Software Maintenance through Supervisory Control

Benoit Gaudin
Lero - The Irish Software Engineering Research Center
University of Limerick
Email: benoit.gaudin@lero.ie

Alessandra Bagnato
Corporate Research Division
TXT e-solutions
Email: alessandra.bagnato@txt.it

Abstract—This work considers the case of system maintenance where systems are already deployed and for which some faults or security issues were not detected during the testing phase. We propose an approach based on control theory that allows for automatic generation of maintenance fixes. This approach disables faulty or vulnerable system functionalities and requires to instrument the system before deployment so that it can later be monitored and interact with a supervisor at runtime. This supervisor ensures some property designed after deployment in order to avoid future executions of faulty or vulnerable system functionalities. This property corresponds to a set of safe behaviors described as a Finite State Machine. The computation of supervisors can be performed automatically, relying on a sound Supervisory Control Theory. We first introduce some basic notions of Supervisory Control theory, then we present and illustrate our approach which also relies on automatic models extraction and instrumentation.

Index Terms—Software Control, Software Maintenance, Supervisory Control Theory, Vulnerabilities.

I. INTRODUCTION

This work deals with fixing or patching faults and security issues from software systems after deployment. We assume that the systems under consideration went through the different life cycle phases (design, implementation, testing, deployment) and is now in maintenance phase. In this case, faults and security issues that have not been detected during the testing phase remain in the system and are reported whenever their corresponding symptoms are observed by the users. Moreover, we consider how deployed systems require to act as quickly as possible in order to generate and deploy a fix/patch for an observed fault or vulnerability issue. In case of software systems, the approach proposed in this paper offers a means to automate the computation of a patch that prevent occurrences of faulty behavior while maintaining the other system functionalities. This approach represents a real benefit as it provides a quick and reliable way of patching deployed systems, which can then run safely until the release of a corrected version of the source code.

This work is part of the EU FP7 FastFIX project [1]. FastFIX considers systems of industrial relevance and its results will include a platform and a set of open source tools to on-line monitoring of execution environments, gathering semantic information on application and user behavior. Using event correlation techniques, FastFIX aims to identify failure symptoms, performance degradation or changes in user behavior and allows for failure replication, patch generation and patch deployment, resulting in a self-healing software application.

Self-healing software is a concept derived from the Autonomic Computing community. Autonomic Computing is an IBM initiative described in [2]. The authors envision systems that follow biological principles and are able to manage themselves. One of the interesting properties that an autonomic system should possess is self-healing. Self-healing represents the ability for a system to automatically recover from attacks or malfunctions and is further discussed in e.g. [3], [4].

Control theory shares some principles with autonomic computing and is considered in this work. Previous works on software control have been investigated in e.g. [5], [6], [7], [8], [9], [10], [11]. The authors of [5] introduce an approach for controlling software systems. The control acts on some variables of the system. In [6] the authors consider the control of communication between network nodes in order to ensure a given level of performance. [8] also applies control techniques in order to provide self-management properties to the system for performance optimization purposes. In [7], the authors consider control techniques in order to resolve conflicting policies to be ensured on the system at runtime. An important aspect of controlling systems is to ensure their stability, i.e. demonstrating that the control achieves what the designer have intended. As explained in [9], this is of great importance whenever system self-management is considered, where the system is left running without human intervention. This point is also explained in [10] where the lack of understanding of how automated actions affect system behavior is seen as one of the main reasons why automatic approaches are not more used. In this paper, we present an approach for software patching by applying control theory techniques described in [11]. This theory applies to formal models and provides a formal description of the remaining behaviors of the system under control.

In this work we consider a formal specification (possible sequences of method calls) of both the system and the requirements. Instead of proving that the system fulfills some requirements, we control it in order to enforce them. This is of particular interest whenever the system is already deployed and modifying the application source code is not straightforward (e.g. remote patching). Figure 1 positions our approach with the development life cycle taken into consideration.

First, as our approach aims to possibly control the system to be deployed, some models of the system need to be created (automatically from the source code in our case) and the system needs to be instrumented: addition of observation and control points. These models and instrumentation are not used until the system has been tested, verified and then deployed.
As systems have become more and more complex, ensuring that their behaviors fulfill given requirements is an important challenge. Model checking techniques can be applied to check whether system specifications ensure a given property. When it is the case, then the specifications are satisfactory. When it is not the case, the specifications have to be re-designed.

A. Autonomic Systems

Self-adaptive systems possess the ability to adapt to changes or situations. In particular, they can modify their configurations or behavior in order to optimize or repair themselves.

