Supervisory Control for Software Runtime Exception Avoidance

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

Paddy Nixon
University of Tasmania
Hobart, Australia
Paddy.Nixon@utas.edu.au

ABSTRACT

The Supervisory Control Theory (SCT) introduced by Ramadge and Wonham offers a framework for the control of Discrete Event Systems. In this paper, we formalize some concepts about corrective software maintenance within this framework. More specifically, we consider SCT as a way to control software systems behaviors and avoid occurrences of runtime exceptions. This approach is attractive as algorithms for controllers synthesis offer a means to automate part of the corrective maintenance process. In this paper, we introduce problems related to removing observed software failures by control, as well as solutions.

Keywords

Formal Method, Self-Adaptation, Control, Monitoring

1. INTRODUCTION

This work deals with software self-adaptation in order to automatically modify the system behaviors when facing runtime faults. More specifically, we consider legacy systems and automatically provide them with self-adaptation capabilities that allow for handling of runtime exceptions. We assume that the systems under consideration went through the different software life cycle phases (design, implementation, testing, deployment). Typically, possible faults such as IO and NullPointer exceptions are not all detected during the testing phase, and remain in the system. These faults are usually reported by the user whenever their corresponding symptoms are observed at runtime. This work extends previous work reported in [Gaudin et al. 2011] by formalizing conditions under which adaptation of the system behavior can be performed autamatically, leading to a correct and suitable solution.

Self-adaptation is a property that is encountered in Autonomic Computing [Kephart and Chess 2003]. As explained in [Dobson et al. 2010] control theory principles are suitable to implement the autonomic feedback loop. Therefore in this work we consider adaptation through the use of control theory, and more specifically the Supervisory Control Theory on Discrete Event Systems developped described in (see e.g. [Wonham 2003]). We aim to automate part of the corrective maintenance process by formalizing some corrective software maintenance problems and providing solutions based on the SCT framework. SCT on DES offers a framework and results for automatically synthesizing a model of a controller from a model of the system behaviors and a property to be ensured (control objective).

In this work, we consider the case where a model encoding all the possible system behaviors is not available. System adaptation is performed according to the discovery of behaviors that were not modeled and are undesired. To our knowledge, no work on SCT so far has considered problems where no model representing all possible system behaviors is available. In this paper, we consider that:

1. A partial model of the program is available and is represented with a Finite State Machine (FSM).
2. This model does not encode all the behaviors of the system that are observable at runtime.
3. Failures correspond to the occurrence of runtime exceptions.

These assumptions apply to the general context of software maintenance. In our approach, Point 1 is tackled following [Corbett et al. 2002, Gaudin et al. 2011], where a partial model of the program is extracted from its source code. Point 2 represents the fact that, as runtime exceptions may not be captured in the program source code, they are not encoded in the model of the system. For modern programming languages such as Java or C♯, the program is executed within a runtime environment. Runtime errors such as exceptions can occur and be observed[1]. Point 3 states that these exceptions are not desired. This implies that the system execution is faulty whenever such an exception is raised. Therefore some correction must be performed on the system so that this behavior can no more occur. In classical corrective maintenance, such modification is performed on the source code which is then recompiled and redeployed onto the user environment. This work aims to apply supervisory control concepts in order to automate this modification. The contribution of this paper is twofold:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
C3S2E-12 2012, June 27-29, Montreal [QC, CANADA]
Editors: B. C. Desai, S. Mudur, E. Vassev
Copyright 2012 ACM 978-1-4503-1084-0/12/06 $15.00.

9[1]E.g. [Gaudin et al. 2011] describes how the Javassist library makes it possible to catch and observe runtime exceptions ([Javassist]).

- It maps some aspects of corrective software maintenance with concepts from the supervisory control theory.
- It provides results on the existence and synthesis of a maximal supervisor, automating part of the software maintenance process. The computation of such a supervisor depends on *extrapolation functions* and under some conditions, optimal extrapolation functions can be determined.

This paper follows the following structure. Section 2 introduces some background on software maintenance and Supervisory Control Theory as well as a mapping between between these fields. Section 3 presents specific control problems and proposes solutions for them.

