Scalable Computing: Practice and Experience
Volume 13, Number 1, pp. 5–20. http://www.scpe.org
ISSN 1895-1767
© 2012 SCPE

FASTFIX: A CONTROL THEORETIC VIEW OF SELF-HEALING FOR AUTOMATIC CORRECTIVE SOFTWARE MAINTENANCE

B. GAUDIN[*], M.H. HINCHEY[*], E. VASSEV[*], P. NIXON [†], J. COELHO GARCIA [‡] AND W. MAALEJ [§]

Abstract. One of the main objectives of self-adaptive systems is to reduce maintenance costs through automatic adaptation. Self-healing is a self-adapting property that helps systems return to a normal state after a fault or vulnerability exploit has been detected. The problem is intuitively appealing as a way to automate the different type of maintenance processes (corrective, adaptive and perfective) and forms an interesting area of research that has inspired many initiatives. As a result, several surveys on self-healing have been published to describe the state of the art in this field. According to those surveys, the major trend towards finding a solution of the self-healing problem relies on redundancy that may concern both architecture and code resources. These approaches are therefore better suited to address adaptive and perfective maintenance. As part of the EU FP7 FastFix project [1], we focus on self-healing for corrective maintenance. We propose a framework for automating corrective maintenance that is based on software control principles. Our approach automates the engineering of self-healing systems as it does not require the system to be designed in a specific way. Instead it can be applied to legacy systems and automatically equip them with observation and control points. Moreover, the proposed approach relies on a sound control theory developed for Discrete Event Systems. Finally, this paper contributes to the field by introducing challenges to the effective application of this approach to relevant industrial systems. Some of these challenges are currently being tackled within FastFix.

Key words: software maintenance, self-healing, software control, context-aware software engineering

1. Introduction.

Software maintenance aims to modify software systems after they are deployed in production ([39, 14]). In [47], the authors divide maintenance activities in three different types: adaptive, perfective, and corrective. Adaptive maintenance is performed to make the computer program usable in a changed environment. Perfective maintenance mainly tackles performance and maintainability issues. Corrective maintenance is performed to correct faults. Within the last 20 years the complexity of both software and communication infrastructures has increased at an unparalleled rate. This level of complexity means that software systems are more prone to unexplained faults, require more support and maintenance, and cost more to deploy and manage. A fundamental challenge faced by the software industry is how to ensure that these complex systems require less maintenance and human intervention. Concepts such as self-healing, autonomic and self-adaptive systems provide an answer by reducing human intervention and reducing the apparent complexity of systems.

Several surveys on self-healing have been published to describe the state of the art of this field. According to these surveys, the major trend towards finding a solution of the self-healing problem rely on redundancy that may concern both architecture and code resources. These approaches assume that systems are designed with adaptive capabilities and are therefore better suited to address adaptive and perfective maintenance. In this article, we focus on self-healing for corrective maintenance. Section 2 recalls existing works on self-healing, automatic diagnosis, and automatic repair of software systems.

We also propose a control theoretic approach to self-healing in order to deal with corrective maintenance. Control makes it possible to drive the system in a range of desired behaviours. It represents an interesting approach to avoid behaviours leading to failures. This is achieved by dynamically disabling some of the implemented features, depending on the current execution of the system. Moreover, the proposed approach automatically synthesizes supervisors in charge of controlling the software. This hence automates the computation of a new suitable range of software behaviours whenever corrective maintenance needs to be performed, e.g. a failure has been reported and behaviours leading to this failure need to be removed or avoided. Our approach consists of a pre-deployment and runtime phase. Each phase is described in Sect. 3. Sect 4 illustrates our approach through one of the case studies considered in FastFix: the Moskitt application.

Finally, challenges to be tackled in order to implement effective and efficient control theoretic self-healing features are discussed in Sect. 5. Most of these challenges relate to the supervisory control theory and its applicability to software system. However we also show that some challenges are common to the research in automatic diagnosis and automatic repair.

[*]Lero - The Irish Software Engineering Research Center, Limerick, Ireland ({benoit.gaudin, emil.vassev@lero.ie, mike.hinchey@lero.ie}@lero.ie).
[†]University of Tasmania, Hobart, Australia (Paddy.Nixon@utas.edu.au)
[‡]Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa, Lisbon, Portugal (jog@gsd.inesc-id.pt)
[§]Technische Universität München, Munich, Germany (maalejw@cs.tum.edu)

2. Overview of Automated Software Maintenance Approaches.

2.1. Self-Healing. Self-healing is a concept which aims to tackle or prevent maintenance tasks in presence of system failures. This concept came with the notion of Autonomic Computing initiated by IBM in [32]. The main goal of Autonomic Computing is to reduce human intervention in component management.

There are different visions of self-healing in the literature. In [52], Rodosek et al. consider self-healing as equivalent to self-repairing and self-immunity, i.e. the ability to resist to infections. A self-healing system must be able to recover and go back to a proper state following some disturbance. This view is shared in [26], where recovery oriented computing is presented as a key aspect of self-healing. The authors consider that healing systems are more concerned with post-fault or post-attack states and more specifically with bringing the system back to a normative state. In [34], the author has a broader view of self-healing. In that work, self-healing tries to identify and eliminate or mitigate the root cause of the fault. In [37], the authors have a similar view of self-healing and recall that the system requires knowledge about its expected behaviours in order to automatically discover system malfunctions or possible futures failures. In [53] the authors describe self-healing as consisting of self-diagnosis and self-repair. The consequence of this observation is that work related to automatic diagnosis and automatic repair or recovery are relevant to the field of self-healing. Finally although self-healing aims for automation, as discussed in works such as [26, 50], even non fully automated healing approaches may also represent self-healing techniques. In [26] for instance, Ghosh et al. introduce the notion of assisted-healing for systems that require some human intervention during their healing process.

Historically, self-healing techniques were inspired by fault-tolerance and these two fields are tightly connected as explained in [15]. This entails that, as for fault-tolerance approaches, self-healing solutions often rely on some system redundancy, such as components, hardware, network nodes, code variants, etc. This observation was also made recently in [45] which also provides a survey on self-healing approaches. The authors classify self-healing techniques and faults tackled. Self-healing techniques can be classified according to the type of systems under consideration and span over service relaunch, checkpointing, architecture based, model based, multi-agent based, reflection based, aspect oriented programming, service discovery and load balancing. From [45] again, the faults tackled by self-healing approaches are classified into crash failure, fail-stop (execution is deliberately inhibited on a failure and detected by other processes), omission (message loss or transmission error), transient (error related to presence of various self recovering faults disturbing other parts of the system), timing and performance (constrained distributed synchronous execution of tasks by a specific amount of time), security and arbitrary (a process confuses the neighbors by providing constantly individual consistent but contradicting information). As pointed out in [55], self-healing is still a relatively immature field and the class of faults tackled by this field remains quite narrow. Moreover, the techniques employed in this field mainly use architecture adaptation in order for the system to provide expected features even without the faulty component. To this respect, these approaches achieve adaptive maintenance in presence of failures.

Recently, other self-healing approaches modifying the behaviours in order to correct them have been considered, e.g. [56, 7, 8, 6]. Therefore these approaches propose self-healing features that are suitable for corrective maintenance.

In [56], the authors present ASSURE, a self-healing approach based on rollback and error handling facilities. When an error occurs at runtime and the system is brought back to a rescue point, pre-defined error handling strategies are executed in a virtualized environment and tested. If it is satisfactory, then the error handling code is applied to production code, modifying the initial system behaviour in order to correct it. This approach makes it possible to self-heal from unknown issues by applying recovery approaches for known issues, that also seem to apply to the unknown ones.

Carzaniga et al. [7, 8, 6] consider a self-healing approach that modifies the behaviour of the component to be healed. The proposed approach, called workaround, consists of replacing a faulty sequence of operations with another that produces the same outputs or effects. Workaround is a model based approach which provides alternative program executions to the failing ones. This approach relies on the observation that libraries often contain feature redundancies. A typical example, provided in [6] is the one of changing an item in a shopping basket. The change item feature can be achieved by composing the remove item and the add item features, i.e. an item change can be seen as the removal of an item followed by the addition of another one.