Autonomic systems can be seen as another term referring to self-adaptive systems. Historically, autonomic systems referred to biologically inspired systems, e.g., the human body immune system. Software systems complexity is increasing as well as the need to understand the varying environments and frequently changing user needs. This leads to increased overhead of maintaining and supporting them. These problems motivate the need of autonomic software paradigm with so called self-* properties such as self-healing, self-configuring, self-managing, self-optimizing and so on.

To achieve the so called self-* capability the system needs to continuously monitor its execution environment, input parameters and produce output. Moreover it needs to detect requirements violations and to aid the system to switch to (predefined) variants of its behaviors that allows restoring requirements satisfaction. These different steps are usually presented as the autonomic feedback loop. Figure 2 recalls the one presented in [14] and shows that an autonomic system must be able to observe itself and its environment, analyse the collected data and have some knowledge about its proper behavior in order to make decision on whether the current observations are satisfying or whether some actions should be performed in order to ensure proper behaviors of the system.

The analysis and decision part can be implemented in separate module from the system itself, that must then be able to observe and act onto the system. The system entry points that allow for observation and action are respectively called sensors and actuators. In our approach these points are implemented through program instrumentation. The analysis phase of Figure 2 corresponds to the automatic design of a supervisor, applying control theory techniques, and the decision phase corresponds to the application of that supervisor onto the system at runtime.

B. Supervisory Control of Discrete Event Systems

As systems have become more and more complex, ensuring that their behaviors fulfill given requirements is an important challenge. Model checking techniques can be applied to check whether system specifications ensure a given property. When it is the case, then the specifications are satisfactory. When it is not the case, the specifications have to be re-designed.

Supervisory Control of Discrete Event Systems ([13]) aims to automatically design a model for a controller that is able to prevent some of the behaviors of the system from occurring. The obtained controller interacts with the system to be supervised as illustrated in Figure 3.

Fig. 1. Our approach with the development life cycle taken into consideration.

Fig. 2. The autonomic feedback loop as described in [14].

Fig. 3. The obtained controller interacts with the system to be supervised as illustrated in Figure 3.
Supervisory Control theory defines notions and techniques that allow for existence and automatic computation of a model of the controller, given a model of the system as well as a property to be ensured.

Applying Supervisory Control techniques requires that a model of the system is available. For Discrete Event Systems (DES), languages over alphabets are often considered. For a more practical aspects, Finite State Machine (FSM) are considered. An FSM is a 4-tuple \((\Sigma, Q, q_0, \delta)\), where \(\Sigma\) is a finite alphabet (set of events), \(Q\) a finite set of states, \(q_0 \in Q\) is the initial state of the FSM and \(\delta : Q \times \Sigma \to Q\) is the partial transition function. Intuitively, for a sequence of events \(s \in \Sigma^*\), \(s\) is a possible behavior of the system if \(\delta(q_0, s)\) is defined. In this case, \(\delta(q_0, s)\) represents the state that the system reaches after the sequence of event \(s\) occurred. If this new state is marked, it usually means that sequence \(s\) corresponds to the completeness of a task. For a state \(q\), \(\delta(q)\) represents the set of events that can be triggered from state \(q\). Finally, the set of behaviors of system \(G\) is the language generated by its FSM and is denoted \(L(G)\).

As shown in Figure 3, a supervisor can be seen as a function that takes a given sequence \(s\) of the system and returns to the system a set of allowed events after \(s\). In some cases, it may be desired that the behaviors of the system is restricted in order to ensure a given property. Such a property may be modeled as a FSM as well, generating a set of “safe” behaviors.

The main goal of the Supervisory Control theory is to compute a new model of the system whose behaviors are all included in the ones described by the control objective. In order to achieve this, it will not be possible anymore to trigger some events that could initially be triggered from the system. However, it is usually unrealistic to assume that any type of events can be disabled in such a way. If for instance, an event corresponds to some sensor reading or the tic of a clock, then it cannot be prevented from occurring.

In order to take such events into account, the alphabet of the system is assumed to be composed of a set of controllable events \(\Sigma_c \subseteq \Sigma\) and uncontrollable events \(\Sigma_u \subseteq \Sigma\). Each event of the system is either controllable or uncontrollable. Intuitively, uncontrollable events represents events that cannot or should not be prevented from occurring.