2. BACKGROUND AND MAPPING

Background on Discrete Event Systems (DES) and Supervisory Control Theory (SCT) are presented in Section 2.1. Section 2.2 briefly presents generalities about classical software engineering processes, with an emphasis on corrective maintenance.

2.1 DES and Supervisory Control

In this work, a system model corresponds to a model of its possible behaviors. Such a model is described by a language L over an alphabet A. The set of languages over alphabet A is denoted $\mathcal{L}(A)$.

Given $s, s' \in A^*$, ss' (or s.s') denotes the concatenation of s and s'. Symbol ϵ denotes the empty string and is such that $s.\epsilon = \epsilon.s = s$. Moreover, we say that $s' \leq s$ whenever s' is a prefix of s (i.e. it exists $s'' \in A^*$ s.t. $s = s's''$). We denote $\overline{L}$ the prefix-closure of a language $L \subseteq A^*$ (i.e. $\overline{L} = \{s \in A^* \mid \exists s' \in L, s \leq s'\}$).

The Supervisory Control Theory (SCT) on Discrete Event Systems (DES) introduced by Ramadge and Wonham is described in [Wonham 2003] and [Cassandras and Lafortune 1999]. In this theory, DES behaviors are modeled with languages over finite alphabets A where each element of A represents an event that can occur in the system. According to Ramadge and Wonham's theory, events of a system are either uncontrollable (A_u), i.e. the occurrence of these events cannot be prevented by a supervisor, or controllable (A_c). Therefore $A = A_u \cup A_c$ and $A_u \cap A_c = \emptyset$.

Formally a supervisor S for a system G is given by a function $S : L(G) \rightarrow \{\gamma \subseteq A \mid A_u \subseteq \gamma\}$, delivering the set of actions that are allowed in G under control of supervisor S after a trajectory $s \in L(G)$. We denote S/G the closed-loop system consisting of the initial system G controlled by the supervisor S. A model of S can be easily derived from a model of S/G and vice-versa.

Because some events are uncontrollable, not any language can represent the set of behaviors of a controlled system. For this reason, the notion of *controllable language* was introduce: if L and K are two languages over the same alphabet A such that $K \subseteq L$ and $A = A_u \cup A_c$ then K is controllable with respect to A_u and L if and only if $KA_u \cap L \subseteq K$. Finally, one can note that the empty language is always controllable.

Basic Supervisory Control Problem - BSCP ([Wonham 2003]) *Given a system G and a control objective $K \subseteq L(G)$, the Basic Supervisory Control Problem consists of computing the model of a supervisor S such that $L(S/G) \subseteq K$ is controllable and maximal, i.e. if any other supervisor S' is such that $L(S'/G) \subseteq K$ and $L(S'/G)$ is controllable then $L(S'/G) \subseteq L(S/G)$.*

As a model of S can be easily derived from S/G, S/G is usually considered as a solution of the BSCP, rather than S itself. Moreover whenever a solution to the BSCP exists, it corresponds to the *maximal sub-language of K that is controllable w.r.t. A_u and $L(G)$* (see [Wonham 2003]). This language is denoted $\text{SupC}(A_u, K, L(G))$ and an algorithm computing it is described in [Wonham 2003]. The complexity of this algorithm is linear in the number of states of the Finite State Machine representing the control objective K and the system G.

2.2 Corrective Maintenance Process

Figure 1 describes the classical software development and maintenance process. Initially the system and its desired features are designed and implemented. These features are then tested until a high enough level of quality is reached, in order for the application to be deployed onto the users environment.

Despite going through a test phase, most software systems are deployed with some remaining bugs, generally unknown from the development team. Some of these bugs manifest themselves when the system is under usage and are reported by the users to the maintenance team. From there, this team aims to analyse and correct this bug in order to release as quickly as possible a new version of the system with increased quality. This increased quality is assessed with the testing of the modified application.

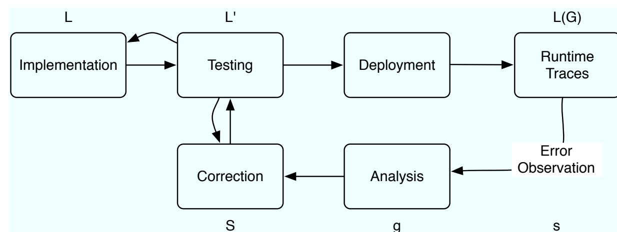

Figure 1: The software development and maintenance process.

