

# FastFIX: An Approach to Self-Healing

Benoit Gaudin and Mike Hinchey

Lero—The Irish Software Engineering Research Centre, University of Limerick, Ireland  
firstName.lastName@lero.ie

**Abstract**—The EU FP7 FastFIX project tackles issues related to remote software maintenance. In order to achieve this, the project considers approaches relying on context elicitation, event correlation, fault-replication and self-healing. Self-healing helps systems return to a normal state after the occurrence of a fault or vulnerability exploitation has been detected. The problem is intuitively appealing as a way to automate the different maintenance type processes (corrective, adaptive and perfective) and forms an interesting area of research that has inspired many research initiatives. In this paper, we propose a framework for automating corrective maintenance and present its early stage development, 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 equips 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 for effective application of this approach to relevant industrial systems.

## I. INTRODUCTION

SOFTWARE maintenance aims to modify a software system after it is deployed in production ([1], [2]). In [3], the authors identify three different types of maintenance: 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. Over 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 failures, 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 hugely complex software systems require less maintenance and human intervention. With 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 (e.g. [4], [5], [6]). According to these surveys, the major trends towards finding a solution to the self-healing problem rely on redundancy that may concern both architecture and code resources. These

The research leading to these results has received funding from the European Community's Seventh Framework Programme managed by REA—Research Executive Agency <http://ec.europa.eu/research/rea> ([FP7/2007-2013] [FP7/2007-2011]) under grant agreement n° [258109]. This work was also supported, in part, by Science Foundation Ireland grant 03/CE2/1303\_1 to Lero - the Irish Software Engineering Research Centre ([www.lero.ie](http://www.lero.ie)).

approaches somehow 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.

We 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 behaviors. It represents an interesting approach to avoiding behaviors that lead to failures. This is achieved by dynamically disabling some of the implemented features. Moreover, the proposed approach automatically synthesizes supervisors charged with controlling the software. Hence, this automates the computation of a new suitable range of software behaviors whenever corrective maintenance needs to be performed, e.g., a failure has been reported and behaviors exhibiting this failure need to be removed or avoided.

Section II introduces the FastFIX project ([7]) goals and the different research aspects that are investigated: context elicitation, event correlation, fault replication and self-healing. Section III presents the early stage development and approach considered for self-healing, which is based on control theory.

Finally, challenges to be tackled in order to implement effective and efficient control theoretic self-healing features are discussed in Section IV. Most of these challenges relate to supervisory control theory and its applicability to software systems.

## II. FASTFIX: MONITORING CONTROL FOR REMOTE SOFTWARE MAINTENANCE

The FastFIX project aims to provide methods and a platform for improved remote maintenance of software applications. The FastFIX platform monitors the execution of applications, their environment and user behaviors. It also provides techniques that analyze the collected data in order to identify symptoms of execution errors, performance degradation, or changes in user behavior.

This platform comprises both a client part which interacts locally with the target application and a server part which receives data from the client in order to perform analyses.

Collecting information from the target application, as well as its environment and users, is the basis for the FastFIX analyses. Therefore, context elicitation and user modeling play a crucial role in the project. These challenges are tackled through lightweight software sensor deployment into the runtime environment, together with facilities to interpret the user behaviors from data representing their interaction with the application.

These sensors can also provide information about the execution of the application itself and its environment, i.e., method calls, variable values, timestamps, etc. This data can then be analyzed by an event correlation component in order to detect anomalies representing possible attacks or application malfunctions. Rule-based systems are often used in order to perform event correlation. However these systems face issues whenever the complexity of the system to be monitored and the amount of possible correlations is large. More specifically, managing a large amount of rules in order to ensure consistency and proper priorities as well as to avoid redundancy, is a very challenging task. The FastFIX project will tackle this issue and investigate rule-based correlation engines that are easier to define and maintain.

The data collected by the FastFIX platform is also used in order to replicate faults whenever they occur. Indeed, fault replication represents an interesting feature in order to diagnose issues. It is first an appealing approach as it avoids manual fault replication from the symptoms reported by the user, which can be incomplete, inaccurate or even irrelevant to the error. Moreover, fault replication techniques address the replication of faults related to concurrency. This is of high interest as these types of faults are usually difficult to reproduce, and hence to diagnose and fix.