Controlling a system consists of restricting its possible behaviors. However this restriction must take into account the controllable nature of the system events. In order to achieve this, Ramadge and Wonham (see e.g. [11]) introduce a Controllability property. A system \(G'\) whose behaviors correspond to a subset of the ones of \(G\) is controllable w.r.t \(\Sigma_u\) and \(G\) if

\[
L(G') \cap L(G) \subseteq L(G')
\]  

Where \(\cdot\) represent the concatenation operator between events. Ensuring the controllability of a subset of the behaviors of a system ensures that the new model does not rely on disabling uncontrollable events and is then practically feasible. The basic supervisory control problem can then 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 behaviors included in the ones of both \(G\) and \(K\).

In this work, we consider solving the BSCP in case where the system corresponds to a program and the control objective corresponds to a set of behaviors that do not expose functionalities with known vulnerabilities. Finally, it is worth noting that it exists more problems that SC theory can tackle. Some of these problems will be discussed in Section V.

**Example 1:** This example considers a system which can perform two actions: act1 and act2. The set of possible behaviors of the system is described in Figure 4(a). From its initial state, the system can perform either act1 or act2. If act1 is performed, then the system enters a state where event end1 can be triggered. This corresponds to a proper termination of act1, leading the system back into its initial state where a choice between executing act1 or act2 can be made again. However, a failure can also occur while performing act1, leading the system into a deadlock state. A similar behavior can be executed if act2 is performed.

In some cases, it may be required to modify the system behaviors in order to fulfill some requirements. When such modifications happen at the design phase, it can then be seen as a model refinement before the system is implemented. If such modifications need to be performed after the system is implemented and deployed, then it can be seen as controlling
the system. Similar to modeling the system behaviors, requirements can be expressed as Finite State Machines. Figure 4(b) provides an example of such possible requirement (control objective). This FSM states that once the system has started, any event can be observed from it, however, if act1 is being observed then end1 is the only event that can be observed after this. Then any event can be observed again. Such a control objective actually aims to ensure that a failure does not happen the first time act1 is being performed.

Given a FSM of the system and the control objective, Supervisory Control theory provides techniques for computing a model of a supervisor that enforces the control objective, taking into account that some events are not controllable. Example 1, it is assumed that the event 'failure' is not controllable and Figure 5 represents a supervisor that ensures the control objective presented in Figure 4(b). This supervisor is actually the most permissive supervisor ensuring this control objective, i.e. no smaller controllable behavior (as defined in Equation 1) can be prevented by control while still ensuring it.

This supervisor actually states that in order to prevent a failure to occur the first time act1 is being performed, there is no other alternative but to prevent act1 to be performed at all. Therefore, in order to fulfill the control objective, the system must be downgraded to only be able to perform act2.

Fig. 5. A supervisor for system \( G \) and control objective \( K \).

### III. APPROACH

In this section, we describe our approach to apply to software maintenance some of the Supervisory Control principles introduced in Section II-B. The approach proposed in this work is described in Figure 6. It consists of 2 phases: one off-line (at design phase) and one at runtime. At design phase, the application is instrumented and a formal model of the behavior of the system is computed. The application model and instrumented application are then used at runtime in order for a supervisor (i.e. patch) to be synthesized from the application model and for the instrumented application to interact with the synthesized supervisor. In our approach, models are represented with Finite State Machines (FSM) and the application model FSM is automatically extracted from the source code\(^2\).

The instrumentation of the application and the model extraction are performed during the application development phase. These actions prepare the application for control after deployment. Once the application is deployed and running, some unexpected faults or security issues may occur. In this case, maintenance needs to be performed in order to fix/patch the application. In our approach we consider expressing properties (control objectives) that represent system behaviors that avoid the observed occurrence of faults or security exploitation. These control objectives can also be represented by Finite State Machines.

This approach restricts behaviors as a means to system patching, i.e. occurrences of faulty behaviors are prevented. This makes it possible to automate the computation of behaviors to be prevented while maintaining the other functionalities of the patched system. In case of software system, this represents a real benefit as this approach provides a quick and reliable way of patching deployed systems before releasing a version with corrected source code.

Fig. 6. Approach for automatic control of software

The Supervisory Control Theory presented in Section II-B can be applied to the Finite State Machines of both the system and the control objective, producing a FSM representing a supervisor that can ensure the control objectives on the system.

Finally, the obtained supervisor can be encoded into a patch file that is used by the instrumented application. This supervisor can then control the application in order to avoid the executions that lead to the previously observed faults or security issues.