In this paper, we consider errors that correspond to exceptions being raised at runtime and that are not handled in the application source code. When a trace leading to such an error is observed at runtime, the goal of the corrective maintenance is to modify the application behaviors in order to avoid future occurrences of this runtime exception.

The behaviors of the system correspond to software traces thatcan be observed through runtime monitoring and contain occurrences of method calls as well as exceptions. In [Gaudin et al. 2011], the authors describe how such monitoring can be automatically achieved for Java programs.

2.3 Mapping of Concepts

Figure 1 summarizes the mapping between concepts related to corrective maintenance and SCT.

[Application Model: L]. First, we consider that the behaviors of the system correspond a prefix-closed language $L(G)$ representing possible traces generated at runtime. The alphabet A of this language consists of method calls and

occurrences of exceptions. A language L included in $L(G)$ can be automatically extracted from the source code in order to model the behaviors of the system (see e.g. [Gaudin et al. 2011, Corbett et al. 2002]). However this model is only partial as it does not take into account exceptions that are not handled in the source code.

[Testing Phase: L']. Considering the testing phase of Figure 1, traces generated by running successful test cases are also modeled as a language L' over A. We assume here that the system is deployed after all the issues exhibited by test cases were resolved. This implies that L' corresponds to a set of desired behaviors of the system with $L' \subseteq L$. Bugs that lead to runtime exceptions being raised may however remain in the system. This corresponds to the case where bugs are not detected during the testing phase.

[Analysis: g]. Considering Figure 1 again, analysis is first required before correcting a system for which an error has occurred at runtime. This step aims for the maintenance team to understand why this error has occurred. This activity relates to the fields of diagnostics and program comprehension (see e.g. [Cornelissen et al. 2009]) and is outside the scope of this work. We restrict here the analysis phase to the determination of the set of possible system behaviors that can lead to a runtime exceptions. Given the observation of one faulty trace, we extrapolate this observation to a set of sequences.

Ideally only the relevant information to the system failure should be extracted from the observed sequence and taken into account for control. In order to answer this concern, we introduce the concept of *extrapolation functions*.

DEFINITION 1. *An extrapolation function over an alphabet A is a function $g: A^* \to \mathcal{L}(A)$ such that $\forall s \in A^*$, $s \in g(s)$.*

Given a sequence of events s over an alphabet A, an extrapolation of s, denoted $g(s)$, represents a set of sequences (i.e. a language) of which s is an instance. Extrapolation functions can for instance encode *the set of sequences that possess the same projection as s for a given projection function $P_{A'}$ with $A' \subseteq A$*, i.e. $g(s) = P_{A'}^{-1}(P_{A'}(s)) \in A'^*$.

Such a set is used in the SC Theory for partial observation problems (see e.g. [Cassandras and Lafortune 1999]). Another possible example of extrapolation corresponds to *the set of sequences that "finish like" s*. For $n \in \mathcal{N}$, $k \leq n$ and $s = \sigma_0.\sigma_1 \dots \sigma_n$,

$$g_k(s) = A^*.(\sigma_{n-k} \dots \sigma_n) \quad (1)$$

represents all the sequences whose last $k+1$ events are the last $k+1$ events of s. For instance, if a sequence s can be completed in $L(G)$ by an event σ representing a runtime exception, then $g_k^\sigma(s).\{\sigma\}$ represents sequences that end by $\sigma_{n-k} \dots \sigma_n \sigma$ in $L(G)$.

The case of extrapolation functions such as g_k is relevant to software systems. As stated in [Pan et al. 2009], software faults often occur relatively close to their observable symptoms, which themselves represent the ending of a sequence.

[Correction: Supervisor Synthesis]. the correction phase of Figure 1 is about modifying the system behaviors. Our approach consists of achieving this by controlling the system. This can be done by embedding a supervisor into the application, that will monitor the system execution and prevent the occurrence of some events when necessary[2]. This approach is attractive as SCT offers means to automatically synthesize supervisors from a system model and a control objective, hence making possible to automate the corrective phase of software maintenance.

