3.12 Castro et al. ........................................................................................ 26
   3.4 Statistical Debugging Solutions ...................................................................... 26
      3.4.1 CBI ......................................................................................................... 26
      3.4.2 Holmes ................................................................................................... 28
      3.4.3 CCI ......................................................................................................... 29
      3.4.4 Conclusions .......................................................................................... 31

4 Existing Systems and Applications for Test Automation .................................. 33
   4.1 Automatic Test Execution .......................................................................... 33
4.1.1 JUnit .......................................................... 33
4.1.2 NUnit .......................................................... 33
4.1.3 Selenium .................................................... 33

4.2 Automatic Test Case Generation .......................................................... 34
  4.2.1 Contract-like Specifications .................................................... 34
  4.2.2 Labeled Transition Systems, (E)FSM, State Charts ...................... 35
  4.2.3 Testing through model-checking .............................................. 36
  4.2.4 Fault Injection ....................................................... 36

5 Shortcomings of Current Solutions ....................................................... 38

6 Open Research Questions ................................................................. 39

7 Conclusion ......................................................................................... 41

Bibliography ............................................................................................ 42
List of Figures

Figure 1 Example system with the Flight Data Recorder (Source: [82]). ............................. 15
Figure 2 BugNet Architecture (Source: [59]). ............................................................... 16
Figure 3 DeLorean Architecture (Source: [58]). .............................................................. 17
Figure 4 Examples of error reports from Dr. Watson (left) and Talkback (right). ............... 18
Figure 5 Example of exploratory replay in PRES (Source:[36]) ...................................... 25
1 Introduction

In the lifecycle of software application development and execution, faults have always been and still are bound to happen. As the IT environment of devices and applications has become denser, more sophisticated and more critical, the software development cycles have become more complex: shorter development cycles, release of prototypical applications, diversity of devices and stronger connections between devices. The dynamism and heterogeneity of modern software environments requires that developers be, as much as possible, aided in reproducing and fixing those faults. Fault replication is a functionality that, if included in a software maintenance infrastructure, provides it with the ability of replicating errors at the development site as similarly as possible to the real circumstances in which the faults occurred. The FastFix project aims at developing a software maintenance platform that allows applications to be monitored, their fault replicated precisely (with additional context information) and finally be automatically patched.

This document analyses techniques, experiments and platforms in fault replication that constitute the state-of-the-art in this domain and upon which the FastFix will develop.

In order to obtain a way to solve a system crash or usage fault, it is highly desirable to have the ability to roll back the system or application to a previous execution state in the vicinity of the buggy code, and deterministically replay the sequence of instructions executed before the crash (see 2.1.2). The discovery of approaches that enable a platform to gather information for deterministic replay without compromising application performance and user privacy is the main focus of fault replication research and development.

This document focuses also on the issue of test automation which is a complementary aspect of FastFix. An ideal software maintenance system should be completely automated. If a software fault were to happen on a client installation, it would be automatically reported to the developers and, based on the report, the maintenance system would generate a patch for the application and transparently update the client installation. Unfortunately, many software faults cannot be solved automatically and require the intervention of software developers using debugging and testing tools. This may happen for many reasons: the application
monitoring was incomplete, the error report only provides partial, probabilistic or anonymized application input data and context information, or the fault is just too complex to be corrected automatically. Therefore, this document also discusses some of the testing tools that can aid maintenance staff in debugging and identifying the sources of software faults and identifies those techniques that can be more useful to complement the features of FastFix.
2 Foundations

2.1 Fault Replication

Fault replication can be defined as the exact repetition of an application execution that included or ended in an abnormal situation. It should be a part of a modern software development and maintenance infrastructure. It is used to help developers correct (debug) faults that are reported. These faults are reported usually automatically by clients that are using the application at stake, which is monitored for maintenance information. The clients who have been unable to accomplish their goals or the application itself detect an error and send (via network) an error report to the software maintenance team who then tries to correct it. There have been many approaches to gathering information from a running program in order to be able to replicate that execution if the program were somehow to fail.

A first, and somewhat “brute force” approach to reproduce the instruction stream is called “content-based” replay [20]. The idea is simple: during the execution at the client side of the application store all the data that is read by the instructions (from the register file and the main memory) into a trace file. After the error report is sent, when the application is replayed at the maintenance site this trace file is used to provide every instruction in the application with the necessary input. This approach is mainly of theoretical importance since in practice it will result in enormous trace files (even though some hardware-assisted techniques based on this idea have actually been proposed [82]).

2.1.1 Sources of Non-Determinism

Computer programs can be seen as precise sequence of instructions that are always executed in the same way. However, during program executions, some factors often interfere preventing that the timing and the sequence of executed instructions be always identical. These factors are the result of either internal (e.g. access order to shared memory locations by multiple threads) or external sources (e.g. inputs from keyboard or network). Because the outcome of an application whose behaviour changes with varying inputs cannot be predicted it is called non-deterministic and those unknown inputs are called sources of non-determinism. If the precise values of all non-deterministic operations (see ahead) are provided in exactly the same order, when a non-deterministic program is re-executed, the re-execution will be deterministic. Obviously this is critical for debugging, since only with a
precise re-execution that can be repeated manifold, can a fault be understood and corrected. The main sources of non-determinism in computer programs are [66]:

- **Processor-specific instructions**: some processors have dedicated input instructions, whose output depends on the processor version. One example is the RDTSC on a Pentium processor, which reads the timestamp counter.

- **Signals**: signals make processors aware of external events, but can happen asynchronously. A signal typically alters memory state, registers and the program control flow.

- **Reads of un-initialized memory locations**: the values read from memory locations which were not initialized often change in different runs.

- **System calls**: certain system calls may present a differing behaviour because their results depend on the environment in which they are running. The Linux system calls gettimeofday() and uname() are some examples.

- **Different access order of shared memory locations**: the interleaving of read/write accesses to shared memory locations by different threads may vary from run to run.

- **Inputs**: the inputs from keyboard and network may not be the same depending on the execution.

### 2.1.2 Deterministic Replay

A more efficient approach to support deterministic replay is re-execution, based on the observation that almost all instructions and states can be reproduced as long as all non-deterministic factors (see 2.1.1) that affect the program’s execution can be replayed in the same way [46]. Deterministic replay operates in two phases:

- **Record phase**: consists of capturing data regarding non-deterministic events and putting that information into a trace file.

- **Replay phase**: the application is re-executed consulting the trace file to force the replay of non-deterministic events according to the original execution.