As explained in [53] self-healing can be seen as a combination of self-diagnosis and self-repair approaches. A broader view of these concepts are *automatic diagnosis* and *automatic repair* which are strongly related to the corrective maintenance process.

2.2. Automatic Diagnosis. Diagnosis is a *proactive* software-maintenance technique driven by *detection* and *isolation* of faults to prevent failures [47]. *Automatic diagnosis* targets the automation of the diagnosis process, where faults are detected and isolated by the system itself, often by applying techniques working on the system architecture or by implementing special *alarms*. The *architecture-based techniques* usually rely on resource redundancy. For example, in [54] is considered the feasibility for a multi-processor system to perform self-diagnosis on some of its processors by some others. Another class of automatic diagnosis techniques is the so-called *correlation-based diagnosis* that considers diagnosis that may provide several types of alarms where an issue is detected by the raise of one or more alarms. In [16] correlation-based diagnosis is applied to Discrete Event Systems and considers the detection of alarms whenever they are not directly observable. In another approach metrics related to system states or performance are correlated as a means for diagnosis [31]. Normal system behaviour is determined by specific metric correlations and faults might be detected whenever there is a deviation from these metrics correlations. Finally, model based techniques form another class of automatic diagnosis. System models take as input some observations of the current system state or behaviour and produce diagnosis. In general, the model based diagnosis is about comparing a system behaviour with actual observed executions [51]. When the observed execution deviates from the expected behaviour provided by the model, this is an indication of a fault occurrence. [29] consider probabilistic models in order to apply this principle.

2.3. Automatic Software Repair and Bug Fixing. Several approaches have been proposed to automate the bug fixing process. *Rollback techniques* maintain a record of "healthy" system states to allow a rollback to the last such state when a fault occurs. Once successfully rolled back to a *healthy state*, the system re-executes after applying certain changes to its input data or execution environment (see e.g. [46, 56, 60, 35, 5]). *Mutation techniques* rely on *Genetic Programming* concepts and are closely related to data structure linking and modification. The *data structure repair* approach [42, 17, 18, 21] uses structural integrity constraints for key data structures to monitor their correctness during the execution of a program. If a constraint violation is detected, then mutations are performed on the system data structures in order to transform them so that they satisfy the desired constraints. *Event Filtering techniques* are usually related to software *security* and *vulnerability*. They consist of automatically creating and detecting *signatures* or *patterns for malicious* attacks such as *control hijacking* and *code injection*. Then these signatures are used for a *filtering check*, so that such attacks cannot break through in the system anymore. Systems following this principles are *PASAN* ([59]), *FLIPS* ([41]) and *ShieldGen* ([13]). *Learning* and *probabilistic* approaches to automatic repair and bug fixing learn from past executions where bugs have been fixed. Applied fixes are stored and can be retrieved and applied again or used in order to infer other possible fixes. Systems such as *Exterminator* ([43]), *BugFix* ([30]) and *ClearView* ([44]) implement such a principle. Finally, in [61] the authors present AutoFix-E, an automatic code fixing approach based *Model Checking*. This approach considers contract violations as failures and calls existing functions whose postcondition fulfills the violated contract. Fix candidates are created from a set of fix templates and the behaviour models.

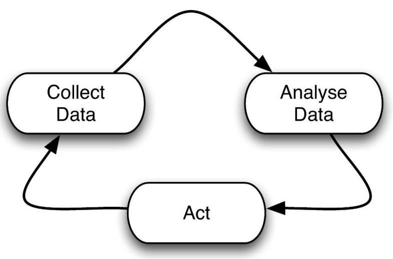
2.4. Conclusion. Self-healing approaches mainly rely on system redundancy which adapt their architecture in order to bypass faulty components but still provide their expected features. Therefore these approaches are related to adaptive maintenance, where the system adapts to changes due to failures.

The authors of [53] suggest a definition of self-healing consisting of self-diagnosis and self-repair. As diagnosis and repair techniques are very relevant to corrective maintenance, such a vision of self-healing is well suited to this maintenance task. However, automatic diagnosis and repair techniques found in the literature focus on analyses and lack of a unified and systematic approach for equipping the system with self-healing facilities implementing the autonomic feedback loop (see Figure 3.1(a)).

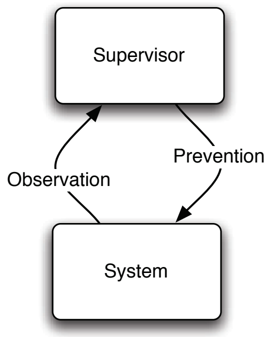
In Sect. 3 we propose an approach for software self-healing that automatically introduces autonomic facilities into an existing system, e.g. sensors and actuators. This approach is based on The Supervisory Control Theory (SCT) for Discrete Event Systems where the corrective maintenance task corresponds to the automatic synthesis of a supervisor. Section 5 introduces the main research challenges associated to the proposed approach.

3. A Control Theoretic Approach to Software Self-Healing. Regarding computing systems, control theory has traditionally been applied to data networks, operating systems, middleware, multimedia and power management ([28]). This section introduces a control-based approach for software self-healing.

Self-Healing is a property of Autonomic Systems [33]. Our approach proposes to automatically equip software systems with autonomic features before deployment so that they can follow the different phases of



(a) The autonomic feedback loop.



(b) The Control Feedback Loop.

Fig. 3.1: The runtime autonomic and control feedback loops.

the autonomic feedback loop presented in Figure 3.1(a). In particular, sensors and actuators are automatically added to the software system in order to realize the *Data Collection* and *Action* phases of Figure 3.1(a). Within FastFix, the *Analysis* phase corresponds to automatic supervisor synthesis. As explained in [20] control theory principles are suitable to implement the autonomic feedback loop. More specifically, this work considers the Supervisory Control Theory (SCT) on Discrete Event Systems. This theory was initiated in [48] and is a model based approach aiming to automate the synthesis of correct models.

Our self-healing approach consists of two different parts: a pre-deployment part which is performed before the system is deployed and where self-healing features are added to the software; and a post-deployment part corresponding to the automatic or semi-automatic execution of the maintenance process where the system self-healing features are employed. The latter part itself consists of supervisor synthesis and runtime supervision. Synthesis is applied using SCT, whenever new runtime system specifications need to be ensured, e.g. when a fault has occurred and behaviours leading to it must be removed. Runtime supervision corresponds to applying the synthesized supervisor to the application at runtime. Overall the presented approach can be seen as a three phase approach: pre-deployment, supervisor synthesis and runtime supervision. These phases are presented in more details in Sect. 3.1.

3.1. Overall Approach. The overall proposed approach is depicted in Figure 3.2. The left-hand side of this diagram represents the pre-deployment phase during which code is instrumented in order to introduce observation and control points (i.e. sensors and actuators) as well as data structures that make it possible for the application to embed and use supervisor models. A binary (or bytecode) application with these facilities can then be obtained through compilation. During the pre-deployment phase, a model of the behaviours is also automatically extracted from source code through control flows and method calls analysis.

During the runtime and maintenance phase, the software artefacts (source code or bytecode) are no further modified. Only models of a supervisor representing their possible runtime behaviours are manipulated in order to maintain the application behaviours within a desired set. These models are embedded in the application at runtime and are modified and replaced whenever an error occurs so that behaviours leading to this error cannot occur in future system executions.