Finally whenever some method execution is disabled by a supervisor, some alternative and systematic actions may be taken. Figure 7 illustrates such a possible action where the user is informed of the prevention of an execution due to control.

Fig. 7. Message indicating that control has been performed and functionality disabled.

Section III-A introduces techniques that can be used to extract Finite State Machines modeling the behavior of the system from source code. Section III-B describes how the...
system can be instrumented in order to interact with the model of the supervisor and Section IV illustrates the approach with examples.

A. Model of the System and Requirements

In this paper, we consider Finite State Machines (FSM) to model the behavior of the system as well as the properties to be ensured by control. FSM possess a formal semantics and can then be used for verification purposes in order to prove or check properties about the system. Such an approach is applied for model checking techniques and model checkers such as SPIN (see e.g. [15]) rely on this formalism. In this work, we consider control rather than verification, meaning that a model of the system is 1) automatically modified in order to ensure a given property and 2) that the obtained new model represents a supervisor (also called a controller) that is used to monitor and control the system at runtime as illustrated in Figure 3.

Considering nowadays systems’ complexity, manually building Finite State Machines that represent the system behavior is tedious and error-prone. Therefore, approaches for automatically extracting FSM from the system source code have been considered. For instance, Bandera (see e.g. [16]) allows for model extraction from Java programs and can output these models in different languages such as PROMELA ([17], [18]).

The Finite State Machines considered by SPIN and Bandera are actually more complex than the ones described in Section II as they consider system variables. The different states of the FSM correspond to different values of the variables of the system. Although these FSM model the behaviors of the system, they are state-oriented and their number of states depends on the possible value range of the system variables.

More recently, the authors of [19] considered an approach to extract FSM from several programming languages such as Java and C. Their approach is behavior-oriented and considers method calls rather than system variables. The resulting extraction process is lightweight and the size of the extracted FSM remains reasonable for analysis. In [19], the analysis performed consists of looking for patterns related to method calls.

Our approach considers a similar view on what part of the behavior of the system are represented by the FSM (method calls rather than variables). However our approach requires a model that is complete in terms of possible observation that can be made at runtime. This is due to the fact that in our approach, the model of the system (or the supervisor) is not used for verification but rather at runtime for monitoring and control. Verification only requires models that capture information related to the property to be checked, while control requires that every possible observation of the system made at runtime is encoded in the supervisor.

In order to illustrate our approach, we describe how Finite State Machines can represent the behavior of the system in our context. Figure 8 presents some code sample and the corresponding FSM extracted from this piece of code. The method calls are extracted and correspond to the edges of the generated FSM. Source code branching (e.g. IF, SWITCH statements, etc) and loops (e.g. FOR and WHILE statements, etc) are also modeled and represent branching and loops in the corresponding FSM.

It is important to notice that the obtained FSMs actually model over-approximation of the possible behaviors of the system. This is partly due to the fact that data is not taken into account in the extraction process. Considering the example of Figure 8 again, it may be the case that method3 always returns True. This would mean that states 4 and 5 of the FSM in Figure 8(b) would never be reached. Detecting such things would require extra analyses which are out of the scope of this paper.

B. System Instrumentation

As illustrated in Figure 3, our approach relies on the use of a supervisor that can observe the behavior of the system and after each observed sequence provides a set of allowed events. In this section, we illustrate how this mechanism can be implemented through the use of code instrumentation. First of all, as described in Section II-B, the supervisor can be seen as a function that given a sequence of events \( s \) returns the set of allowed events after this sequence. Implementing such a method is straightforward whenever a FSM of the supervisor is available. If \( q \) represents the state of the FSM reached after sequence \( s \), then the set of allowed events after \( s \) corresponds to the reachable events from \( q \), i.e. \( \delta(q) \).

Figure 9 illustrates how code can be instrumented in order to implement the control loop presented in Figure 3. The underlying principle of the approach is that the function encoded by the supervisor is used when a method is called in order to decide if its body should be executed. Not every method may be instrumented and not all the instrumented methods may be controlled. For instance, preventing the execution of a method body for a method which must return some value is an issue as this value may be needed. One approach to this issue is
to only consider controlling methods that do not return any value. This point is detailed further in Section V.

In this work, we consider Java applications and introduce a new Supervisor class to the system. This class is declared as static and possess two attributes: an FSM representing the model of the supervisor and a state representing the current state of this model. This FSM is instantiated from a “patch” file containing the model of the supervisor. The current state is instaniated as the initial state of this FSM. The Supervisor class also contains a static method called `Supervisor.accepts` which takes a string representing a method name (e.g. `m`) and returns a boolean. The value of this boolean is true if and only if the model of the supervisor encodes that `m` can be triggered from the current state of the model. Whenever the execution of `m` is authorized, `Supervisor.accepts` also updates the current state of the model as well as the sequence of observed method calls.