The most problematic source of non-determinism comes from timing variations caused by hardware, such as cache state, memory refresh-scrubbing, and timing variations on buses and devices [59][69]. Therefore, even when re-executing the same code with the same input on
the same machine, the exact time of execution of an instruction or a segment of code could vary from one run to another. While normally timing variations usually affect only performance, not logic execution and states, there are a few important exceptions. For instance, timing variations can affect an application’s outcome when a thread is scheduled out, or when a thread receives a signal or callback function. To handle these sources of non-determinism, a typical way is to record them with logic time [48] instead of physical time. For example, the system can use the number of executed instructions to remember when a thread is scheduled out or when a signal is delivered. Replaying thread scheduling and interrupt/signal handling according to logic time is indeed sufficient [11][32][41][70][74][79][81] to ensure the accuracy of the re-execution, at least in uniprocessor systems.

Implementing replay for multiprocessors (shared-memory multiprocessors [SMPs] and multi-core processors) is much more challenging than for single processor systems. In fact, with multiple threads executing simultaneously on different processors, their interleaving in the access to shared resources (e.g. memory) can vary from one run to another. Because writes on one processor can affect reads on another processor, the results of memory races (near simultaneous accesses to the same data by different threads on different processing cores) must be recorded and replayed. This raises the additional need to identify memory races and capture the global orders of shared memory accesses. The costs associated with these activities can in practice be very high [21][60][65]. Obviously, this is not an issue when threads are independent and do not access shared data concurrently. Nevertheless, the transition to multi-core processors has fostered a number of research efforts aimed at developing novel mechanisms for execution replay in multiprocessor systems [41][21][47][24][65].

### 2.1.3 Privacy

A major issue that affects existing fault replication mechanisms is that, in order to be able to reproduce a fault occurred in a production system, the user must consent to send detailed information concerning the state of the locally deployed application. Unfortunately, the possibility of leakages of sensible information from the failure report can raise serious privacy concerns and prevent the release of information to the programmers.

Tackling this problem with standard data obfuscation techniques [4] is clearly not an option, given that even minimum careless changes of the execution trace may in theory compromise
the accuracy of the replay process, failing to reproduce the faulty behaviour. Or, even worse, corrupting the trace and completely preventing the re-execution. A fundamental, open research question is therefore whether it is possible to devise novel “smart” data obfuscation techniques able to ensure adequate privacy guarantees, while still preserving the meaningfulness of the released failure information.

2.1.4 Trust in Software Maintenance

Indeed, performing fault replication brings to the forefront the issue of the trust relationship between software developers and their clients. Obviously, if users run their software in an isolated environment, they have guarantees that their private information will not be leaked to the outside. Once the security of a software installation begins being exposed by connecting it to an outside network, users have to, at some level, trust developers with their personal data. It is easy to see that there are privacy issues at many levels of the software design/maintenance lifecycle. For example:

- Does the application only report the information it says it’s reporting?
- Once the information is in the developers computers, who will access to it?
- Can developers be trusted to use the information in a limited and professional way?
- Even if I don’t want to send personal information, does an anonymous error report, with technical information about how my application failed, reveal information about my actions?

This list of security concerns makes it obvious that, once a person uses an application, which includes software maintenance interactions with the application developers site there needs to be a clear, and most likely legally binding, agreement as to what information is shared and how it is used.

2.1.5 Statistical Debugging

The deterministic replay approach is not the only way to improve the task of debugging. Statistical debugging is a recent approach that aims at isolating bugs by analyzing information gathered from a great number of users. This idea is based on the notion that software applications are usually executed by a large user community. Therefore, instead of trying to detect the bug relying only in data from executions from a single user, statistical debugging attempts to speed up the bug tracking process by distributing the monitoring
across different clients. By doing this it is possible to extract afterwards patterns of similarity among the universe of executions collected that could lead to the failure.

In general, the infrastructure for statistical debugging consists of a central database, which receives user reports from both successful and unsuccessful runs, and a module to analyze statistically the collected data. After isolating the failure, the central site can send back to the users a patch to fix the application.

This technique also raises the issue of how much information to record during production runs in order not to degrade runtime performance at the user site. Alongside, it must take into account scalability issues, given that the central site has to be able to manage all the received reports.

### 2.2 Test Automation

Testing a system can be described as providing it with stimuli and observing whether the corresponding system outputs are consistent with the observer’s expectation. This process aims to give confidence regarding the system reliability. Tests that induce the observation of some misbehaviour, highlight an error in the system. However, successful testing does not indeed aim to ensure that the system is error free, as errors usually cannot all be detected through testing. It is actually agreed that a good test is a test that exhibits a failure. Moreover, the current systems complexity makes it impossible to perform testing in an exhaustive way, i.e. not all the possible inputs of the system can be used as stimuli for testing. It is true though that the most tests are successfully passed by the system, the more our confidence in its reliability increases.

Testing is part of a classical software development phase. However, the quality and relevance of tests being produced are very sparse. One explanation is that implementing test cases can be a very tedious process. Usually, the choice of test cases is decided from source code and relies on human expertise. This is why some works have been conducted in order to automate this process, leading to the notion of test automation.

Test Automation can relate to two different concepts. First it can refer to the “automation of test execution”. This case consists of techniques such that implemented tests can be automatically executed and decision on whether the application passed the tests or not is also automatic. A typical and well known tool for technique is JUnit for Java.
Test automation may also refer to the “automation of test cases generation”. This case considers the creation of tests cases in order to test an application without manual intervention. These test cases may be designed from some specifications or models. The techniques required are far more advanced than in automation of test execution and may rely on model-based approaches, static analysis, etc.
3 Existing Systems and Applications for Fault Replication

Tracking down and fixing bugs in released software can be a nightmare, costing a significant amount of time and money. The main difficulty lies in being able to reproduce the bug at the developer site. Section 2.1 described the naïve approach of recording the result of all operations that a program performs in order to replicate them. It is obvious even to the most inexperienced observer that this poses an unacceptable performance overhead. Writing every memory operation into persistent storage more than doubles the cost of executing an application. However, from that point on, several, more sophisticated techniques have emerged.

3.1 Approaches to Fault Replication

In the following section, we overview, in more detail, some of the more recent approaches to execution replay, classifying them along two main directions: 1) whether they rely on ad-hoc hardware extensions, or are purely software-based, and 2) whether they can be applied to multiprocessors, or only to uniprocessors. (For a more thorough analysis of the field we recommend an early survey by Dionne et al. [22] and later ones by Huselius [37] and Cornelis et al. [20]).

3.2 Hardware-assisted Solutions

Most of the hardware-assisted approaches for execution replay address the general case of multiprocessor systems. Both hardware and software solutions to fault replication have, during the applications’ record phase, to gather inputs from sources of non-determinism. The main difference between both is in the way that they try to overcome the performance overheads of the recording phase. Hardware solutions leverage the speed of hardware but, in general, use quite straightforward techniques. Ahead we will see that software approaches, because of the comparative slowness but higher design flexibility, have become extremely sophisticated. Although addressing fault replication with a dedicated hardware is outside the scope of FastFix, next we describe the most relevant hardware-assisted fault replication solutions.
3.2.1 Bacon et al.
Bacon et al. [3] first proposed supporting multiprocessor replay in hardware by spying the cache-coherence protocol (snooping). Their simulated system uses a hardware instruction counter piggy-backed to cache-coherence messages to identify sharing.