3. SC AND MAINTENANCE FOR RUNTIME EXCEPTION AVOIDANCE

In the following of this paper, we present problems related to the automation of software correction, as control problems. We provide results about the existence of a solution to these problems.

In our setting, $L(G)$ represents the actual system behaviors and L only a partial model, which can be completed with the occurrence of runtime exceptions. Moreover, L itself represents a control objective as described in the BSCP of Section 2.1, i.e. the set of behaviors the system should be performing, without occurrence of runtime exceptions.

3.1 Corrective Problem (CorrectP)

CorrectP: Given an incomplete model L of a controlled system G, an observed system behavior of the form $s\sigma \in L.A_u$ that is not included in this model (i.e. $s\sigma \notin L$) and an extrapolation function g^σ, compute a maximal supervisor S on L that avoids the occurrence of sequences $s'\sigma$ of $L(G) \setminus L$ where $s' \in g^\sigma(s)$.

Intuitively CorrectP deals with the case where a sequence $s\sigma$ is observed during the execution of a controlled system and s is part of the system model L but $s\sigma$ is not. When an undesired sequence $s\sigma$ occurs with $s \in L$ but $s\sigma \notin L$, it is extrapolated by sequences of the form $(L \cap g^\sigma(s)).\{\sigma\}$. This set encodes that sequences of L that are similar to s are expected to be completed by σ and is then added to the model of the system L resulting into a new model of the system that takes into account newly observed behaviors, i.e. $L \cup (L \cap g^\sigma(s)).\{\sigma\}$. However, behaviors of the form $(L \cap g^\sigma(s)).\{\sigma\}$ are undesired. Preventing their futur occurrences is obtained by synthesizing a supervisor that only considers L as a control objective. We now introduce Proposition 1, which provides a solution to the CorrectP.

PROPOSITION 1. *Let A_u, A be two alphabets such that $A_u \subseteq A$. We also consider a prefix-closed language L over alphabet A, a sequence $s\sigma \in (A^*.A_u)$ and an extrapolation function g^σ. A solution to the CorrectP exists if $SupC(A_u, L, L \cup (L \cap g^\sigma(s)).\{\sigma\})$ is not empty. If it is the case, then*

$$SupC(A_u, L, L \cup (L \cap g^\sigma(s)).\{\sigma\})$$

is a solution to CorrectP.

Finally CorrectP rely on the definition of an extrapolation function g. However, a relevant extrapolation function may be difficult to determine with certainty. Therefore Subsection 3.2 considers the case where a range of extrapolation functions is available. For this case, Section 3.2 provides a result for the determination of the most appropriate extrapolation function.

3.2 Extrapolation Determination Problem (ExtrapP)

9[2]The reader can refer to [Gaudin et al. 2011, Gaudin and Bagnato 2011] for more details on how this can be achieved for Java programs.

In this subsection, we consider a similar problem to CorrectP: the model L of the system is incomplete and an undesired sequence $s\sigma$ is observed and can be extrapolated with extrapolation functions. However unlike for CorrectP, we consider a range of extrapolation functions $\{g_i^\sigma\}_{1\leq i\leq n}$ instead of a unique one as well as a set traces L' generated from successful test cases. The problem under consideration in this subsection is to determine which g_i^σ is the most appropriate extrapolation function in order to avoid the occurrence of extrapolated traces as well as ensuring a minimal set of behaviors given by s.

ExtrapP: Given an incomplete model L of a system G, i.e. $L \subseteq L(G)$, an observed system behavior of the form $s\sigma \in L.A_u$ that is not included in this model (i.e. $s\sigma \notin L$), a set of extrapolation functions $\{g_i^\sigma\}_{1\leq i\leq n}$ and a set of minimal behaviors L', determine g_i and compute a maximal supervisor S on L that allows sequences of L' and avoids the occurrence of sequences $s'\sigma$ of $L(G) \setminus L$ where $s' \in g^\sigma(s)$. $\diamond$

Solving ExtrapP corresponds to determining the index i for which the solution to CorrectP w.r.t. L, $s\sigma$, g_i^σ contains L' and is maximal. In other words, whenever it exists, a solution to ExtrapP is given by $\max\{\mathrm{SupC}_i | L' \subseteq \mathrm{SupC}_i\}$, where $\mathrm{SupC}_i = \mathrm{SupC}(A_u, L, L \cup (L \cap g_i^\sigma(s)).\{\sigma\})$. Therefore a solution to the ExtrapP exists if it exists $i \in \{1,\ldots,n\}$ *s.t.* $L' \subseteq \mathrm{SupC}_i$ and $\mathrm{SupC}_i \neq \emptyset$. Moreover if there is such a maximal SupC_i, then it is a solution to ExtrapP. Theorem 1 provides a condition under which there is such a maximal SupC_i.

THEOREM 1. *Let A_u, A be two alphabets such that $A_u \subseteq A$. We also consider a language L over alphabet A, a sequence $s\sigma \in L.A_u$ and a set $\{g_i^\sigma\}_{1\leq i\leq n}$ of extrapolation functions such that $i \leq j \Rightarrow g_i^\sigma(s) \subseteq g_j^\sigma(s)$. We denote*

$$SupC_i = SupC(A_u, L, L \cup (L \cap g_i^\sigma(s)).\{\sigma\})$$

If it exists $i \in \{1,\ldots,n\}$ such that $SupC_i \neq \emptyset$ and I denotes $\max\{i \in \{1,\ldots,n\} | L' \subseteq SupC_i\}$, then $SupC_I$ is a solution to ExtrapP.

Theorem 1 provides a condition under which a solution to ExtrapP exists, which corresponds to the case of ordered extrapolation functions g_i^σ. When the extrapolation functions under consideration can be ranked according to their generalizing power, then a solution to ExtrapP exists if it allows for successful tests and can be computed. This offers a maximal solution to the problem of reducing software behaviors in order to avoid the occurrence of a previously observed issue. Moreover, this solution is validated with respect to test cases.

4. CONCLUSION

This paper considers automatic adaptation of software behaviors in presence of a runtime exception. Our approach relies on the Supervisory Control Theory on Discrete Event Systems. Our work maps the corrective software maintenance process to supervisory control concepts. Automatic supervisor synthesis helps automate the software correction phase. This relies on observation of sequences violating the system model. This model is automatically extracted from source code. Unlike for traditional Supervisory Control problems, this model is only partial as it does not take into account runtime exceptions that are not handled in the source code. Our approach introduces the notion of *extrapolation functions* which encode sequences that follow a similar pattern as the faulty traces observed at runtime. Given an extrapolation function, we provide a solution to automatic supervisor computation. Moreover we consider the case of extrapolation functions with nested generalization power. We show that under this condition and in presence of traces generated by successful test cases, an optimal solution can be computed whenever it exists. As future works, we will investigate on other relevant extrapolation functions that capture interesting patterns for software traces. Results on this topic will provide our approach with different patching strategies, making it applicable to a wider range of software symptoms.

5. ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under the grant agreement FP7-258109. This work was also supported, in part, by Science Foundation Ireland grant 03/CE2/I303 1.

6. REFERENCES

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

[Corbett et al. 2002] CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PASAREANU, C., AND ZHENG, H. 2002. Bandera: Extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*. IEEE, 439–448.

[Cornelissen et al. 2009] CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L., AND KOSCHKE, R. 2009. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on 35,* 5, 684–702.

[Dobson et al. 2010] DOBSON, S., STERRITT, R., NIXON, P., AND HINCHEY, M. 2010. Fulfilling the vision of autonomic computing. *Computer 43*, 35–41.

[Gaudin and Bagnato 2011] GAUDIN, B. AND BAGNATO, A. 2011. Software maintenance through supervisory control. In *34th annual IEEE Software Engineering Workshop*.

[Gaudin et al. 2011] GAUDIN, B., VASSEV, E., HINCHEY, M., AND NIXON, P. 2011. A control theory based approach for self-healing of un-handled runtime exceptions. In *International Conference on Autonomic Computing (ICAC2011)*. Karlshruhe (Germany).

[Javassist] JAVASSIST. http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[Kephart and Chess 2003] KEPHART, J. O. AND CHESS, D. M. 2003. The vision of autonomic computing. *Computer 36*, 41–50.

[Pan et al. 2009] PAN, K., KIM, S., AND WHITEHEAD, JR., E. J. 2009. Toward an understanding of bug fix patterns. *Empirical Softw. Engg. 14,* 3, 286–315.

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