Some unknown possible failures of the system may occur at runtime, requiring the application to be corrected. The observation of such a failure indeed indicates that the system behaviour is not satisfactory and needs to be modified. Self-healing capabilities aim to correct the system behaviour so that the observed failure can no longer occur. Such corrections are performed by modifying the supervisor that interacts with the application at runtime. Considering the Supervisory Control Theory introduced in [48], this can be automatically achieved when a control objective is provided. In some situations, this control objective can be automatically derived from observations of failures during the system execution [25]. In general, control objectives can also

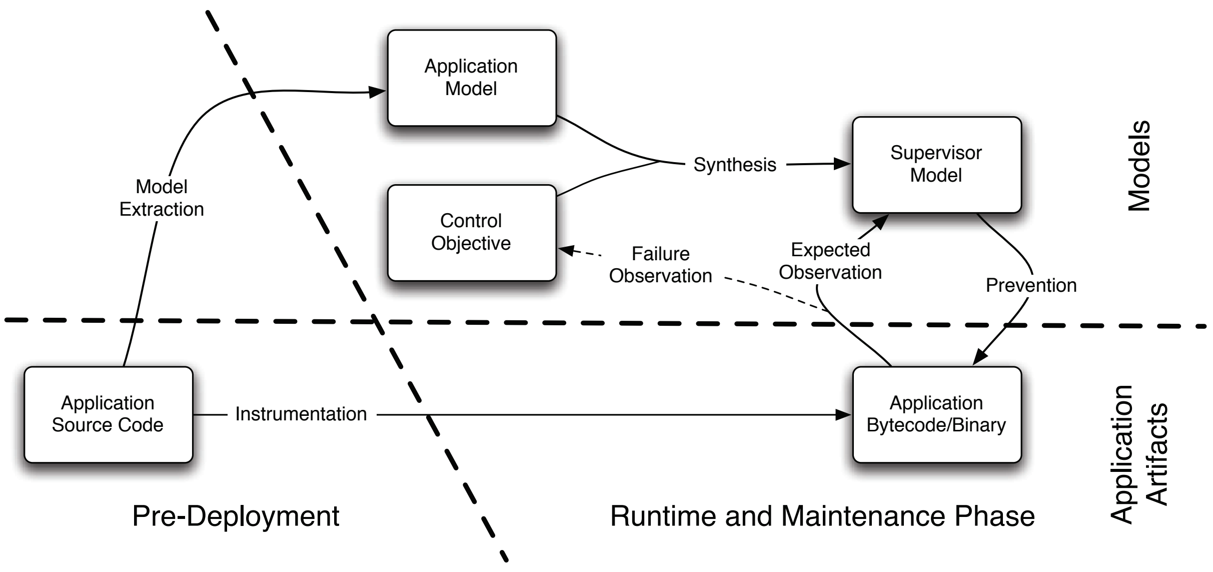


Fig. 3.2: A Software Control Approach to Self-Healing.

be provided by expertise. The accuracy and relevance of the expertise involved in designing a control objective will impact on the accuracy and relevance of the corrective solution applied to the system. For instance, diagnosis can help design a more accurate control objective. However, in cases where deep analyses and diagnostics cannot be conducted (e.g. when the amount of time that is necessary to perform this task is too long), a simple control objective excluding the undesired previously observed sequences of method calls can be submitted to the supervisor synthesis algorithm. However in this case, the resulting supervisor may act more coarsely and unnecessarily remove some of the system behaviours. This depends on how representative of an undesired behaviour the observed sequence is.

3.2. Pre-Deployment Phase. The pre-deployment phase aims to prepare the software application so that control and synthesis can be performed at runtime. This preparation consists of 2 subtasks: code instrumentation and model extraction. Each of these tasks is performed in an automated fashion.

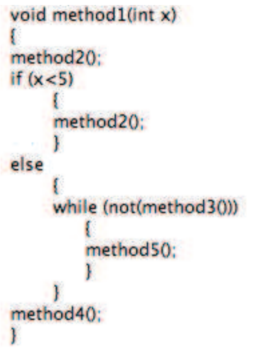
Code instrumentation is performed in order to introduce observation and control points as well as to embed a supervisor in the application, as illustrated in Figure 3.6. Intuitively, automatically instrumenting the application code consists of automatically embedding a supervisor into the system as well as adding conditional statements in each method body so that method invocations can be observed and executions of method bodies can be prevented at runtime [1].

Moreover the approach introduced in this section relies on the automatic design of a model of the application behaviours. In its basic form, this model can be a Finite State Machine whose transitions represent method calls. An over-approximation of the behaviours of the application can be obtained from the source code by considering methods, branching and loops as illustrated in Figure 3.3.

Some tools have been implemented in order to extract and analyze models represented as Extended Finite State Machines (EFSM), i.e. FSM associated with variables. PROMELA is an FSM-based modeling language. PROMELA models can be used as input to the SPIN tool, which can then model-check this model against some properties. Bandera ([11]) extracts FSM from Java code. Bandera offers the possibility of exporting the extracted models into the PROMELA format. More recently in [27], the authors proposed an efficient approach for model extraction from programs. The approach makes it possible to deal with different but syntactically similar programming languages such as C++ and Java.

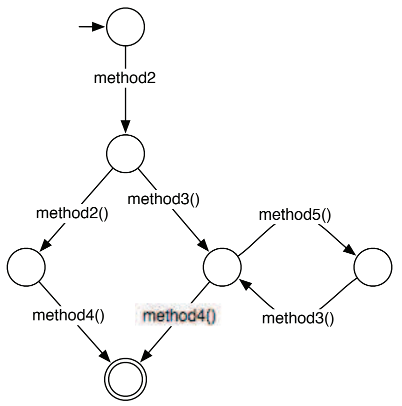
In all these approaches however, only some particular parts of the programming language are considered. When the extracted models are meant to be used for model-checking, the choice of the program parts to be

[1]More details on the runtime aspect is provided in Sect. 3.4.



```
void method1(int x)
{
method2();
if (x<5)
      {
      method2();
      }
else
      {
      while (not(method3()))
            {
            method5();
            }
      }
method4();
}
```

(a) An example of method declaration.



(b) An FSM modeling the system behaviours of method1.

Fig. 3.3: Illustration of FSM extraction.

extracted can be driven by the property to be model-checked (e.g. [11]). In case of software maintenance, one usually does not know which part of the application is faulty before an error occurs. Therefore monitoring relevant information about the occurrence of an error requires to cover a large part of the application. Moreover as the relevance of approach for self-healing relies on the observation made at runtime, it requires that the application models encode these possible runtime observations. In our approach, the extracted models actually encode all the possible occurrences of method invocations, for methods declared in the application, i.e. invocations of methods declared in external components are not considered. This characteristic is related to the fact that the extracted model is used for on-line monitoring and capture relevant information when an error occurs. Therefore an important challenge for model extraction consists of obtaining a complete application model. This requires that the model complies with the specification of the language compiler or virtual machine so that features such as threads and graphical components are treated appropriately.

In order to extract models on large applications, we use a modular approach. A typical output of the model extraction mechanism is depicted in Figure 3.4. It consists of a set of Finite State Machines, each of them possessing one initial state (represented as an hexagon in the figure) and possibly several final states (represented as double-circle states). From each of these initial state, only one event can be triggered, i.e. event m_i for each FSM$_i$ and for $1 \leq i \leq 3$. Moreover these events, called *triggering events*, do not appear in any other transition or in any other FSM. Therefore, when observed at runtime, these events uniquely characterize which FSM is running and initiates any of the behaviours of this FSM.

Considering software applications, triggering events represent methods that are not called from within the application. In our approach, this also takes into account the fact that methods within the application may call an external method that is overridden by a method that is declared in the application itself. This means that triggering events may only be called through external events such as a call from an external component, from user interactions, from occurrences of system events, etc.

Triggering events make it possible to capture concepts such as the behaviours associated to button clicks of a graphical interface (e.g. method *actionPerformed* in Java SWING), the start of a new thread (e.g. method *run* in Java), etc.

Run methods represent concurrency in the application at runtime. This concurrency is also present in the model as triggering events are declared in different FSM that can run concurrently. However more modularity is also introduced in the model whenever this is possible. For instance, a method may be a triggering event although its behaviour does not run concurrently. For instance all actionPerformed methods run on the same thread, the *EventDispatch* thread. In this case, apparent concurrency in the model does not represent actual concurrency at runtime. This approximation is however an interesting means to lower the complexity of the

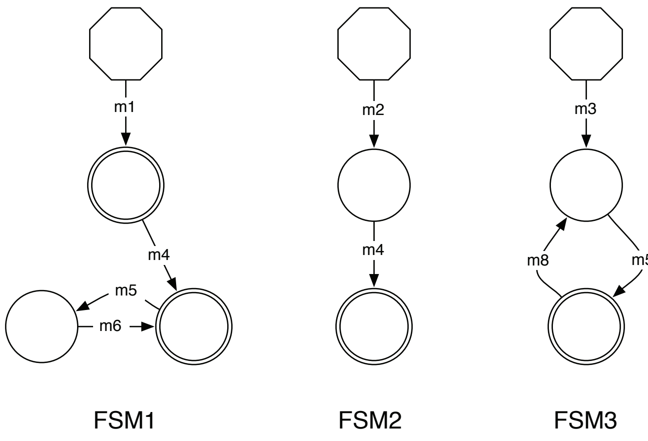


Fig. 3.4: Structure of the extracted FSMs.

extracted models. A modular model extraction approach indeed avoids the computation of a single Finite State Machine, which would become intractable for large applications. This approach allows to extract models of all the possible application behaviours and makes it scalable by splitting the problem into the one of extracting an FSM for each of its entry point.

The runtime dynamic of this model is described in more details in Sect. 3.4.

3.3. Synthesis Phase. The design of such a supervisor corresponds to determining how the application behaviours must be modified in order to avoid undesired behaviours. However designing such a supervisor is a challenging and prone to error task. Moreover the high complexity of software applications makes it difficult to take manually into account all the possible failures that can occur and need to be prevented. For this reason, supervisors may need to adapt at runtime so that they take into account newly observed undesired behaviours, hence performing corrective maintenance. Such an approach is described in Figure 3.5(a).

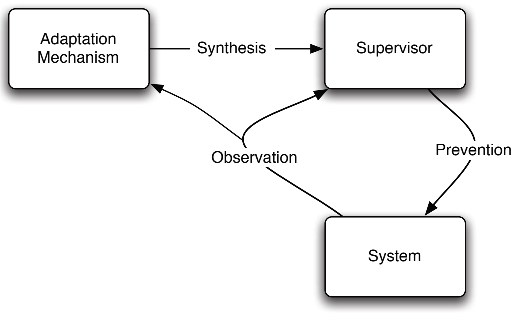
Our approach considers the automatic synthesis of such supervisors. More specifically we consider techniques that automatically compute the model of a supervisor given a model of the application behaviour and a model representing a set of desired behaviours[2]. The Supervisory Control Theory (SCT) on Discrete Event Systems introduced by Ramadge and Wonham [49], offers such a framework and techniques for the automatic synthesis of supervisors.

SCT is a formal theory that aims to automatically design a model for a supervisor ensuring some safety property. The Supervisory Control Theory defines notions and techniques that allow for existence and automatic computation of a model of the supervisor, given a model of the system as well as the property to be ensured. In this theory, models of a system G are represented by languages over alphabets of events, denoted $L(G)$. These languages correspond to sets of sequences of events, each representing a possible behaviour/execution of the system.

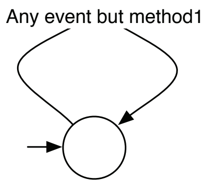
Although not as general as languages, Finite State Machine (FSM) are used to model the possible behaviours of the system as well as the supervisor and the properties to be ensured by control. Regarding the modeling of supervisors, Figure 3.1(b) shows that they can be seen as a function that takes a given sequence s and returns to the system a set of allowed events after s. The function S representing the supervisor can be encoded by a FSM G_S such that for all $s \in L(\mathrm{S})$, $S(s)$ represents the set of events that can be triggered from the state reached in G_S after sequence s.

Supervisors ensure a given property, called *control objective*. Such a property is modeled as a FSM as well, generating a set of "safe" behaviours and meaning that the behaviours that are not encoded by this FSM are undesired. For instance, Figure 3.5(b) represents a very simple control objective which models that method1 must never be executed.

[2]Behaviours that do not belong to this set are undesired.



(a) A Software Control Adaptation View.



(b) A simple control objective.

Fig. 3.5: Software control Adaptation view and a simple control objective.

The main goal of the Supervisory Control theory is to automatically synthesize a model of a supervisor that ensures that the system behaviours are all included in the ones described by the control objective. The theory also considers that not every event can or should be disabled by a supervisor. Such events are said to be uncontrollable. In order to take such events into account, the alphabet of the system is assumed to be composed of a set of *controllable* events ($A_c \subseteq A$) and *uncontrollable* events ($A_u \subseteq A$). Each event of the system is either controllable or uncontrollable. Controlling a system consists of restricting its possible behaviours taking into account the controllable nature of the system events. In order to achieve this, Ramadge and Wonham (see e.g. [63]) introduce a property called *Controllability*. A system G' whose behaviours correspond to a subset of the ones of G is controllable w.r.t A_u and G if $L(G').A_u \cap L(G) \subseteq L(G')$. A controllable set of behaviours G' ensures that no sequence of uncontrollable events can complete a sequence of G' into a sequence of G that is no longer in G'. In other words, the controllability condition ensures the synthesized supervisor can be effectively implemented with respect to the available controllable events. We now define the basic supervisory control problem, which can be stated as the following:

Basic Supervisory Control Problem (BSCP): Given a system G and a control objective K, compute the maximal controllable set of behaviours included in the ones of both G and K.

Ramadge and Wonham (see e.g. [63]) have shown that a solution to the BSCP exists if and only if the maximal controllable set of behaviours included in the ones of both G and K is not empty. They also provided an algorithm computing this FSM which encodes a most permissive supervisor ensuring the control objective (see e.g. [63]). This algorithm can be seen as a function that takes as inputs a set of uncontrollable events A_u, a FSM representing the control objective K and a FSM representing the behaviours of the system G. In our proposed approach, corrective maintenance is applied by modifying the application behaviours. Determining the set of behaviours to be ensured by control is performed solving the BSCP. The obtained model is then use to control the application. Part of the mechanism involved to achieve this is described in Sect. 3.4 and part of it is performed during the pre-deployment phase and is described in Sect. 3.2.

The control objective of Figure 3.5(b) illustrates the case where it is desired to prevent occurrences of method1. Although in some situations such an objective represents the most relevant property to ensure on the system, it may also represent an approximation due to lack of knowledge. The root cause of the failure that leads to the design of this control objective may not indeed come from method1 but from other methods calling method1. If the developers can only observe that the failure occurs when method1 is executed, then preventing the occurrence of method1 appears to be the most straightforward way to avoid the failure.

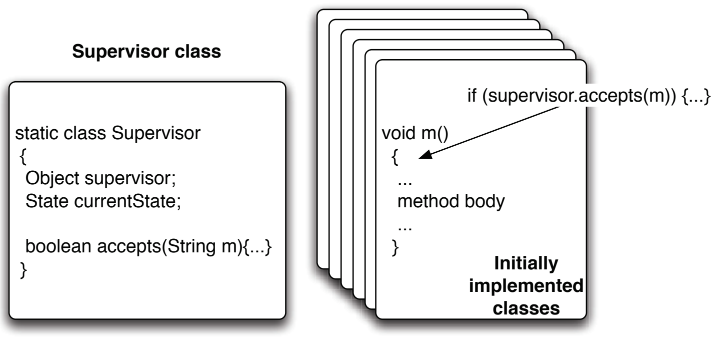
The algorithm solving the BSCP provides a new model of a supervisor which will be used by the application in order to prevent the future occurrence of undesired behaviours. In general, a restart of the application is necessary in order to take into account the newly computed supervisor model.

Finally the extracted models are represented as a composition of FSMs. Classical supervisory control techniques require that a single FSM represents the system behaviours. Such a FSM can be obtained by computing the composition of the FSM representing each component. However, this computation leads to a state explosion problem and represents an important challenge of the supervisory control theory. Some works on control on concurrent systems have been conducted (e.g. [62, 19, 24]) and can be applied to the extracted model. In particular, conditions stated in [24] for efficient modular supervisor synthesis are fulfilled by the model extracted as in Sect. 3.2. For instance one such requirement is that shared event between FSMs are controllable. This requirement always hold with our model at runtime as there is actually no shared event between the modelled concurrent FSMs. This is due to the fact that each FSM is being executed on a different thread at a given time and that the knowledge of the thread on which a method is invoked indicates which FSM this event is belonging to.

3.4. Runtime Supervision. When an error occurs at runtime, the observed behaviour is used in order to modify the extracted model as described in Sect. 3.3. The resulting model encodes a supervisor to be applied to the application at runtime. This section describes this mechanism.

We first consider the control phase which follows the principle illustrated in Figure 3.1(b). In this diagram, the supervisor observes and controls the current behaviours of the system. These behaviours are represented as sequences of events.

As illustrated in Figure 3.6, the model of the supervisor is embedded in the application. More specifically, the model of the supervisor can be considered as an object whose current state can be updated whenever a method of the application to be controlled is invoked. Each time a method is called, then method *accept* is called. First, this method makes the supervisor aware of the method being invoked and updates its knowledge of the current behaviour of the application. Second, this method returns a boolean value indicating whether the supervisor allows the body of the method to be executed. Such an approach allows for dynamic restriction of the system executions, e.g. a method execution may be prevented after a given sequence and allowed after another one.



```
static class Supervisor
{
 Object supervisor;
 State currentState;

 boolean accepts(String m){...}
}
```

```
if (supervisor.accepts(m)) {...}
void m()
 {
  ...
  method body
  ...
 }
```

Fig. 3.6: A possible code instrumentation offering observation and control points.

The models obtained from model extraction and presented in Figure 3.4 represent concurrent Finite State Machines. However the concurrency between these FSMs may not correspond to the one of the threads created during the execution of the application. Considering Figure 3.4 again, although FSM_1 and FSM_2 are modelled as concurrent FSMs, it may be the case that m_1 and m_2 are always executed on the same thread. This may happen for instance when the application to be controlled is an API and m_1 and m_2 are always called from methods of an external component that run on the same thread.

The dynamic of the concurrent FSMs of the model we consider is unlike the standard parallel composition of FSMs (see e.g. [9]). Instead the dynamic of the model considers that only a subset of the concurrent FSMs may run simultaneously. This mechanism is embedded in the implementation of the supervisor and consists of

- mapping at runtime the observed current thread and method call to the appropriate running FSM in order for it to update its current state,
- mapping at runtime the observation of a triggering event, i.e. the first event that can be triggered from a FSM to the corresponding FSM.

Figure 3.7 illustrates the runtime mechanism of concurrent FSM model. We assume here that the model consists of a pool of n concurrent FSMs $\{G_0, \ldots, G_n\}$. In this example, the first method invocation observed is the triggering event associated to G_3 and is executed on thread *Thread1*. Then some of G_3's behaviours may be executed on this thread as well as the triggering event of FSM G_6 on thread *Thread2*. Then G_3 and G_6 run in parallel on their respective threads when the method corresponding to the triggering event of G_2 is invoked on thread *Thread3*. Then the current behaviours of G_3 completes and the method associated to the triggering event of G_7 is invoked on thread *Thread1*. Finally, FSMs G_7, G_6 and G_2 run in parallel on their respective thread until the behaviour of G_6 completes.

This runtime dynamic is sound as triggering events only occur from the initial state of an FSM and do not appear in any other ones. Therefore when a method corresponding to a triggering event is invoked on a thread, there is no ambiguity as to whether it initiates the behaviour of an FSM on this thread or extends the behaviour of the FSM currently associated to this thread: the first case indeed applies. Moreover when a method that does not correspond to a triggering event is invoked on a given thread, it corresponds to the a transition of the FSM currently associated to this thread. The information about the thread on which the method is called removes any ambiguity on the FSM for which the corresponding event belongs to.

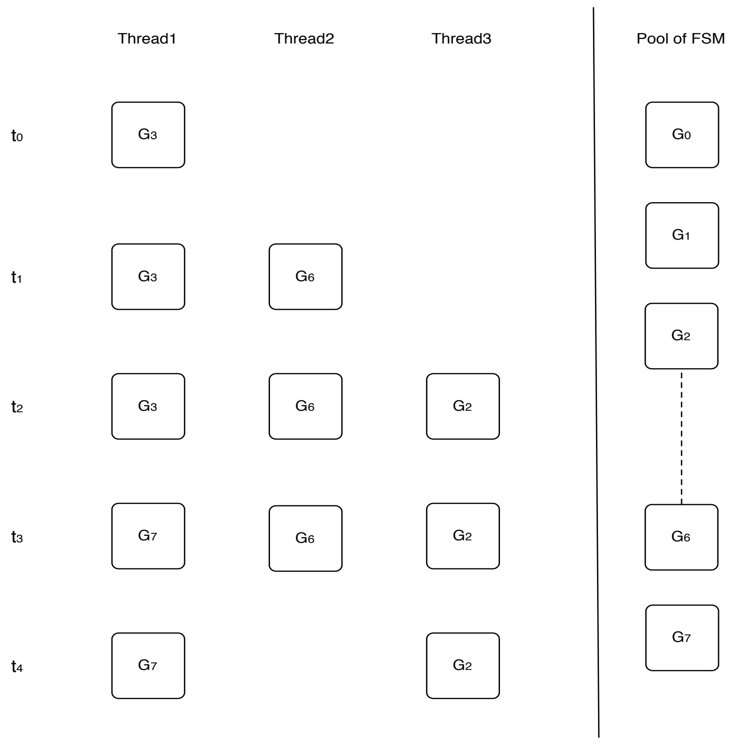


Fig. 3.7: The Runtime Dynamic of the Model Concurrent FSMs

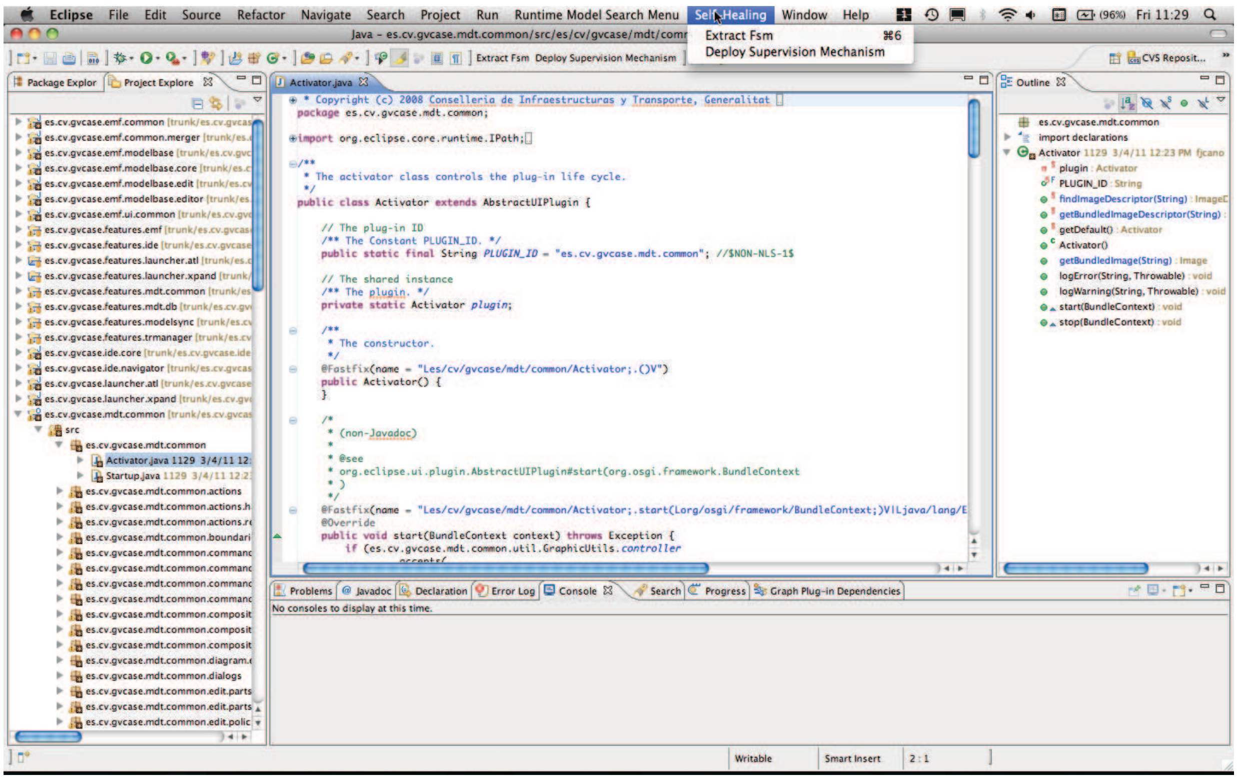
Finally, the supervisor embedded in the FastFix target application is a declared as a *synchronized* object and it is therefore safe to call it form different threads. Such an approach makes it possible for the supervisor to control behaviours that spread over several threads. However, this approach introduces some extra concurrency between threads, i.e. threads have to share an extra resource: the supervisor.