3.2.2 Flight Data Recorder
Flight Data Recorder (FDR) [82] explores snooping much further, using modern commercial workloads. Rather than replaying the entire execution, FDR focuses on replaying the last one second of execution before a crash. They instrument the cache-coherence hardware to detect memory races and generate constraints, using a modified version of Netzer’s algorithm [63] to reduce redundant constraints.

![Figure 1 Example system with the Flight Data Recorder (Source: [82]).](image)

Input and interrupts are also logged. Checkpoints are implemented by logging old values of memory as it is modified. In case of a crash, the entire memory is dumped, which in conjunction with the old memory log, allows the state of one second ago to be reconstructed. The rest of the log allows FDR to replay the entire state of the system as it executed for the last second before the crash. Afterwards, the FDR team explored various optimizations to the system, including techniques to reduce the number of constraints logged and efficiently compress them [83]. These optimizations reduced the constraint log size (around 2 MB/s in [82]) by a factor of 25 on average. The memory checkpoint data made up the majority of FDR’s log.
3.2.3 BugNet

In order to analyse a system crash, however, the entire state of the system may not be necessary. BugNet also focuses on multiprocessor systems and makes use of dedicated hardware buffers to provide architecture support to trace runtime information to re-execute instructions that preceded a system failure. BugNet is based on FDR, but, contrary to this solution, it doesn't strive to replay the full system execution. Rather, it focuses on detecting application level bugs and hence replays only executions in user code and shared libraries.

Rather than attempting to checkpoint the whole state, BugNet [59] logs the first read from shared memory in each replay interval, or when a data race is detected. Only the state of the register file (including the instruction pointer) is regenerated by execution replay. Checkpoints can be terminated by program crashes, interrupts, system calls and context switches. Regarding terminations caused by application failures, the logs generated during recording will be gathered to help the debugging.

The memory state that affects execution can be reconstructed from the log. During replay, each processor is capable of replaying independently of the other processors. Constraints are recorded only as an aid to debugging, to help correlate the interleaving of processor execution, and are not necessary for correct execution.

This method results in small log sizes, where a log size of around 500KB allows capturing a replay window size of 10 million instructions. Since 500KB is not too much, users can effortlessly communicate the log back to the developer. Furthermore, BugNet has very little performance overhead, and the area overhead is around 48 KB for the few hardware buffers required.
The authors of BugNet went on to develop a different kind of constraint. Rather than logging constraints between individual instructions (which they call a *point-to-point method*), they use barrier-like logs of memory operations called Strata [60].

### 3.2.4 DeLorean

DeLorean [58] is a novel hardware-assisted scheme for deterministic replay, where instructions are atomically executed by processors as blocks (or chunks), similarly to transactional memory [35] or thread-level speculation [14]. Then, rather than recording data dependences, it logs the commit total order of chunks. This results in two main advantages over the previous schemes. First, since a processor’s memory accesses can overlap and reorder within and across the same-processor blocks, DeLorean can record and replay an execution at a comparable speed to that of the current most aggressive consistency models. Second, it achieves a substantial reduction in log size.

Given that one has to make trade-offs between performance and log size, DeLorean provides two different executions modes. OrderOnly records at the speed of release consistency (RC) execution and replays at about 82% of that speed. This opposes to majority of other schemes, which only record at sequential consistency execution speeds and do not present details regarding replay speeds. Moreover, OrderOnly only needs 1.3 bits of memory-ordering log per processor per kilo-instruction, which is 16% of the log size needed by the state-of-the-art Regulated Transitive Reduction (RTR) [83] design.

A second execution mode called PicoLog reduces the memory-ordering log to 0.05 bits per
processor per kilo-instruction, which is 0.6% of the log size of RTR. In this mode, the authors estimate that the total memory-ordering log of an 8-processor 5-GHz machine is only about 20GB per day. In addition, PicoLog has a lower execution speed, 86% of RC’s execution speed, which is still higher than typical SC speed.

3.2.5 Conclusions
Unfortunately, the main issue with hardware-assisted solutions is that, despite recent works having suggested optimizations to reduce hardware complexity [55][21], they still require significant hardware modifications, none of which exists today, outside of simulated environments. Another fact that hardware systems bring to the foreground is that even with specially designed machines, the overhead and storage requirements of monitoring infrastructure are very large.

3.3 Software–based Solutions
This section presents an technological progression of the approaches to solve the problem of replicating faults using only software support: from error reporting tools (3.3.1) to tentative executions (3.3.11).

3.3.1 Error Reporting Tools
Microsoft’s Dr. Watson tool [57] and Mozilla’s Talkback [62] are examples of current solutions to gather and analyze the reason for a program crash. All these tools collect information that represents the final snapshot of execution state when the program crashed.

![Image of error reports from Dr. Watson and Talkback]

Figure 4 Examples of error reports from Dr. Watson (left) and Talkback (right).
While these crash reports have some utility, it is highly desirable to have the ability to roll back to a previous execution state in the vicinity of the buggy code, and deterministically replay the sequence of instructions executed before the crash. That is not possible with this traditional error reports which only describe the final state of an application and do not include enough information for re-execution.

### 3.3.2 Checkpointing

Checkpointing is an old technique [7] for providing system level support for rollback. Originally used for fault tolerance for distributed systems [27], checkpointing enables storing the previous execution state of a system in a failure-independent location. When the system fails, the program restarts from the most recent checkpoint in either a different machine or the same machine after fixing the cause of the failure. Being geared towards surviving hardware and operating system failures, most of these approaches, e.g. [1][50][80], are too heavy-weight to support rollback for software debugging.

This has motivated the research of a number of software-based execution replay techniques specifically targeted at supporting testing and debugging activities. We start by reviewing those targeting uniprocessor systems and then move to discuss solutions that cope with the additional sources of non-determinism characterizing multi-processor systems.

Two important classification criteria for software-based execution replay solutions is their degree of transparency for the applications (whether the application notices the fault replication infrastructure), and whether they are require modifications to the compiler, operating system or underlying virtual machine.

### 3.3.3 Igor

Igor [28] was one of the first software-based for deterministic replay. It relies on the checkpointing techniques for replaying programs and reconstructing the application state from a given previous checkpoint. However, since Igor does not record external I/O events, re-execution may not be identical to the original if the environment has changed. Furthermore, Igor does not support non-determinism caused by multithreaded programs.

This method operates collecting information at individual virtual memory pages level. To achieve this, it makes use of a new `pagemod()` system call, which determines the set of pages that have been changed since the previous checkpoint. To control checkpoints it employs another system call - `ualarm()`.
In the replay phase, Igor consults the log file to get the most recent checkpoint for each virtual memory page. After that, it uses an interpreter to continue the execution from the last checkpoint up to a instruction defined by the user.

However, Igor involves changes to (1) the compiler - to log data allocations, (2) both the library and loader - to initiate the trace and to enable dynamic function replacement. The recording overhead during production runs varies from 50% up to 400%. In addition, re-execution is about 140 times slower, thus becoming an unattractive approach.

3.3.4 Flashback

Flashback [71] is implemented as an operating system extension and it provides deterministic replay to assist software debugging. Flashback uses shadow processes to capture non-deterministic interactions between the process and the operating system, including system call invocations, memory-mapping, shared memory usage for multi-threaded applications and signals. For instance, if a process makes a read system call, Flashback records the return value of the system call and the data that the kernel copies into the read buffer. During replay, when this specific system call is found, the previous recorded value is then injected to the process by Flashback.

This tool provides three primitives:

- **checkpoint**: captures the current state and returns a handler state, allowing the program roll back to if required.
- **discard(x)**: discards the captured checkpoint provided, avoiding any future attempts to roll back to this specific state.
- **replay(x)**: rolls back the process to the previous execution state pointed by the state handler provided and then the execution is deterministically replayed up to where the **replay()** primitive is called.

These primitives are guaranteed through the use of shadow processes. A shadow process is a snapshot of the running process created by replicating the in-memory representation of the process in the operating system. Its creation is achieved by creating a new structure in the kernel and initializing it with the contents of the monitored process structure (e.g. registers contents, process memory, file descriptors etc). The pointer to the shadow process is stored in the current process structure. The copy-on-write mechanism is used in order to reduce
overhead. Moreover, since Flashback’s intent is not to recover from neither system crashes or hardware failures, one does not need to persistently store shadow processes, which still further reduces overhead.

Flashback can keep two or more shadow processes per process in execution. The use of multiple shadow processes can be useful for progressive roll-backs and replaying during debugging. For multithreaded applications, one needs to ensure that the different threads are scheduled in the same relative order during repetition. Flashback does this by tracing thread scheduling information during original execution to later force the same interleaving between threads during replay.

The main limitation of Flashback is that it requires modification to debugging tools to incorporate support for the framework. It then allows developers to change the source code of the debugged program explicitly to include calls to Flashback primitives. Nevertheless, it does not allow the programmers to incorporate their own debugging information to the program during the replay stage. Besides that, Flashback is also not suitable for profiling purposes because the replay mechanism does not allow the instrumentation of the target program for the replay phase. Finally, recording and replaying of signals and deterministic replay of multi-threaded applications are outlined in the future work but it is not currently supported by Flashback.

### 3.3.5 Jockey and liblog

Jockey [68] and liblog [30] take a different approach (and face additional different technical challenges) by implementing the record-replay mechanism as a user-space library that runs as a part of the target process. A major issue in this case is how to ensure deterministic replay of thread scheduling without the ability to observe and control context switches among kernel threads. Existing solutions to such a problem are unfortunately still suboptimal, imposing the adoption of a user-level cooperative scheduler on top of the OS scheduler, and preventing to support applications that intentionally use tight infinite loops (e.g. to implement home-grown spinlocks). An interesting feature of the solution in [30] is its ability to cope with distributed C/C++ applications, providing tools to ensure consistent group replay by ensuring to see consistent snapshots of the state across multiple processes [Lamp78] and to transparently trace message propagation from machine to machine.
3.3.6 DejaVu
A similar intent is shared by DejaVu project [45], which however targeted Java applications and built a modified Java Virtual Machine. In the Java domain, the work in [84] proposed a checkpointing/replaying technique that operates purely at the language level, without the need for JVM-level or OS-level support. This approach relies on an innovative combination of static (compile-time) and dynamic (run-time) analysis techniques to identify the relevant portion of the application's state to be captured as well as the program points at which capture/replay should occur. Unfortunately, this approach is restricted to work only with single-threaded applications.

3.3.7 Bressoud et al.
The idea of using a virtual machine to achieve the benefits of execution replay without needing to modify the software running on it was first proposed by Bressoud et al.[9]. This system uses execution replay to enable a high-availability primary-backup system. The main system (or primary) is logged, and the logs fed to one or more backup systems, which replay the logs immediately. This guarantees that the backup system is in the same state as the logging system, ready to take over in the event of a failure. In addition to facilitate replay on a different physical machine, encapsulating the execution replay logic within a VM provides the advantage of being able to use execution replay on an operating system kernel.

The above discussed mechanisms ensure deterministic re-execution only in uniprocessor systems, not coping with the non-determinism source associated with possible data races among threads simultaneously running on different processors.

3.3.8 InstantReplay
InstantReplay [49] was one of the first software solutions for deterministic replay on multiprocessors. It consists in a technique to replay shared memory accesses using an ordering-based approach. This technique allows the access to shared memory objects only through well-defined protocol CREW (Concurrent-Reader-Exclusive-Writer) primitives. This protocol is instrumented by InstantReplay for execution replay, and sets down one of two possible states for each shared memory object:

- **concurrent-read**: all the processors can read, none can write.
- **exclusive-write**: one processor (the owner) can read and write; all the others do not
have access.

Then, each shared memory object is extent with a version number that is incremented after each write access during both record and replay phases. All threads record versioning information to its own trace file.

During the record phase a reader traces the current version number of its shared object. In turn, a writer traces the current version number of its shared object and the number of readers since the previous write access on his shared object. During the replay phase a reader waits until the current version number of its shared object matches the previously traced version number. On another hand, a writer blocks until the version number on its shared object matches the previously traced version number and until the number of readers also matches the traced count.

This technique tends to require huge amounts of data recorded when the granularity of shared memory accesses within the program is very small. Thus, this technique can be considered virtually useless in systems where each memory access is deemed as shared.

More recently, [61] proposed a software-only implementation of Strata [60], one of the most optimized hardware based solutions. Almost all of these state-of-the-art software solutions impose more than 10X–100X production-run overhead, making them impractical [21]. The overhead comes mainly from capturing the global orders of shared memory accesses.

3.3.9 SMP-Revirt

To address the overhead problem, SMP-Revirt [24] recently made clever use of page protection (instead of instrumenting every shared memory access) to capture shared memory interactions among threads. Due to the page-level granularity, this method works well for applications with coarse-grained data sharing, such as some regular matrix applications. Unfortunately, for applications that have much finer-grained data sharing and more false sharing (such as server applications), this method still imposes 10X or more overhead on 2 or 4 processors [24]. Furthermore, in presence of some realistic workloads this approach was shown to suffer of serious scalability issues when the number of processors increases from 2 to 4. This is a result of increased false sharing and page contention [23], just like in page-based software distributed shared memory systems [51].
3.3.10  R2
R2 [32] proposes an innovative method that replays an application by (1) recording the results of functions selected by programmers during production run, and (2) returning the results during replay from the log rather than executing the functions. The level of the selected functions plays an important role in overhead, because the lower the level is, the more information there is that needs to be recorded. In contrast, the higher the level is, the less detail there is that can be replayed for root-cause analysis, since those selected functions are not executed at all (i.e., their execution is “fast-forwarded”). The latter case may reproduce some failure symptoms without reproducing the bug, if the buggy code is “fast-forwarded” during replay [32].

3.3.11  PRES
PRES [65] is a record/replay technique to help reproduce bugs on multiprocessors. PRES (Probabilistic Replay with Execution Sketching) aims at reducing the number of attempts to reproduce the bug, but relaxing the idea of replaying it at the first try. By doing this, PRES can minimize the recording overhead during production runs, albeit bringing a little increased bug replaying time during diagnosis. However, assuming that diagnosis is done offline and automatically, this trade-off can probably be well tolerated by programmers.

The authors make also another pertinent observation: as long as the occurred bug can be reproduced, it is less important for programmers to reproduce it with precisely the same execution paths seen in the original execution. Thereby, during production runs PRES logs only partial execution information into a sketch. An intelligent partial-information replayer will use this sketch later to reproduce the bug via multiple attempts to reconstruct the missing information necessary for reproduction.

In particular, PRES operates in three stages:

- **Production run**: records only relevant events in an execution sketch, which will be then sent to the developer site if a bug occurs, (the authors do not assume any privacy issues). PRES instruments the code using Intel's tool Pin and employs 5 different techniques of sketch recording:
  - **SYN**: records the global order of synchronization operations (recording points at the return point of lock and unlock operations).
  - **SYS**: records the global order of system calls in addition to the global order
recorded with SYN (recording points after the return of system calls).

- **FUNC**: records the global order of function calls (recording points at the function entry and return).

- **BB**: records the global order of basic blocks (recording point at the beginning of each basic block that does not contain any form of jump instruction).

- **BB-N**: optimizes BB method by recording only every \( n \)th basic block.

For testing purposes, the authors have also implemented another scheme, called RW. This scheme records the global order of accesses to the same shared variables from different threads, like other previous multiprocessor deterministic replay systems.

- **Reproduction phase**: automatically repeats the multiple attempts of replaying the program until the bug is revealed. After each failed replay attempt, feedback is generated to improve future tries. To re-execute the program, PRES uses a module named PI-Replayer that consults either execution sketches or feedbacks for previous replay attempts at every non-deterministic point. Alongside, a Monitor controls each replay, searching both for executions that do not match, at some point, the original sketch (here the execution is stopped and feedback is generated) and the moment at which the failure is reproduced.

- **Diagnosis phase**: PRES intelligent replayer leverages complete information from previous stage to reproduce the bug with 100% certain during each replay.

The obtained results show that sketching methods can reduce significantly the logging overhead during record phase, and allow the bug reproduction with high probability within an acceptable time.

![Figure 5 Example of exploratory replay in PRES (Source:[36])](image-url)
3.3.12 Castro et al.
The reference in terms of privacy techniques in replication of faults is the work of Castro et al. [13]. This technique assumes that the application is being continuously monitored, and that a complete record of its execution is made so that it can be replayed in the future. When there is a fault, the program is re-executed symbolically - traversing the same paths of execution that led to the error - in which the input is replaced by symbols. These symbols are being constrained in each branch of the program, which enables the system to calculate, as the final result, all the conditions that the input data must meet for the execution path to be the same as that which led to the failure. Thus, using an SMT (Satisfiability Modulo Theories), you can create a new input that travels the same path of execution - and thus lead to the same flaw – as the original input, but that is independent from the original input. However, for paths that have a narrow field of possibilities or with only one value, there is a loss of privacy of user data. This loss is measurable through an entropy measure of the new input, which is presented to the client when it is asked for permission to send the error report with new input. The main disadvantages of this technique has the great disadvantage are the large computational burden and the fact that it may not guarantee the privacy of user data.

A very similar technique was performed in Camouflage tool [19], which maintains the same approach but using an extension of JavaPathFinder [8] to generate the conditions on the input and YICES [25] to solve and create a new input. It applies only to Java programs while Castro technique can be used on any application that runs directly on processors of the x86 family. The great disadvantage in these two approaches is the great performance cost it imposes on the client. This is due to the monitoring software that has a constant cost on the application, as well as to the symbolic re-execution, which is computationally heavy.

3.4 Statistical Debugging Solutions
In this section, we will give an overview of some solutions developed to provide statistical debugging.

3.4.1 CBI
CBI [52][53] was one of the first systems to employ statistical debugging. CBI (Cooperative Bug Isolation) is a sampling infrastructure for gathering information software executions produced by its user community. After collecting information, CBI performs an automatic
analysis of that information to help in isolating bugs.

CBI is based on sampling, that is to say, monitors information only from time to time. This brings the benefit of having a modest impact on the performance of the user's program. However, given that some bugs occur rarely, it becomes more difficult to track them. In other words, one needs to guarantee that the sampling is statistically fair, so that the analysis is consistent with the happening events. CBI addresses this by using a Bernoulli process to do the sampling.

The information regarding program runs is collected via *predicate profiles* from both successful and failing executions. Predicate profiles are particular points of the program that are instrumented to provide data about their values. There are three categories of predicates logged:

- **Branches**: for every conditional, there are two predicates to track whether the true or false branch was taken, respectively.

- **Returns**: at each call point of functions which return scalar values, there are three predicates to track whether the return value is smaller than 0, larger than 0 or equal to 0.

- **Scalar-pairs**: at each scalar assignment $x = \ldots$, identify each same-typed in-scope variable $y_i$ and each constant expression $c_j$. There are three predicates to track whether the new value of $x$ is smaller, greater or equal to $y_i$ and $c_j$, respectively.

The data gathered across multiple executions of the program is integrated into feedback reports. Conceptually, the feedback report for a particular execution consists of a bit-vector, with two bits for each monitored predicate (observed and true). These observed bits indicate whether the predicate was ever observed, while the true bit state whether the predicate, if observed, was ever true. In addition, there is a one final bit representing overall execution success or failure.

This approach has the advantage of producing always the same size of data independently of the sampling density or running time. Unfortunately, this implies a significant loss of information, since the order of observations is not recorded.

CBI's automatic bug isolation process proceeds with the statistical analysis of the information gathered in order to pinpoint the likely source of the failure. Given that many of
the logged predicates are irrelevant, CBI assigns a score to every predicate to identify the best failure predictor among them. The predictors are scored based on sensitivity (accounts for many failed runs) and specificity (does not mis-predict failure in a successful execution). Using these metrics, CBI selects top predictors.

One of the main problems with CBI system is that it relies on a code duplication-based instrumentation scheme that effectively doubles the size of the program. Such a large increase in code size may not be suitable in practice for some applications. Moreover, although CBI provides a regularizing logistic regression to isolate a memory corruption error (a crash in version 1.06 of the GNU implementation of `bc` [53]), it does not address another non-deterministic bugs.

### 3.4.2 Holmes

HOLMES [16] is statistical debugging tool that isolates bugs by finding paths that correlate with failure. Inspired by previous work of CBI, HOLMES goes one step forward in statistical debugging by investigating the impact of using path profiles to improve the accuracy of bug isolation. It is based on the observation that paths are a natural candidate for debugging as they capture more information about program execution behavior than predicate profiles. For instance, paths can provide more context information on how the buggy code was exercised which help the task of debugging, while predicates can only locate the point in code where the error occurred.

HOLMES can operate in two modes:

- **Non-Adaptive Debugging**: this mode implements CBI's statistical debugging algorithm using path profiles instead of predicate profiles. Like previous work, HOLMES instruments the program and collect path profiles information during program runs, which is then aggregated in feedback reports. The feedback reports have the same structure as those of CBI. In the next step, gathered paths are assigned numeric scores to determine the top predictor of bug from the set of all available paths. These scores follow also the metrics specified in CBI approach.

- **Adaptive Debugging**: is a mode that arises from the fact that in large programs, usually only a small fraction of the code is buggy and thus relevant to debugging. Contrary to sampling, HOLMES adaptive technique starts with no instrumentation. In the initial phase, HOLMES receives only bug reports, which consists of a stack
trace and partial state of the program at the point of failure. After obtaining an enough number of bug reports, HOLMES employs static analysis to point out portions of code that more likely contain the causes of the failure. Then, these portions of code are instrumented to monitor useful information and collect detailed profiles and then are redeployed in the field. Because only important parts of the code are instrumented, HOLMES avoids the need for sparse random sampling. The process repeats, but this time HOLMES collects partial profiles in place of bug reports. These profiles are later analyzed using the same techniques as in the non-adaptive mode. The analysis compute a set of bug predictors and if some of them are strong enough to explain the failure, then the iterative process ends (a predictor is classified as strong if its score exceeds a defined threshold and weak otherwise). If that does not happen, HOLMES expands its search by using static analysis and bug predictors to identify other parts of code that are closely related to the weak predictors. In practice, this consists in identifying a set of functions that interact with weak predictors to be profiled in the next iteration. This iterative process carries on until strong predictors are found and all bugs have been explained.

This automatic scheme is attractive for software maintenance, as it avoids the tedious manual task of selectively replace client binaries with instrumented versions in order to collect more information about the problem. Therefore, developers can focus exclusively on fixing bugs.

However, weak predictors can be sparse. Hence, given that HOLMES explores only near weak predictors, it is possible to be stuck with no new sites available to explore.

3.4.3 3.4.3 3.4.3 3.4.3

Unfortunately, all the previous approaches described are not suitable to track concurrency bugs. These kind of bugs arise from the non-determinism inherent to operations involving multiple threads. Thus, they cannot be captured by predicates or profiles used in prior work, which focus only on one thread at a time. Thereby, new research has been done to address concurrency bugs with statistical debugging. The Cooperative Concurrency Bug Isolation (CCI) is the first to tackle these issues.

CCI [42] is a low-overhead instrumentation framework to diagnose production-run failures caused by concurrency bugs. CCI works by recording specific thread interleavings during the original run, using then statistical models to identify strong bug predictors among the
information recorded. This approach is built upon CBI philosophy, so CCI also leverages sampling to keep low overheads in production runs and relies on statistical models to discover the root causes of the failure.

However, unlike CBI, CCI strives to find causes of concurrency bugs. This implied the need for new techniques of sampling that address the non-deterministic challenges of these kind of errors. For instance, CCI sampling may require cross-thread coordination, because concurrency bugs involve multiple threads. Moreover, it must also keep each sampling period active for some time, because concurrency bugs always involve multiple memory accesses.

Thereby, CCI consider three different instrumentation schemes, that offer different tradeoffs between performance and failure-predicting capability:

- **CCI-Havoc**: tracks whether the value of a memory location is changed between two consecutive accesses from one thread. This captures the change of program states in the view of one thread at two nearby points, and may help to diagnose atomicity violations.

- **CCI-FunRe**: tracks function re-entrance: simultaneous execution by multiple threads. This captures the simultaneous execution by multiple threads, and may help to diagnose errors arising from misuse of thread-unsafe functions.

- **CCI-Prev**: tracks whether two consecutive accesses (read or write) to one memory location come from the same thread or distinct threads. This captures interactions among multiple threads at a fine granularity, and may help to diagnose data races and atomicity violations.

In order to maintain correctness and good coverage with the previous schemes, CCI has to make some decisions with reference to the sampling style. The following tradeoffs must be taken in account:

- **Thread-coordinated vs independent sampling**: when predicates only work upon data from single threads, the decisions regarding when to start/stop the sampling can be independent. On the other hand, some predicates require tracing multiple thread information, thus need a thread-coordinated sampling. Considering the three schemes mentioned before, one can say that CCI-Havoc and CCI-FunRe require thread-independent sampling, while CCI-Prev needs thread-coordinated sampling.
• Length of each sampling period: CCI predicates must consider multiple executions points together, so the sampling period must be large enough to cover related accesses to shared memory locations, but not too long, otherwise the overhead will degrade performance. According to the length of sampling period, one can refer that CCI-Havoc and CCI-Prev require bursty sampling and CBI-FunRe non-bursty.

• Correctness of predicate evaluation: monitoring interleaving patterns often needs previous information, such as the history of preceding access to some variable. Unfortunately, sampling does not provide a complete history of program execution. Because of that, correct analysis of certain predicates requires a mixture of both unconditional and sampled instrumentation. In CCI-Prev case, CCI has to differentiate data from earlier sampling periods from that collected during current period, in order to guarantee correctness. It achieves this using a generation counter that is incremented to distinguish fresh from stale entries in CCI-Prev's hash table. For CCI-Havoc, the process is similar to CCI-Prev, since this scheme also guarantees correctness as long as fresh information is used. The difference is that CCI-Havoc uses thread-local hash tables in place of a global hash table. On the other hand, CCI-FunRe requires that all invocations to a certain function be instrumented. This because sometimes fresh information indicating that there were no concurrent executions of a function \( F \) may not be completely true, given that a non-sampled invocation to \( F \) may have occurred before the current sampling period began. CCI addresses this by performing an always active instrumentation for all function invocations and only apply sampling to predicate evaluation and recording.

The evaluation results for CCI showed that sampling significantly decreases monitoring overhead. Most of the runtime overhead experienced was lower than 10% for the applications tested. However, for memory-access intensive applications, the instrumentation schemes still incur very high monitoring overheads.

3.4.4 Conclusions
As exemplified in this section there is a significant body of work tackling the problem of fault replication. The most sophisticated solutions leverage advance techniques such as code analysis and execution synthesis. The most significant challenge remains the performance overhead induced by application monitoring and in some cases by symbolic execution or the generation of anonymized inputs. Therefore, identifying domains where the performance of
fault replication can be improved is one of the greatest challenges for the research community in general and the FastFix project in particular.

Another important open area of investigation is on the design of efficient fault replication mechanisms for concurrent applications running on top of (nowadays mainstream) multi-core systems. As previously highlighted, in fact, conventional software based approaches [49][61] incur in overheads that are unacceptable for production systems. More recent approaches, such as [32][65], appear more attractive from the performance perspective, requiring substantially less logging activities. On the other hand, as acknowledged in these works, the effectiveness of these techniques in reproducing buggy behaviors can be affected by the nature of the faults and by the choice of the recording points.

Finally, most of the existing recording/replay techniques for multi-core systems have been designed to be executed on a single machine. Achieving decoupling between the recording and the re-execution phases poses the additional issue of being able to replicate in lab the environmental conditions of the production run. Virtual machine technologies can certainly simplify this task, but this would require extending them to include efficient replay mechanisms coping with the additional sources of non-determinism proper of multi-processor systems. To the best of our knowledge, commercial VM solutions (such as VMWare [79]) only permit re-playing execution of a uniprocessor VM.
4 Existing Systems and Applications for Test Automation

In this section, we introduce state of the art and tools related to Test Automation. Section 4.1 will deal with the automation of test execution while section 4.2 is related to the automatic generation of test cases. These testing related tools are important auxiliaries for FastFix. Once faults are reported to the developers of a FastFix application, the FastFix platform will attempt to automatically correct the application with a patch. However, if human intervention is needed, the ability to revise automatic tests to the application or to inject tests into it, may be critical to identify and correct errors.

4.1 Automatic Test Execution

In this section, we present approaches and tools for automatic test execution. The most famous of this family of tools are probably JUnit for Java and, NUnit for the .Net platform.

4.1.1 JUnit

JUnit is a testing framework for the Java language which allows testers to specify test cases as well as conclusions (pass or fail) that should come out of each case. A great attraction to JUnit comes from the fact that it is well integrated to the language as test cases are implemented in Java classes. This allows to write test cases in Java themselves and also to test the raise of exceptions.

4.1.2 NUnit

NUnit is to the .Net framework what JUnit is to Java. It relies on the same principles and possesses similar features.

4.1.3 Selenium

Selenium is a testing framework dedicated to web-based application. One of its key features is that it allows for testing application on different web browsers. This makes it possible to avoid the writing of test cases for several web browsers, hence offering a means to test the portability of the application over different platforms.

A second key feature of Selenium is the ability to record interactions between the user and the application and to automatically replay these interactions. This facilitates test case
generation process by automating code creation from the recorded interactions.

4.2 Automatic Test Case Generation

In the following, we introduce concepts and approaches for automatic test case generation. As mentioned in [71], “Test generation is the process of deriving a set of test cases from a formal specification”. In other terms, automatic test generation is usually based on specifications or models. Therefore, this survey focuses on model-based test generation approaches. As such approaches have recently been surveyed in [1], we present the results introduced in that work in the following.

Model-based testing approaches may rely on very different types of models. In this section we describe approaches based on contracts, state machines, labeled transition systems, temporal formulas, abstract data types and data flow models.

4.2.1 Contract-like Specifications

Contracts represent conditions that can be formally expressed regarding variables of a program. Typical examples of contracts in code are assertions and branching conditions (e.g. conditions in an IF, WHILE or FOR statement). Contracts can also be specified through annotation in the code such as with the JML language for Java.

JET (see e.g. [11]) is a testing program that relies on JML annotations. In order to test a function, JET randomly picks values and check they fulfill the precondition given by the JML annotations before executing the function. Then the resulting state of the program is checked against the post-conditions associated to the function. If the post-condition is not fulfilled, then an inconsistency is detected.

A similar concept is used by the T2 tool (see e.g. [67]), which is also dedicated to Java. However, T2 selects default values for objects to be created and reads specification in plain Java (typically through assertions). T2 can randomly generate sequences of methods to be tested.

In [31] the authors introduce DART (Directed Automated Random Testing) is an automatic test generator that aims to cover different paths of a program. During execution, DART creates symbolic constraints (linear integer arithmetic) in order to generate values that allow to take some paths over other path. CUTE for C code and jCUTE for Java code build on DART in order to also deal with approximate pointer constraints. EXE (see e.g. [11]) is
another such extension for C code that allows dealing with bit vector arithmetic. Pex is a Microsoft product based on symbolic execution in order to generate test data ([75]). Therefore, it relies on similar principles as DART and CUTE. However, Pex is dedicated to the .NET framework.

Microsoft also developed SpecExplorer ([78]) extracts test cases from an automaton build automatically from Spec# specifications. SpecExplorer possesses several strategies to explore and select paths during test execution. These tests can also be driven by specifying actions to be taken following some rules.

Finally DOTgEAr (Dataflow-Oriented Testcase-generation with Evolutionary Algorithms) was developed by Norbert Oster as his PhD thesis. DOTgEAr applies to Java programs and aims to generate test cases following several criteria such as minimizing the number of test cases and maximizing coverage.

4.2.2 Labeled Transition Systems, (E)FSM, State Charts

Labeled Transition Systems (LTS) are like Finite State Machines with the difference that they may possess an infinite set of states or an infinite alphabet (set of events). They can be used to model the possible behaviors of the system to be tested. LTS approaches are usually considered with the IOCO conformance. This conformance describes the expected actions following a given sequence of actions of the system.

In [40], the authors describe TGV, a test generation tool based on LTS. From an LTS modeling some specification of the system under test and a test objective, TGV automatically generates test cases for IOCO. If all the test cases are executed and successful, then one can conclude the system under test conforms to its specification for IOCO and for the set of traces represented by the test objective. As exhaustive testing is usually unpractical, the test objective aims to reduce the set of traces on which the test are meant to focus.

In [18], STG is introduced, a tool build on the principles from TGV. The main difference between TGV and STG is that STG makes it possible to handle Symbolic Transition Systems, i.e. LTS that are extended so that variables can be taken into account in the model.

AGATHA ([6]) is another tool for symbolic test generation. It was used on industrial system such as in avionics and can read specifications written in different languages such as UML, STL and STATEMATE.
TorX ([76]) is a random test generator that evaluates IOCO conformance. It creates test cases on the fly, i.e. during execution of the tests.

Autolink ([45]) is a test generation tool integrated to the Tau tool set. Test generation relies on state exploration with a test objective. The state exploration can be performed following an exhaustive or random way. Autolink produces test cases in the TTCN language, a programming language designed for testing communication protocols.

Finally TDE/UML is a test generation tool from Siemens Corporate Research ([33][34]). TDE/UML uses UML models to which requirements about coverage and constraints. From there tests can be generated for Java and C++. However, these tests cases are not executable and need to be implemented in some other language such as JUnit.

### 4.2.3 Testing through model-checking

As exhaustive testing is rarely applicable, it is then crucial to be able to define relevant parts of the system under test to be tested. The ability of model-checking to provide counter examples can then be used to determine test cases.

FShell ([38]) is a framework for testing C programs. It uses model-checking techniques to determine inputs that fulfill structural constraints of the program as well as coverage criteria. FShell relies on a SAT solver called CBMC.

As explained in [36], UPPAAL is a model-checking tool performing conformance testing. More specifically UPPAAL can generate tests for real-time systems.

### 4.2.4 Fault Injection

Fault injection is a validation method for which many different techniques have been developed over the years. Fault injection exhibits the occurrences of faults in a system in order to evaluate its fault handling mechanism. Fault injection has traditionally been applied to hardware systems but has more recently also been applied to software systems.

Software faults can be injected at either compile-time or runtime. Compile-time fault injection consists of modifying the source code of the application in order to emulate hardware or software fault. Runtime fault injection requires a mechanism for triggering. Such mechanisms may be time-outs, exceptions and runtime code insertion.

The time-out approach consists of raising interrupt event after a given time and give control to the fault injection. Similar to the time-out approach, the exception approach generates an
interruption and invokes the fault injection mechanism. However, the conditions under which the interruption occurs are not just related to timing. Finally, runtime code insertion is an approach that allows for addition of instructions into the code invoking errors at runtime.

Examples of injected faults can vary from manipulating the contents of CPU registers or the operation codes or operands of machine instructions.

Currently, there are some fault injection tools available. Xception ([12]) for instance is a tool that relies on some features that can be found in several standard processors. No modification of the software is required in order to use this tool. FERRARI ([41]) is another fault injection system based on the UNIX operating system. FERRARI can emulate faults by modifying the program state during execution. FIRE ([55]) and Jaca ([56]) can monitor C++ and Java applications as well as inject faults, and the PROPANE (PROPagation ANalysis Environment) tool ([37]) can inject faults in C-code. The code needs however to be instrumented so that traces of data can be logged. PROPANE introduces software faults using mutation techniques manipulating variables and memory contents. Finally TroubleMaker is a fault injection tool for applications running on Linux systems. It automatically and randomly selects scripts that then run introducing errors in the system. Although some existing predefined scripts are available, more scripts can be implemented and tailored to specific purposes.
5 Shortcomings of Current Solutions

Unfortunately, the main issue with hardware-assisted solutions is that, despite recent works have suggested optimizations to reduce hardware complexity [55][21], they still require significant hardware modifications, none of which exists today, except in simulations. A major issue that affects existing fault replication mechanisms is that, in order to be able to reproduce a fault occurred in a production system, the user must consent to send detailed information concerning the state of the locally deployed application which can raise serious privacy concerns. Standard data obfuscation techniques [4] are clearly not an option, given that even minimum “blind” alterations of the information used to drive program re-execution may compromise the accuracy of the replay process, failing to reproduce the faulty behaviour. Or, even worse, corrupting the trace and completely preventing the re-execution. Another important open area of investigation is on the design of efficient fault replication mechanisms for concurrent applications running on top of (nowadays mainstream) multi-core systems. Finally, most of the existing recording/replay techniques for multi-core systems have been designed to be executed on a single machine. Achieving decoupling between the recording and the re-execution phases poses the additional issue of being able to replicate in lab the environmental conditions of the production run. Virtual machine technologies can certainly simplify this task, but this would require extending them to include efficient replay mechanisms coping with the additional sources of non-determinism proper of multi-processor systems. To the best of our knowledge, commercial VM solutions (such as VMWare [79]) only permit re-playing execution of a uniprocessor VM.
6 Open Research Questions

The development of novel execution sketches logging schemes, based on the automatic extraction of semantic models of the applications, is expected to significantly increase the probability of correctly reproducing the occurrence of faulty behaviour, while minimizing the amount of logged information, and hence the overhead during production runs.

Finally, the choice of integrating the re-execution logic within state of the art Virtual Machine platforms (such as the Xen hypervisor [5]) will permit to avoid intrusive, and normally problematic, OS customizations, while permitting to run applications unmodified and at near native speeds.

By introducing obfuscation-based privacy preservation mechanisms, the fault replication platform developed within the FastFix project will be adoptable in a much wider range of realistic scenarios. These include, for instance, e-Health applications, where the sensible nature of the data managed by the application may otherwise prevent the disclosure of fault related information. This is also the case of common, widespread applications that happened to be in use by independent, competing industrial organizations which may refrain from reporting detailed failure information unless provided with strong guarantees concerning the absence of private information leakages.

In order to avoid privacy concerns associated with the release of failure information, we will investigate novel data obfuscation techniques, which will preserve the program re-execution's accuracy. To this end, we will study analysis techniques, operating at both the source code and binary level, aimed at identifying the set of data alterations (e.g. input scrambling) viable without altering the recorded application's control flow. A starting point for our investigation will be some recent results in binary code [64][10] and source code [26][84] analysis. These techniques are currently used to automatically extract structural information on the application's structure (e.g. identifying the subset of the current application state that could potentially affect the subsequent execution, or how the values of given variables may affect the control flow). We plan to leverage on analogous techniques to derive a semantic model of the information logged by the failure dump. This model will then be used to derive which subset of the failure information, how, and to what extent, could be obfuscated without compromising the accuracy of the replay phase.
These analysis techniques will be used also to gather knowledge concerning the structure of the application structure and derive cost/benefits models that will drive the selection of the recording points to use in an execution sketch, so to maximize the effectiveness of some of the most advanced approximate checkpointing techniques such as PRES [65] and R2 [32].

In order to achieve portability (e.g. by avoiding OS-level patches), application transparency and facilitate remote replaying, we aim to integrate, to the best of our knowledge for the first time, the above described execution replay techniques within a virtual machine capable of ensuring deterministic replay of multi-threaded applications in unmodified multiprocessor (and multi-core) systems.
7 Conclusion

This document sets the context of the challenge of fault replication by describing the fundamental concepts and existing solutions. In particular, the challenge of deterministic replay of non-deterministic replay is discussed in great detail and the main challenges highlighted: the performance overhead of monitoring applications, the users’ privacy concerns and the difficulty of accurately recording multi-threaded and multi-core applications.

Although outside the scope of FastFix, we demonstrate that hardware solutions although they exist in theory and simulation, the cost and complexity of dedicated hardware is a major hurdle to their adoption. The document also describes the most relevant software-based approaches including those using a statistical debugging methodology.

A panorama of the state of the art regarding fault injection and automatic test generation is also presented. These are techniques, which enable the collection of meaningful data for automatic model design, and that are likely to become important complements to the fault replication and self-healing functionalities of the FastFix platform.

We highlight the main open research issues made obvious by gaps in the state of the art and demonstrate and summarize how FastFix aims to tackle them.
Bibliography


D5.1: State of the art in fault replication and test automation


[77] TWO, M. C., POOLE, C., CANSDALE, J., FELDMAN G, NEWKIRK, J.,