Therefore, implementing the monitoring of method calls and updating at runtime the model of the current state of the model of supervisor accordingly can simply be achieved by systematically instrumenting each method `m` with:

\[ \text{Supervisor.accepts}(m); \]  

With this approach whenever an authorized method is called at runtime, it is indicated to the `Supervisor` class which can then update the current state of the model.

The inserted code can also be augmented so that the body of unauthorized method calls are not executed. This should only be done for controllable methods while uncontrollable methods will be instrumented as described in Statement (2). In this work, we assume that only methods that do not return any value can be controllable. This allows for instance to prevent the execution of methods associated to graphical elements, e.g. the `actionPerformed` methods associated to Swing buttons do not return any value. Controllable methods are instrumented as follows:

\[ \text{if } \neg \text{Supervisor.accepts}(m) \text{ then return; } \]  

Statement (3) indicates that whenever a method `m` is called, it is first checked if calling this method from the current state of the supervisor model is authorized. If it is the case, then the body of the method is executed normally and the current state is updated. If the method call is not authorized then the method exits before its body is executed.

Method `supervisor.accepts` and the instrumentation presented in Statements (2) and (3) make it possible for a supervisor modeled by an FSM to control a Java program as illustrated in Figure 3.

Although source code instrumentation is considered in our approach, other techniques can be envisaged in order for the supervisor to interact with the system. For instance, for Java programs, it is possible to instrument the Bytecode, leaving the source code unchanged. Javassist ([20]) is a library that makes it possible to instrument the Bytecode, offering entry points to perform actions when a method is called.

Javassist offers many facilities to instrument bytecode among which the `insertBefore` method. Given a method `m` of a Java program, the `insertBefore` method allows to insert Java code that will be executed before the body of `m`.

Javassist also makes it possible to add new .class files to some existing program and to refer to the corresponding classes in the code inserted through the `insertBefore`.

### IV. Example

In this subsection, we illustrate the applicability of supervisory control principles to a concrete example. We consider a basic calculator with a graphical interface, presented in Figure 10.

In this subsection we consider two examples. First Example 2 shows how supervisory control techniques can be applied to control the application so that specific sequences of events that lead to some vulnerabilities cannot be executed. More specifically, Example 2 considers the case where the system does not handle some `division by zero` exception.

We also consider Example 3 where it is assumed that it is possible to edit the text field of the calculator. In this example, supervisory control is used to prevent possible BOF vulnerabilities.

**Example 2**: Figure 11 represents a Finite State Machine modeling the behaviors of the calculator. Each event of the Finite State Machine represents the processing associated to each of the button of the interface presented of Figure 10. For instance, event `0` represents the call of the method activated when button `0` is pressed. i.e label `0` represents the call of the method that is triggered when the graphical event representing the click of button `0` is caught. All the other events...
Fig. 11. A model of the calculator behaviors.

except for 'arithmeticException' (i.e. 1, ..., 9, +, -, *, / and 'clear') represent the method call activated when the corresponding button is pressed. Event 'arithmeticException' represents the raise of an exception corresponding to division by zero in this example. Labels of the form All \ {0,..,9} in Figure 11 represents the set of all events except for the ones belonging to X. The FSM of Figure 11 encodes the possible visible behaviors of the calculator and makes it possible to determine the behaviors possibly following after a given sequence of events. For instance, if a user presses buttons '1', then '/', and then '0', then we can infer that the FSM of Figure 11 started from state 0 and went through states 1, then 2 and finally 3. From this state the event arithmeticException cannot be triggered but it might be triggered if the user then presses any of the buttons '+', '-', '/', '*' and '='.

The Finite State Machine presented in Figure 12 encodes the set of all possible behaviors that do not contain sequences of '/' followed by sequences of '0' and then followed by '='. Intuitively, this control objective aims to prevent the execution of sequences in which '/' can be directly followed by '0' and then '='. It is assumed here that the control objective is manually designed after the observation of an unexpected exception being raised. In some cases however, the control objective can be automated generated ([21]). This type of control objective can be derived from observation of a runtime exception whenever such a sequence occurs. Note that in this example, it is assumed that dividing by zero only leads to an exception whenever button '=' is pressed. This can result from an implementation choice of the calculator where processing of the operands is only performed when '=' is pressed.