The information about the target application collected at runtime is used in order to perform self-healing. When a failure occurs, the self-healing capabilities make it possible to automatically modify the application behaviors so that this failure cannot occur in future executions. Performing such automation is a challenging task and better suits types of faults for which no new behavior creation is required. For this reason, the FastFIX self-healing mechanism is flexible and also allow for humans-in-the-loop in order to tackle those software fixes that cannot be automated.

Finally, as the collected data is sent to the FastFIX server for analyses, the system execution and user information must be sent from the client machine to the maintenance team. As this information may contain sensitive personal data, it is important to ensure user confidentiality. This is tackled in FastFIX using obfuscation techniques (e.g. [8]). Obfuscation techniques aim to abstract the actual variable values into *restricted domains* so that it is not possible anymore to precisely determine what values were used by the user. However, the domain in which the value is abstracted is accurate enough in order to replicate the application execution in a similar way as the one performed with actual user data.

Figure 1 illustrates how the different aspects tackled by FastFIX can be combined and act in a complementary manner in order to achieve effective and efficient remote software maintenance. On the client side of the diagram are the user, the FastFIX target application and the FastFIX client itself. This client monitors the user interactions with the target application as well as the application execution itself. From this information, it is able to perform analyses and identify application failures or changes in the user behaviors. It can then provide feedback and recommendations to the user themselves, or send

the collected information to the FastFIX server for further processing. This data can be preprocessed in order to be obfuscated so that no sensitive user information is sent to the server.

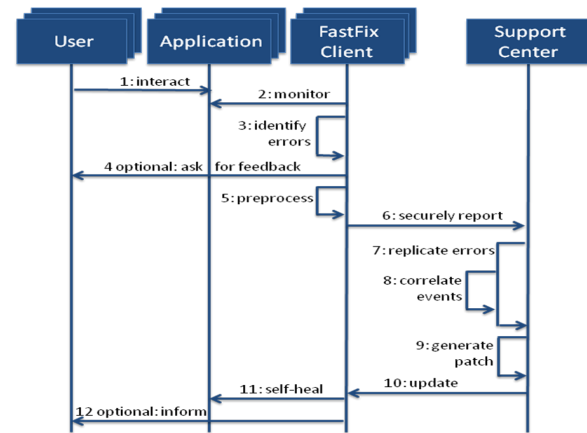


Fig. 1. The FastFIX overview.

The information sent to the FastFIX server can then be used in order to perform event correlation, fault replication and automatic patch generation, through self-healing. The outcomes of these analyses are then reported back to the FastFIX client so that actions are taken on the target application, e.g. applying patches or providing user recommendations.

The rest of this paper focuses on the FastFIX approach to self-healing, which relies on Supervisory Control Theory for discrete event systems.

### III. A CONTROL THEORETIC APPROACH TO SOFTWARE SELF-HEALING

In computing systems, control theory has traditionally been applied to data networks, operating systems, middleware, multimedia and power management ([9]). This section proposes a control-based approach for the self-healing of software systems.

With this approach, systems can be automatically equipped with autonomic features and therefore follow the autonomic feedback loop of Figure 2(a) at runtime. In particular, sensors and actuators are automatically added to the software system in order to realize the *Data Collection* and *Action* phases of Figure 2(a). The *Analysis* phase is related to control theory. The system sensors and actuators also implement the feedback control loop presented in Figure 2(b) and two types of analyses can therefore be achieved: runtime control decision and automatic supervisor synthesis. Runtime control decision can ensure the avoidance of known undesired behaviors during execution.

Supervisor synthesis is used to automatically modify the system behaviors. It represents the core technique for corrective maintenance in our approach. The overall proposed approach is detailed in the remainder of this section.

Our self-healing approach consists of two different phases: a pre-deployment phase which is performed before the system

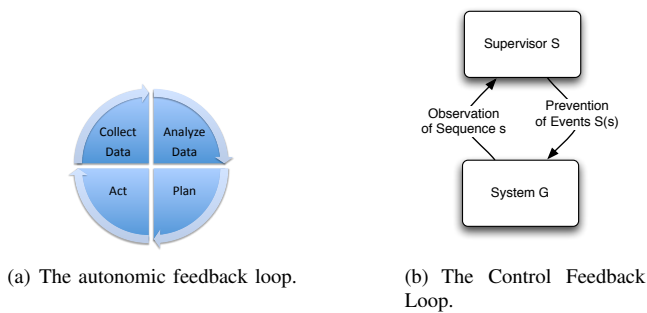


Fig. 2. The runtime autonomic and control feedback loops.

is deployed and where self-healing features are added to the software; and a post-deployment phase 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 the system control at runtime as well as supervisor synthesis whenever new runtime system specifications need to be ensured, e.g., when a fault has occurred. Overall the presented approach can be seen as a three phase approach: pre-deployment, control and synthesis. The pre-deployment phase prepares the system for control and synthesis. However, the concepts related to the pre-deployment phase depend on the ones related to the control and synthesis phases. We will therefore discuss the pre-deployment phase last.

#### A. Control Phase

We first consider the control phase which follows the principle illustrated in Figure 2(b). In this diagram, the supervisor observes and controls the current behaviors of the system. These behaviors are represented as sequences of events.

As a basic case, we consider that the events that can be observed by the supervisor consist of method calls. This can be further augmented for instance with other program statements such as conditions and also values passed to method parameters. Therefore we consider that the behavior of a software application is described by the sequence of method calls that occur at runtime<sup>1</sup>. Each time a method is executed, the supervisor is aware of it and can update its knowledge regarding the current behavior of the system.

Implementing Figure 2(b) requires the addition of observation (sensors) and control (actuators) points. In order to achieve this, we consider embedding some code that models the supervisor into the software 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 called. Moreover, control can be performed by preventing method executions as modeled in the supervisor. Figure 3 illustrates this idea where a *Supervisor* type is added to the software application. This type (or class) also provides a method *accepts* which given the name of a

method returns true if and only if the supervisor will allow that this method is called from the current state. Whenever a method is allowed, its body is executed and the current state of the supervisor object is updated.

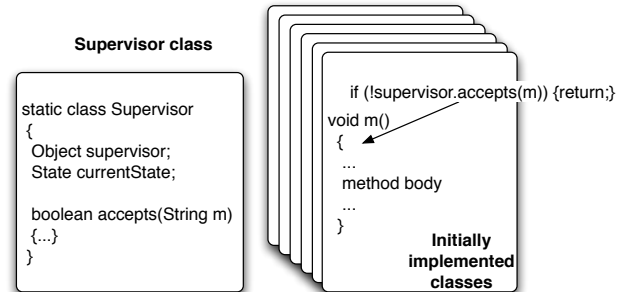


Fig. 3. A possible code instrumentation offering observation and control points.

If the method is not allowed by the supervisor, then it must not be executed. A basic approach to achieving this is described in [10], [11] and consists of returning abruptly as indicated in Figure 3. 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.

#### B. Synthesis Phase

Paragraph III-A describes the principle and possible mechanisms for controlling software application behaviors at runtime by means of a supervisor. The design of such a supervisor corresponds to determining how the application behaviors must be modified in order to avoid undesired behaviors. However designing such a supervisor is a challenging task and prone to error. Moreover, the high complexity of software applications makes it difficult to manually take 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 behaviors, hence performing corrective maintenance. Such an approach is further described in Figure 4(a) and considers automatic synthesis of such supervisors. More specifically, we consider techniques that automatically compute the model of a supervisor given a model of the application behavior and a model representing a set of desired behaviors<sup>2</sup>. Supervisory Control Theory (SCT) on Discrete Event Systems introduced by Ramadge and Wonham ([12]) 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. Supervisory Control Theory defines notions and techniques that allow for the existence and automatic computation of a model of the supervisor, given a model of the system as well as the property

<sup>1</sup>For simplicity, we discuss the approach with this basic setting. The general case is discussed in Section IV.

<sup>2</sup>Behaviors that do not belong to this set are undesired.

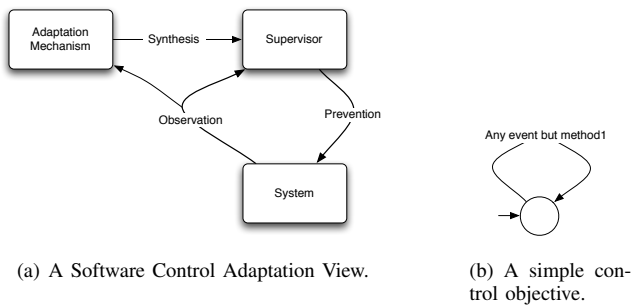


Fig. 4. Software control Adaptation view and a simple control objective.

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 behavior/execution of the system.

Although not as expressive as languages, Finite State Machines (FSM) are used to model the possible behaviors of the system as well as the supervisor and the properties to be ensured by control. Regarding the modeling of supervisors, Figure 2(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  $s \in L(S)$ ,  $S(s)$  represents the set of events that can be triggered from the state reached in  $G_S$  after sequence  $s$ . It is worthwhile noting that in the case where events only represent method calls, program variables are not taken into account in the application behavior model. The state space of the corresponding FSM does not therefore correspond to the one of the program variables. Instead the model encodes loops and branching points in the program, limiting the state space explosion issue related to large systems (more details are provided in Section III-C and more particularly in Figure 5).

Supervisors ensure a given property, called the *control objective*. Such a property is modeled as a FSM as well, generating a set of “safe” behaviors and meaning that the behaviors that are not encoded by this FSM are undesired. For instance, Figure 4(b) represents a very simple control objective which models that `method1` must never be executed.

The main goal of Supervisory Control Theory is to automatically synthesize a model of a supervisor that ensures that the system behaviors 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 behaviors taking into account the controllable nature of the system events. In order to achieve this, Ramadge and Wonham (see for example [13]) introduce a property called *Controllability*. A system  $G'$  whose behaviors

correspond to a subset of those of  $G$  is controllable w.r.t  $A_u$  and  $G$  if  $L(G') \cdot A_u \cap L(G) \subseteq L(G')$ . A controllable set of behaviors  $G'$  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 in the following.

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

Ramadge and Wonham (see for example [13]) have shown that a solution to the BSCP exists if and only if the maximal controllable set of behaviors included in those of both  $G$  and  $K$  is not empty. They also provide an algorithm computing this FSM which encodes a most permissive supervisor ensuring the control objective (see for example [13]). 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 behaviors of the system  $G$ . In our proposed approach, corrective maintenance is applied by modifying the application behaviors. Determining the set of behaviors to be ensured by control is performed through solving the BSCP. The obtained model is then used to control the application. Part of the mechanism involved in achieving this is described in Section III-A and part of it is performed during the pre-deployment phase and is described in Section III-C.

### C. Pre-Deployment Phase

The pre-deployment phase aims to prepare the software application before deployment so that control and synthesis can be performed at runtime. This preparation is not application specific and the same processing is applied to any software systems under consideration. It consists of two 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. These features will allow for software control such as depicted in Figure 2(b). Intuitively, automatically instrumenting source code for this purpose consists of automatically augmenting the application source code with statements such as in Figure 3, i.e., embedding a supervisor into the system as well as adding conditional statements in each method body so that method calls can be observed and method body execution can be controlled at runtime.

Moreover, model extraction from source code is performed in order to obtain a model of the behaviors of the system. As mentioned in Paragraph III-A, application behaviors are represented with sequences of method calls. An over-approximation can be obtained from the source code by considering methods, branching and loops as illustrated in Figure 5.

### D. Overall Approach

The proposed overall approach is depicted in the diagram of Figure 6. The left hand side of this diagram represents the

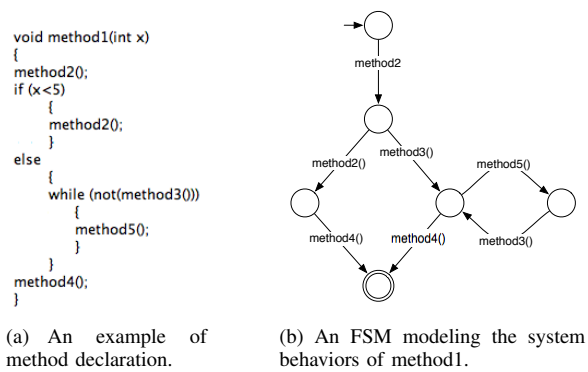


Fig. 5. Illustration of FSM extraction.

pre-deployment phase at which code is instrumented in order to introduce observation and control points as well as data structures that make it possible to represent and manipulate 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 behaviors is also automatically extracted from the source code by analyzing control flows and method calls in the application.

During the runtime and maintenance phase, the software artifacts (source code and binary code or Bytecode) are modified no more. Only models of a supervisor representing their possible runtime behaviors are manipulated in order to control the application so that only desired behaviors can be executed. If no control is necessary at first, then the extracted model of the application can be used as a supervisor model. This will certainly have no effect on the implemented application behaviors.

Some unknown possible failures of the system may occur at runtime, requiring the application to be healed. The observation of such a failure indeed indicates that the system behavior is not satisfactory and needs to be corrected.

In this approach, this correction is performed by modifying the supervisor that interacts with the application at runtime. Using Supervisory Control Theory as introduced in Paragraph III-B, this can be automatically achieved when a control objective is provided (in this approach, a model of the possible behaviors of the software application was computed in the pre-deployment phase and is therefore assumed to be available). In some situations, this control objective can be automatically derived from observations of failures during the system execution (see for example [10]). In general, control objectives can also be provided by expertise. The accuracy and relevance of the expertise involved in designing a control objective will have an 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), then a simple control objective excluding the  $u$  previously-observed undesired sequences of

method calls can be submitted to the supervisor synthesis algorithm. Of course this latter option may correspond to a coarse control of the application, unnecessarily removing proper (acceptable) behaviors.

The control objective of Figure 4(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 in 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.

In any case, 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 behaviors. In general, a restart of the application is necessary in order to take into account the newly computed supervisor model.

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 objectives, 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. Some of these challenges are detailed in Section IV.

#### IV. CHALLENGES

The control theoretic self-healing approach introduced in previous sections poses several challenges. Most of these challenges are directly or indirectly related to performance and complexity. These issues are related 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 Section III is flexible enough to allow for complexity reduction by considering only sub-parts 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 6 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. For this purpose, challenges related to system observability and controllability, to system modeling, to designing control objectives, to concurrency and to correction types to be applied, are discussed in the rest of this section.



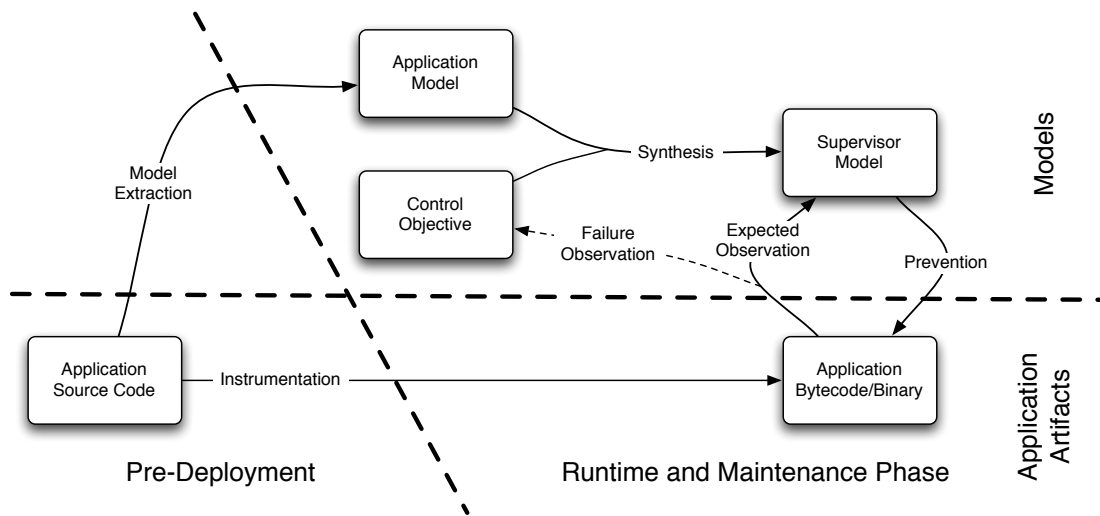


Fig. 6. A Software Control Approach to Self-Healing.

#### A. Controllability and Observability.

Section III introduces a control theoretic approach for the automation of corrective maintenance processes. This approach models the possible executions of the system as sequences of method calls. The approach relies on a supervisor that observes some of the method calls at runtime (observability). Moreover the control mechanism disables the occurrence of some method calls in order to ensure some properties of the application (controllability).

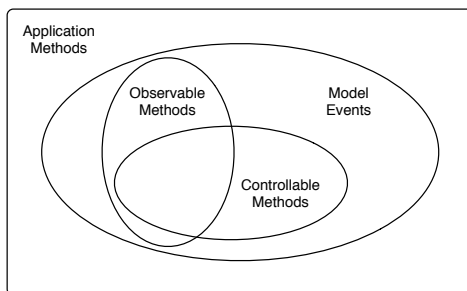


Fig. 7. Observable and Controllable Software Methods.

Figure 7 represents the relationship between observable and controllable methods as well as how they relate to the sets of application methods and the methods that are part of the application model (representing model events). First the model events correspond to a subset of the application methods. Second, observable and controllable methods represent a subset of the model events. The observable methods are those whose call can be observed by the supervisor at runtime. The controllable methods represent those whose execution can be prevented by the supervisor.

The amount of model events and observable events has an impact on the size of the application model. Moreover,

the complexity of the supervisor synthesis as well as the runtime overhead increase with this model size. On the other hand, the relevance of the supervisor synthesis increases with the amount of information present in the application model. Therefore, as our approach aims to systematically automate the introduction of method observability, one important challenge is to find approaches to determine the best trade-off between the amount of methods to be made observable and the amount of information required for relevant synthesis.

Moreover, defining controllable methods is also a challenging task. Although it is technically feasible to prevent any software method to be executed at runtime, this is however unreasonable for some methods as it may have an impact on the application's integrity. For instance, preventing the execution of a method that returns an object, which will be later processed during the program execution, may introduce new possible failures in the system. Therefore, procedures (i.e., methods that do not return any object) are better suited to be controllable. This however does not fully guarantee the application's integrity as procedures may modify global variables. Techniques such as slicing (e.g., [14]) could be considered in order to determine methods whose execution has a very limited impact on the global execution of the application. Another type of method that possesses such a property are methods capturing events from graphical interfaces and executing code in reaction to button clicks, for instance. These methods represent good candidates for controllable methods as they are not called from other parts of the application or third party applications.

#### B. Finite State Machines and Variables.

In Sections III and IV-A, we represent application models as Finite State Machines, where the transitions represent method calls. Although this view of the system behaviors makes it possible to take into account the past execution in order to

decide on the control actions to be taken, it does not explicitly take into account the 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. Indeed, with this approach, the states of the model do not encode a possible tuple of values of the application variables. Instead states only encode branchings and loops of the program (as illustrated in Figure 5).

The downside of this approach is that information on the system behaviors is not as accurate as if variable values were taken into account. For instance, disabling the occurrence of a method call by control may depend on the values of the parameters with which the method is called (if any). Therefore, taking into account some of the application variables in the approach while preserving its scalability is a challenging task.

Several works have considered supervisory control on FSM with variables (e.g., [15], [16], [17], [18]). 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 ([19]) or data obfuscation techniques (e.g., [8]).

### C. Automatic Extraction of Application Models.

The approach introduced in Section III relies on the automatic design of a model of the application behaviors. In its basic form, this model can be a Finite State Machine whose transitions represent method calls. As explained in Section IV-B, such a model can be extended in order to take system variables into account. Extended FSM can then be considered as a way to model the application behaviors.

Some tools have been implemented in order to extract and analyze models represented as EFSM. For instance, PROMELA allows for program modeling with FSMs. PROMELA models can be used as input to the SPIN tool, which can then model-check this model against some properties. Bandera ([20]) is a tool that allows for FSM extraction from Java code. Bandera offers the possibility of exporting the extracted models into the PROMELA format. More recently in [21], 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. The approach described in Section III requires that the model obtained of the application is complete; i.e., any observable program execution should be encoded in the model. This characteristic is related to the fact that the extracted model is used for on-line monitoring and must take into account all the possible system behaviors. 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. This aspect is also related to the notion of concurrent behaviors and is detailed further in Section IV-E.

### D. Improving Application Models from Runtime Observations.

As mentioned in Section IV-C, the model of the application behaviors should be complete in order to be used to monitor the application at runtime. However although some variables of the system are taken into account in the model, some others may still be abstracted, leading to a model that is an over-approximation of the possible application behaviors.

However, some information regarding both the path of the model taken as well as the variable values observed during the system execution represent valuable information regarding actual possible behaviors. For instance, if a transition of the FSM modeling the system results from over-approximating the actual system behaviors, it will not be triggered. Therefore if after a large number of executions it is observed that some transitions have never been triggered, one may conclude that these transitions are over-approximating the system behaviors and should not be taken into account for analysis. This leads to an improvement of the application model from the observations made at runtime.

This is an important point as although model completeness is important in order to ensure adequate monitoring of the system at runtime, it may unfortunately also induce some over-restrictive control. In other words, software control may be over-restrictive when taking into account parts of the model that actually do not correspond to actual application behaviors.

Improving the model relevance is therefore an important challenge in order to ensure the most accurate control on applications. Considering the approach detailed in Section IV-B, such improvements can be obtained from variable observations and determination of relationships between parameter domains of the application methods. Such relationships can be learned through probabilistic approaches (e.g., [22]) or system identification techniques ([23]).

### E. Multi-threading and Concurrent Control.

Most software applications possess several components running on different threads. They can therefore be modeled as a composition of FSMs, each modeling a component. In this section we consider the control of concurrent systems. Classical supervisory control techniques require that a single FSM represent the system behaviors. Such an 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 supervisory control theory. Some work on control of concurrent systems have been conducted (e.g., [24], [25], [26]) and even in the case of models with variables in [18]. However, more work is required in this area in order to improve the state-of-the-art. Moreover, one specificity of the approach described in Section III is that the number of components running concurrently varies over time. Finally, the

composition mode between components is different from the usual synchronous and parallel composition that is considered in classical supervisory control.

#### F. Designing Control Objectives.

Our proposed approach relies on the synthesis of supervisors from a model of the system behaviors and a control objective. This control objective is represented by a FSM and encodes safety properties over the system behaviors. 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 for the description of complex conditions under which some method calls must not be executed.

As mentioned in Section III-D and illustrated in Figure 6, the control objective may be obtained manually, and automating its design is a difficult challenge.

Some results in this direction have been obtained in [10] in the specific case of un-handled exceptions. As a general matter, tackling the automatic design of control objectives is very much related to automatic fault and anomaly detection (e.g., [27]) as well as automatic diagnosis. Therefore techniques related to automatic diagnosis can contribute to automating control objective designs and should be further investigated in the context of automatic supervisory control.

Control policies can also consider performing some actions whenever control is applied to the system, creating new behaviors. This can be subject to different strategies from which one needs to be selected.

#### V. CONCLUSION

This paper describes the EU FP7 FastFIX project, which tackles issues related to remote software maintenance. In order to achieve this, the project considers approaches relying on context elicitation, event correlation, fault-replication and self-healing. After introducing the general objectives addressed within FastFIX, we describe its self-healing approach and early development, which aim to automate the generation of patches, hence reducing time and cost related to some of the corrective maintenance tasks.

This self-healing approach relies on control theory. We describe its different components and phases and introduce some of its challenges. This paper points out the challenges that are related to supervisory control theory. It also describes some challenges, such as automating the design of control objectives as well as introducing new behaviors into the application.

#### REFERENCES

- [1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. ACM*, vol. 21, pp. 466–471, June 1978. [Online]. Available: <http://doi.acm.org/10.1145/359511.359522>
- [2] M. Davidsen and J. Krogstie, "Information systems evolution over the last 15 years," in *Advanced Information Systems Engineering*. Springer, 2010, pp. 296–301.
- [3] J. Radatz, "IEEE standard glossary of software engineering terminology," *IEEE Std 610121990*, vol. 121990, 1990.
- [4] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," *Decis. Support Syst.*, vol. 42, no. 4, pp. 2164–2185, 2007.
- [5] O. Shehory, J. Martinez, A. Andrzejak, C. Cappiello, W. Funika, D. Kondo, L. Mariani, B. Satzger, M. Schmid, A. Andrzejak *et al.*, "Self-Healing and Recovery Methods and their Classification," *Self*, 2009.
- [6] H. Psailer and S. Dustdar, "A survey on self-healing systems: approaches and systems," *Computing*, vol. 91, pp. 43–73, 2011, 10.1007/s00607-010-0107-y. [Online]. Available: <http://dx.doi.org/10.1007/s00607-010-0107-y>
- [7] "Fastfix project consortium: Fastfix project homepage, [www.fastfixproject.eu/](http://www.fastfixproject.eu/)."
- [8] D. Bakken, R. Rameswaran, D. Blough, A. Franz, and T. Palmer, "Data obfuscation: anonymity and desensitization of usable data sets," *Security & Privacy, IEEE*, vol. 2, no. 6, pp. 34–41, 2004.
- [9] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, *Feedback control of computing systems*. Wiley-IEEE Press, 2004.
- [10] 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, June 2011.
- [11] B. Gaudin and A. Bagnato, "Software maintenance through supervisory control," in *34th annual IEEE Software Engineering Workshop*, June 2011.
- [12] P. J. Ramadge and W. Wonham, "Supervisory control of discrete event processes," in *Feedback Control of Linear and Nonlinear Systems*, ser. LNCIS, vol. 39. Springer-Verlag, Berlin, Germany, 1982, pp. 202–214.
- [13] W. M. Wonham, "Notes on control of discrete-event systems," Department of Electrical and Computer Engineering University of Toronto, Tech. Rep. ECE 1636F/1637S, July 2003.
- [14] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.
- [15] M. Skoldstam, K. Akesson, and M. Fabian, "Supervisory control applied to automata extended with variables-revised," *Relatório técnico, Goteborg: Chalmers University of Technology*, 2008.
- [16] T. Le Gall, B. Jeannot, 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.
- [17] 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*, vol. 2, no. 1, pp. 87–91, 2005.
- [18] B. Gaudin and P. Deussen, "Supervisory control on concurrent discrete event systems with variables," *American Control Conference, 2007. ACC'07*, pp. 4274–4279, 2007.
- [19] 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: ACM Press, New York, NY, 1977, pp. 238–252.
- [20] 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.
- [21] N. Gruska, A. Wasykowski, 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: ACM, 2010, pp. 119–130.
- [22] R. Michalski, J. Carbonell, and T. Mitchell, *Machine learning: An artificial intelligence approach*. Morgan Kaufmann Pub, 1983.
- [23] L. Ljung and E. Ljung, *System identification: theory for the user*. Prentice-Hall Upper Saddle River, NJ, 1987, vol. 280.
- [24] Y. Willner and M. Heymann, "Supervisory control of concurrent discrete-event systems," *International Journal of Control*, vol. 54, no. 5, pp. 1143–1169, 1991.
- [25] 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.
- [26] B. Gaudin and H. Marchand, "An efficient modular method for the control of concurrent discrete event systems: A language-based approach," *Discrete Event Dyn Syst*, vol. 17, no. 2, pp. 179–209, Apr 2007.
- [27] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.