3.5. Summary. The control theoretic approach for self-healing proposed in this section raises several challenges. Some of these challenges correspond for instance to automating the introduction of autonomic features into legacy applications; automatically extracting relevant and accurate models from source code; applying supervisory control theory on large systems; designing accurate control objective, etc. They also relate to different fields of computer science such as software engineering (e.g. software modeling, logging, maintenance), formal methods and control theory. Sect. 4 illustrates this approach on an industrially relevant application: Mokistt while Sect. 5 presents challenges related to our approach.

4. Example. This section applies the approach described in Sect. 3. More specifically, it illustrates the pre-deployment phase on a industrially relevant application: Moskitt [2]. Moskitt is an open source software initially developed for the *Conselleria de Infraestructuras, Territorio y Medio Ambiente*, built on top of Eclipse and which supports modeling tasks. This application is used as a case study within the FastFix project. It consists of numerous modules implemented as OSGI bundles [3]. The applicability of our automated model extraction and supervision deployment mechanisms is illustrated on Moskitt.

Our model extraction and supervision deployment mechanisms have been implemented as an Eclipse plugin, illustrated in Figure 4.1. Table 4.1 presents results regarding the scalability and efficiency of the approach and Figure 4.2 illustrates the outcome of the instrumentation embedding supervisors within the application.

As shown in Figure 4.1, our plugin implements the pre-deployment phase of our self-healing approach, and contains two features: model extraction and supervision deployment. Model extraction is performed through static analysis of the application source code. The different Moskitt bundles appear on the left-hand side of Figure 4.1. For this example, we used a MacBook Pro with a 2.6Ghz dual core i7 processor and 4GB of RAM.


Eclipse File Edit Source Refactor Navigate Search Project Run Runtime Model Search Menu Self-Healing Window Help
Extract Fsm ⌘6
Deploy Supervision Mechanism
Java - es.cv.gvcase.mdt.common/src/es/cv/gvcase/mdt/comr
Extract Fsm Deploy Supervision Mechanism
Package Explor
Project Explore
Activator.java
Outline
Problems
Javadoc
Declaration
Error Log
Console
Search
Progress
Graph Plug-in Dependencies
Writable
Smart Insert
2 : 1

```
* Copyright (c) 2008 Conselleria de Infraestructuras y Transporte, Generalitat
package es.cv.gvcase.mdt.common;

import org.eclipse.core.runtime.IPath;

/**
 * The activator class controls the plug-in life cycle.
 */
public class Activator extends AbstractUIPlugin {

    // The plug-in ID
    /** The Constant PLUGIN_ID. */
    public static final String PLUGIN_ID = "es.cv.gvcase.mdt.common"; //$NON-NLS-1$

    // The shared instance
    /** The plugin. */
    private static Activator plugin;

    /**
     * The constructor.
     */
    @Fastfix(name = "Les/cv/gvcase/mdt/common/Activator;.()V")
    public Activator() {
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext
     * )
     */
    @Fastfix(name = "Les/cv/gvcase/mdt/common/Activator;.start(Lorg/osgi/framework/BundleContext;)V|Ljava/lang/E
    @Override
    public void start(BundleContext context) throws Exception {
        if (es.cv.gvcase.mdt.common.util.GraphicUtils.controller
```

```
No consoles to display at this time.
```

Fig. 4.1: Screenshot of the FastFix Self-Healing Component applied to Moskitt.

Table 4.1 presents results about the model extraction and supervision deployment mechanisms on the Moskitt bundles. First, 54 of the Moskitt bundles were considered, representing more than 20000 method declarations. About 2500 FSMs were extracted from these bundles (one per triggering event) in around 6 minutes and 10 seconds. The size of the extracted FSMs vary from 2 states up to 1381 states. However, 3 FSMs were discarded as their non deterministic version has more than 50000 states[3].

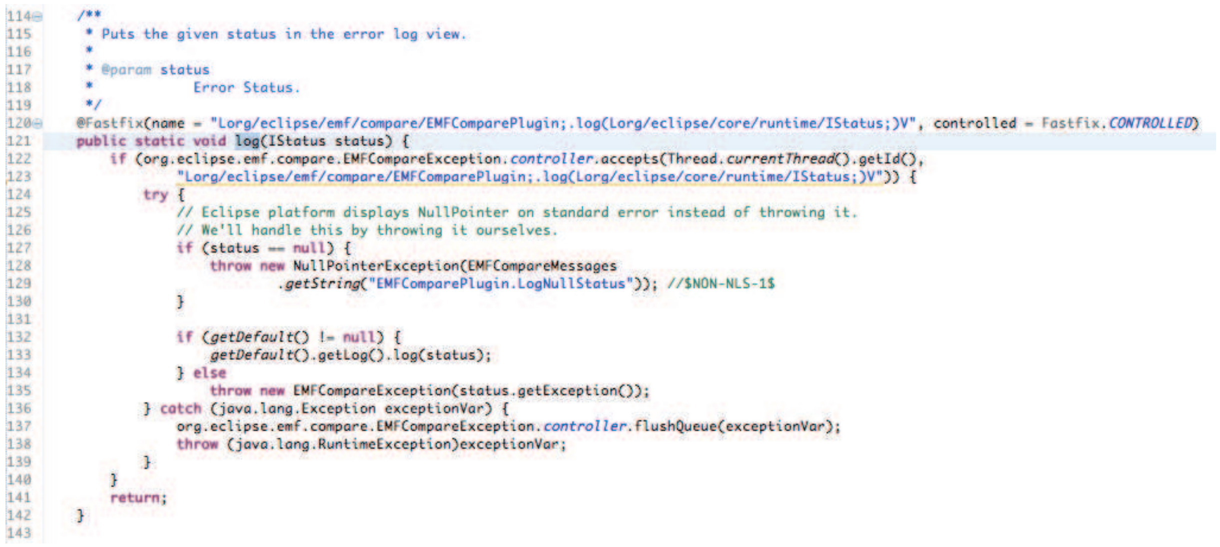
Finally, Figure 4.2 represents the result of the supervision deployment mechanism in the *log* method of the *EMFComparePlugin*. Line 122 and 123 show the call to the *accepts* method from the supervisor. If this method returns *true*, then the method intent to execute the contents of the *try* statement in Line 124. The contents

[3]In this work, a special version of the determinisation algorithm is used which does not ensure an equivalent behaviour to the initial one. However, this algorithm complexity is linear rather than exponential in the classical case.

Nb Bundle	54
Nb Fsm	2492
Nb Methods	21525
Extraction Time	370749 ms
Avg Min FSM Size	2
Avg Max FSM Size	85

Table 4.1: Application of the model extraction and supervision deployment mechanisms to Moskitt. Time is in millisecond and FSM sizes represent numbers of states.

of this *try* clause represent the initial body of the *log* method. If an un-handled exception occurs during the execution of the try clause, then it is caught and the behaviour recorded by the supervisor at runtime and leading to this exception is flushed into a log file for further analysis and patch generation. Moreover, the exception is thrown again in case other deployed supervisors need to be aware of its existence.



```
/**
 * Puts the given status in the error log view.
 *
 * @param status
 *            Error Status.
 */
@Fastfix(name = "Lorg/eclipse/emf/compare/EMFComparePlugin;.log(Lorg/eclipse/core/runtime/IStatus;)V", controlled = Fastfix.CONTROLLED)
public static void log(IStatus status) {
    if (org.eclipse.emf.compare.EMFCompareException.controller.accepts(Thread.currentThread().getId(),
            "Lorg/eclipse/emf/compare/EMFComparePlugin;.log(Lorg/eclipse/core/runtime/IStatus;)V")) {
        try {
            // Eclipse platform displays NullPointer on standard error instead of throwing it.
            // We'll handle this by throwing it ourselves.
            if (status == null) {
                throw new NullPointerException(EMFCompareMessages
                        .getString("EMFComparePlugin.LogNullStatus")); //$NON-NLS-1$
            }

            if (getDefault() != null) {
                getDefault().getLog().log(status);
            } else
                throw new EMFCompareException(status.getException());
        } catch (java.lang.Exception exceptionVar) {
            org.eclipse.emf.compare.EMFCompareException.controller.flushQueue(exceptionVar);
            throw (java.lang.RuntimeException)exceptionVar;
        }
    }
    return;
}
```

Fig. 4.2: A Moskitt method automatically instrumented in order to enable Supervisory Control.

This example shows the feasibility of applying the pre-deployment phase of our proposed approach on an industrially relevant application. Work such as [24] ensures the feasibility of the supervisory control algorithm on concurrent FSMs such as the ones extracted from Moskitt.

5. Challenges. The control theoretic self-healing approach poses several challenges. Some of them are discussed in this section and related to current research efforts. Most of the challenges under consideration are due to system complexity. Complexity relates to the system size, the system model size, the efficiency of the analyses and supervisor synthesis as well as the need for a low overhead during runtime execution.

The approach in Sect. 3 is flexible enough to allow for complexity reduction by considering only subparts of the system to be observed, controlled and modeled and also by approximating the system and control objective models through abstractions. However, reducing the amount of information available to the framework described in Figure 3.2 alters the quality of the supervisors, that can be automatically synthesized and therefore the relevance of the self-healing solution to be applied. Therefore trade-offs between scalability and relevance of the approach have to be determined, posing several challenges. For this purpose challenges related to system observability and controllability, to system modeling, to designing control objective (related to automatic diagnosis), to concurrency and to corrections to be applied (related to automatic repair) are discussed in the rest of this section.

5.1. Finite State Machines and Variables. In Sect. 3 we represent application models as Finite State Machines, where the transitions represent method calls. Although this view of the system behaviours takes into account past executions in order to decide on the control actions to be performed, it does not explicitly take into account system variables. This approach has an interesting upside: the state space of the model is in general smaller than the state space of the application. Without considering system variables, the states of the model do not encode a possible tuple of values of the application variables. Instead states only encode control-flow information (branchings and loops) of the program (as illustrated in Figure 3.3), reducing the model state space.

The downside of this approach is that information on the system behaviours is not as accurate as if variable values were taken into account. For instance, disabling the occurrence of a method call may be dependent on the values of the parameters with which the method is called (if any). Therefore, taking into account some of the application variables into the approach while preserving its scalability is an important but challenging tasks.

Several works have considered supervisory control on FSM with variables: [58, 57, 38, 36, 23]. Although Extended Finite State Machines offer a compact way of representing potentially large, or even infinite system state spaces, the supervisor synthesis takes into consideration the system state space itself. In order to tackle this issue, abstractions of the variable values rather than the possible values themselves should be considered for analysis. This can be done in the same spirit as for Abstract Interpretation ([12]) or data obfuscation techniques (e.g. [4]). Obfuscation techniques aim to abstract the actual variable values into *restricted domains*. Using an FSM makes it easier to calculate the restricted domain of each variable at each point. As transitions that correspond to tests and branches on application variables are performed in the application model, the conjunction of the conditions applied to each variable can be calculated, resulting in the conditions needed to reach the particular point, i.e. the path condition. Naturally, the path condition is a result of the particular values of the program's variables: if the path condition includes the clause $x > 0$ this means that x was tested for being positive somewhere along the execution path and indeed it was positive. Implicitly, the path condition obfuscates the specific variable values for the execution.

5.2. Automatic Recovery. In its basic form, the approach described in Sect. 3 generally requires that the application is restarted in order to take new supervisors into account. This ensures a proper monitoring of the system by the new supervisor. Restarting the application sets the system behaviour model to its initial state. This ensures that the new supervisor can be applied to the system: when it exists a supervisor can always be applied from the system's initial state. One challenge for our approach consists of providing an automated means for avoiding the application relaunch whenever a new supervisor is to be applied. This challenge can be tackled by considering checkpointing techniques such as described in Sect. 2.3.

Checkpointing an entire application is time consuming. In order to lower the rate (and cost) of checkpointing, full checkpoints of the whole application may be complemented with intermediate incremental checkpoints [22] of the memory pages or objects that have changed since the latest full checkpoint. However, the main challenge for checkpointing in a supervised application is to synchronize the application states with model states. Code instrumentation can be used in order to annotate the checkpointing data with the corresponding application model state. In this way both application and model can easily be restarted at the same point. When rollbacks are performed together with a modification of the supervisor (e.g. so that the system does not run towards the previously occurred error), it may not be possible to restart a supervised application at the latest checkpoint. The supervisor model may indeed have been modified so that the model states associated with the latest checkpoint no longer exist. This problem can be sidestepped by rolling the application back to a point where the application execution does not include any state of the supervisor model that has been modified. This can be verified by storing, with each checkpoint, the current supervisor model state as well as all the states that have been visited before. If, when the supervisor model is changed the list of modified states is also stored, then it becomes possible to choose a checkpoint that does not include any modified states.

5.3. Designing Control Objectives. Our proposed approach relies on the synthesis of supervisors from a model of the system behaviours and a control objective. This control objective is represented by a FSM and encodes safety properties over the system behaviours. It is possible for instance to describe what methods must not be executed after some given executions. If the control objective also provides information on the variables of the system, then it allows to describe complex conditions under which some method calls must not be executed.

As mentioned in Sect. 3.1 and illustrated in Figure 3.2, the control objective may be obtained manually and automating its design is a difficult challenge.

Some result in this direction have been obtained in [25] in the specific case of un-handled exceptions. As a general matter, tackling the automatic design of control objective is very much related to automatic fault and anomaly detection (e.g. [10]) as well as automatic diagnosis. Specification mining techniques ([40]) can also be employed in order to extract from the observed undesired trace the pattern that characterize the occurrence of an error.

6. Conclusion. This document deals with software self-healing as investigated in the FastFix FP7 EU project, and focuses on corrective maintenance. A brief state-of-the-art on self-healing is presented and concludes that the research achieved so far is better suited for adaptive and perfective maintenance rather than corrective maintenance.

This work introduces a control theoretic approach which offers a solution to self-healing for corrective maintenance. We describe its different phases: model extraction, supervision deployment and runtime supervision. Results about the feasibility of applying this approach on an industrially relevant system are presented. Finally this paper points out the challenges related to the proposed approach, such as the automatic design of control objective and improving on the application models.

REFERENCES

[1] *Fastfix project consortium: Fastfix project homepage, www.fastfixproject.eu/.*

[2] *The moskitt project: Homepage, http://www.moskitt.org/eng/moskitt0/.*

[3] *Osgi eclipse: Homepage, http://www.eclipse.org/osgi/.*

[4] D. Bakken, R. Rarameswaran, D. Blough, A. Franz, and T. Palmer, *Data obfuscation: anonymity and desensitization of usable data sets*, Security & Privacy, IEEE, 2 (2004), pp. 34–41.

[5] G. Candea and A. Fox, *Crash-only software*, in Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003, pp. 12–20.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, *Automatic workarounds for web applications*, in FSE, 2011.

[7] A. Carzaniga, A. Gorla, and M. Pezzè, *Healing Web applications through automatic workarounds*, International Journal on Software Tools for Technology Transfer (STTT), 10 (2008), pp. 493–502.

[8] ———, *Self-healing by means of automatic workarounds*, in Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, ACM, 2008, pp. 17–24.

[9] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 1999.

[10] V. Chandola, A. Banerjee, and V. Kumar, *Anomaly detection: A survey*, ACM Computing Surveys (CSUR), 41 (2009), pp. 1–58.

[11] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and H. Zheng, *Bandera: Extracting finite-state models from Java source code*, in Software Engineering, 2000. Proceedings of the 2000 International Conference on, IEEE, 2002, pp. 439–448.

[12] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, 1977, ACM Press, New York, NY, pp. 238–252.

[13] W. Cui, M. Peinado, H. Wang, and M. Locasto, *Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing*, in Security and Privacy, 2007. SP '07. IEEE Symposium on, May 2007, pp. 252 –266.

[14] M. Davidsen and J. Krogstie, *Information systems evolution over the last 15 years*, in Advanced Information Systems Engineering, Springer, 2010, pp. 296–301.

[15] R. de Lemos, *ICSE 2003 WADS Panel: Fault Tolerance and Self-Healing*, (2003).

[16] R. Debouk, S. Lafortune, and D. Teneketzis, *Coordinated decentralized protocols for failure diagnosis of discrete event systems*, Discrete Event Dynamic Systems, 10 (2000), pp. 33–86.

[17] B. Demsky and M. Rinard, *Automatic detection and repair of errors in data structures*, in Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, ACM, 2003, pp. 78–95.

[18] ———, *Data structure repair using goal-directed reasoning*, in Proceedings of the 27th international conference on Software engineering, ACM, 2005, pp. 176–185.

[19] M. deQueiroz and J. Cury, *Modular supervisory control of large scale discrete-event systems*, in Discrete Event Systems: Analysis and Control. Proc. WODES'00, Kluwer Academic, 2000, pp. 103–110.

[20] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, *Fulfilling the vision of autonomic computing*, Computer, 43 (2010), pp. 35–41.

[21] B. Elkarablieh and S. Khurshid, *Juzi*, in Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on, IEEE, 2009, pp. 855–858.

[22] E. Elnozahy, D. Johnson, and W. Zwaenepoel, *The performance of consistent checkpointing*, in Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on, IEEE, pp. 39–47.

[23] B. Gaudin and P. Deussen, *Supervisory control on concurrent discrete event systems with variables*, American Control Conference, 2007. ACC'07, (2007), pp. 4274–4279.

[24] B. Gaudin and H. Merchand, *An efficient modular method for the control of concurrent discrete event systems: A language-based approach*, Discrete Event Dyn Syst, 17 (2007), pp. 179–209.

[25] B. Gaudin, E. Vassev, M. Hinchey, and P. Nixon, *A control theory based approach for self-healing of un-handled runtime exceptions*, in 8th International Conference on Autonomic Computing (ICAC 2011), Karlsruhe, Germany, 06/2011 2011.

[26] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya, *Self-healing systems - survey and synthesis*, Decis. Support Syst., 42 (2007), pp. 2164–2185.

[27] N. Gruska, A. Wasylkowski, and A. Zeller, *Learning from 6,000 projects: lightweight cross-project anomaly detection*, in ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis, New York, NY, USA, 2010, ACM, pp. 119–130.

[28] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback control of computing systems*, Wiley-IEEE Press, 2004.

[29] C. Hood and C. Ji, *Proactive network-fault detection [telecommunications]*, Reliability, IEEE Transactions on, 46 (2002), pp. 333–341.

[30] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, *Bugfix: A learning-based tool to assist developers in fixing bugs*, in Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on, IEEE, 2009, pp. 70–79.

[31] M. Jiang, M. Munawar, T. Reidemeister, and P. Ward, *Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring*, in Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, IEEE, 2009, pp. 285–294.

[32] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, Computer, 36 (2003), pp. 41–50.

[33] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, IEEE Computer, 36 (2003), pp. 41–50.

[34] A. D. Keromytis, *Characterizing self-healing software systems*, in In Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS, 2007.

[35] N. Kolettis and N. D. Fulton, *Software rejuvenation: Analysis, module and applications*, in Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), 1995, pp. 381–395.

[36] R. Kumar and V. Garg, *On computation of state avoidance control for infinite state systems in assignment program framework*, Automation Science and Engineering, IEEE Transactions on, 2 (2005), pp. 87–91.

[37] S. Laster and A. Olatunji, *Autonomic Computing: Towards a Self-Healing System*, (2007).

[38] T. Le Gall, B. Jeannet, and H. Marchand, *Supervisory control of infinite symbolic systems using abstract interpretation*, in 44nd IEEE Conference on Decision and Control (CDC'05) and Control and European Control Conference ECC 2005, Seville (Spain), December 2005, pp. 31–35.

[39] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, *Characteristics of application software maintenance*, Commun. ACM, 21 (1978), pp. 466–471.

[40] D. Lo, S. Khoo, and C. Liu, *Mining temporal rules for software maintenance*, Journal of Software Maintenance and Evolution: Research and Practice, 20 (2008), pp. 227–247.

[41] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo, *Flips: Hybrid adaptive intrusion prevention*, in Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), 2005, pp. 82–101.

[42] M. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid, *A case for automated debugging using data structure repair*, in Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, 2009, pp. 620–624.

[43] G. Novark, E. Berger, and B. Zorn, *Exterminator: Automatically correcting memory errors with high probability*, in Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, ACM, 2007, pp. 1–11.

[44] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al., *Automatically patching errors in deployed software*, in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 87–102.

[45] H. Psaier and S. Dustdar, *A survey on self-healing systems: approaches and systems*, Computing, 91 (2011), pp. 43–73. 10.1007/s00607-010-0107-y.

[46] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, *Rx: Treating bugs as allergies—a safe method to survive software failures*, ACM Transactions on Computer Systems (TOCS), 25 (2007), p. 7.

[47] J. Radatz, *IEEE standard glossary of software engineering terminology*, IEEE Std 610121990, 121990 (1990).

[48] P. J. Ramadge and W. Wonham, *Supervision of discrete event processes*, in Proc. of 21st IEEE Conf. Decision and Control, Orlando, FL, Dec. 1982, pp. 1228–1229.

[49] ———, *Supervisory control of discrete event processes*, in Feedback Control of Linear and Nonlinear Systems, vol. 39 of LNCIS, Springer-Verlag , Berlin, Germany, 1982, pp. 202–214.

[50] O. Raz, P. Koopman, and M. Shaw, *Enabling automatic adaptation in systems with under-specified elements*, in WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, 2002, ACM, pp. 55–60.

[51] R. Reiter, *A theory of diagnosis from first principles*, Artificial Intelligence, 32 (1987), pp. 57–95.

[52] G. D. Rodosek, K. Geihs, H. Schmeck, and B. Stiller, *Self-healing systems: Foundations and challenges.*

[53] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, Transactions on Autonomous and Adaptive Systems (TAAS, 4 (2009).

[54] A. Sengupta and A. Dahbura, *On self-diagnosable multiprocessor systems: diagnosis by the comparison approach*, Computers, IEEE Transactions on, 41 (2002), pp. 1386–1396.

[55] O. Shehory, *A self-healing approach to designing and deploying complex, distributed and concurrent software systems*, in ProMAS'06: Proceedings of the 4th international conference on Programming multi-agent systems, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 3–13.

[56] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, *Assure: automatic software self-healing using rescue points*, in ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 2009, ACM, pp. 37–48.

[57] M. Skoldstam, K. Akesson, and M. Fabian, *Modeling of discrete event systems using finite automata with variables*, in Decision and Control, 2007 46th IEEE Conference on, IEEE, 2007, pp. 3387–3392.

[58] ———, *Supervisory control applied to automata extended with variables-revised*, Relatório técnico, Goteborg: Chalmers Uni-

versity of Technology, (2008).

[59] A. Smirnov and T.-c. Chiueh, *Automatic patch generation for buffer overflow attacks*, in IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security, Washington, DC, USA, 2007, IEEE Computer Society, pp. 165–170.

[60] M. Sullivan and R. Chillarege, *Software defects and their impact on system availability-a study of field failures in operating systems*, in Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21), 1991, pp. 2–9.

[61] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, *Automated fixing of programs with contracts*, in ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis, New York, NY, USA, 2010, ACM, pp. 61–72.

[62] Y. Willner and M. Heymann, *Supervisory control of concurrent discrete-event systems*, International Journal of Control, 54 (1991), pp. 1143–1169.

[63] W. M. Wonham, *Notes on control of discrete-event systems*, Tech. Report ECE 1636F/1637S, Department of Electrical and Computer EngineeringUnivertsity of Toronto, July 2003.

Edited by: Dana Petcu and Daniela Zaharie
Received: March 1, 2012
Accepted: April 15, 2012