For illustration purposes, we consider here that the actions related to the buttons of the interface of Figure 10 correspond to the controllable events except for button '='; i.e. \( \Sigma_u = \{=\} \). Figure 13 represents the supervisor obtained using classical supervisory control algorithm as presented in [11]. According to this supervisor, occurrences of 'arithmeticException' cannot occur after a division by zero. As division by zero is the only condition for raising this exception in the system, combining the supervisor with the system prevent any occurrence of 'arithmeticException'.

Example 3: We now consider that the system under consideration follows a client-server architecture and that Figure 10 represents the client side of the application and that actual computations are performed on the server side. Server side computations are only called from the client side when pressing one of the operation button (i.e. +, -, *, /) or the '=' button. It is also assumed that the text field of the calculator can be edited and that values can therefore be entered through a keyboard. Figure 14 represents a Finite State Machine modeling the behaviors of the calculator in this case. The FSM is quite similar to the one of Figure 11, except that a new event is introduced, i.e. 'focusGained', modeling that the text field has received focus and is ready for edition.

For the sake of argumentation, we assume here that there is a possible BOF vulnerability due to the possibility of editing
the calculator text field. There may be indeed no control on neither the client nor the server side in order to avoid buffer overflow.

Figure 15 represents a possible control objective which aims to avoid buffer overflow issue for the calculator. This control objective states that only events related to digit buttons (i.e. 0,..,9) and the clear button can be executed after the focus has been gained by the calculator text field. Pressing one of these buttons overwrites the content of the text field. Moreover, this control objective indicates that no more than 3 digits can be entered in the calculator\(^3\). Therefore enforcing this control objective on the system ensures that no numbers of more than 3 digits can be sent on the server side to be processed, thus avoiding buffer overflows.

Using the classical SupCont supervisory control algorithm again, a maximal supervisor is provided in Figure 16. This FSM corresponds to the one representing the calculator behaviors in Figure 11, except that:

- events \(=, +, -, *, /\) cannot occur anymore after the calculator text field gained focus, unless some digit buttons or the clear button have been pressed first.
- only numbers with no more than 3 digits can be entered and processed by the calculator.

\(^3\)This number is only kept to 3 to ensure the readability of the figures in this example. Higher numbers could be used depending on what bound is suitable to avoid a buffer overflow.

V. DISCUSSION AND CONCLUSION

This paper presents an approach for controlling software execution after the system has been deployed. The approach consists of instrumenting the program before it is deployed so that it is ready to interact with a supervisor at a later stage, e.g. when some fault or vulnerability has been discovered. Section IV shows example where supervisory can be applied to prevent sequences that may exhibit runtime exception or vulnerability issues. In these examples, some FSM modeling the system and control objectives are provided. These FSMs represent over-approximations of the behaviors of the system to be controlled. It is to be noticed that the less accurate the approximation is, the more brutal the supervisor can be. This can however be compensated by the control objective.

Another point of discussion is related to the notion of controllable and uncontrollable events. As pointed out in Section III-B, one may consider that some methods should be controllable because of their structure, e.g. they return some value and preventing the execution of their body is not appropriate. The architecture of the system is another criteria that should be taken into account for decision as to what method should be controllable. In case of a client-server architecture, one may decide to control the client side, the server side, or both. This choice would impact on the event controllability. Considering Example IV again, it is there assumed that only buttons are controllable, allowing for a supervision of the client only. This approach is interesting as graphical interface components are good candidates for controlling the user interaction with the system. Such an approach also allows for customized control
where the control objective could vary depending on the user, i.e., different supervisors may apply to different instances of the client application.

On the other hand, it may be safer to control the server application instead of the client one. This would ensure that whatever action is being performed on the client side, the control objective is ensured on the server side. This may be of interest when there is a risk that the supervisor can be disabled or modify on the client application.

Control can also be applied globally on both the client and server applications, providing the strength of both previously mentioned approaches. Another approach would be to focus on the communication protocol between the client and server application.

Finally, only method calls are considered in our model. Adding information about the values of the program variables to our models would improve the quality of the control being applied to the system. However, this may considerably increase the complexity of analyzing such models. Ongoing work from our European Research Project FastFIX ([1]) will study this aspect and evaluate the feasibility of our methodology on industrial application.

ACKNOWLEDGEMENTS

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]) through the FastFIX project, under grant agreement no [258109]. This work was also supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES