Technical Report No. NJU-SEG-2011-IC-001

Run-time Systems Failure Prediction via Proactive Monitoring

Pengcheng Zhang, Henry Muccini,

Andrea Polini and Xuandong Li

Postprint Version. Originally Published in:
International Conference on Automated Software Engineering 2011
Page(s): 484 - 487

Run-time Systems Failure Prediction via Proactive Monitoring

Pengcheng Zhang[1,2], Henry Muccini[3], Andrea Polini[4], and Xuandong Li[1]
[1]State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing – P.R.China 210093
[2]College of Computer and Information, Hohai University, Nanjing – P.R.China 210098
[3]Dipartimento di Informatica, University of L'Aquila, L'Aquila – Italy
[4]Computer Science Division, School of Science and Technologies, University of Camerino, Camerino – Italy
pchzhang@hhu.edu.cn, muccini@di.univaq.it, andrea.polini@unicam.it, lxd@nju.edu.cn

***Abstract*—In run-time evolving systems, components may evolve while the system is being operated. Unsafe run-time changes may compromise the correct execution of the entire system. Traditional design-time verification techniques difficultly cope with run-time changes, and run-time monitoring may detect disfunctions only too late, when the failure arises. The desire would be to define advanced monitors with the ability to predict and prevent the potential errors happening in the future. In this direction, this paper proposes CASSANDRA, a new approach that by combining design-time and run-time analysis techniques, can "look ahead" in the near execution future, and predict potential failures. During run-time we on-the-fly construct a model of the future k-step global state space according to *design-time specifications* and the *current execution state*. Consequently, we can run-time check whether possible failures might happen in the future.**

***Index Terms*—Proactive run-time Monitoring, Component-based Software Engineering, Software Evolution, Failure Prediction.**

I. Introduction

For current software systems that can dynamically change during run-time, such as service-based and component-based, existing verification techniques are inadequate to *prevent* failures to happen.

Traditional design-time verification techniques (like model checking) have been revised in order to cope with evolution (e.g., [4]) but they alone cannot fully support the analysis of run-time evolving systems. Run-time verification techniques, based on system monitoring approaches, have become fundamental to ensure the correctness of run-time evolving systems and have received increasing attention by the research community [7], [10]. New run-time monitoring techniques like those in [8], [11], [5] have been proposed, as well as techniques for combining both design-time and run-time verification [3], [13], [14]. However, they have the ability to detect errors only too late, that is, when a failure happens. What is desirable is instead a technique that, while applied to run-time evolving systems, can *predict* potential errors that may happen in the near future.

Towards this research goal, this paper proposes CASSANDRA, a novel proactive monitoring approach to predict potential failures by looking ahead the current execution state. To this end, CASSANDRA *combines design-time and run-time information* for *proactive run-time verification of dynamic component-based systems*. While traditional *passive* monitoring techniques enable the timely detection of possible problems or malfunctions by recording the execution traces and/or appropriate parameter values during the system execution, and *reactive* monitoring intervene and drive the system behavior based on the analysis results of the monitored behaviors. CASSANDRA uses a *proactive* monitoring that aims at predicting failures that may potentially happen in the near future [2].

The proposed proactive monitoring and verification technique uses run-time information to identify the current execution state, and checks whether the projection of a design-time model k steps ahead of the current state satisfies a set of wanted/unwanted properties. The execution state is captured by run-time traces obtained by monitoring the component-based system execution. Each component equips with a design-time model in the form of interface automata [6]. The properties we considered are interaction temporal properties among different components.

The approach, that is general in principle, is briefly described in Section II. The customization of CASSANDRA to the OSGi [1] component model and framework is presented in Section III. Section IV concludes the paper and provides a list of pointers for future work.

II. Cassandra, an Approach to Run-Time Failure Prediction of Dynamically Evolving Systems

In Greek mythology, Cassandra was the daughter of king Priam and queen Hecuba of Troy. She had the gift of prophecy but at the same time Apollo placed a curse on her so that no one would ever believe her predictions. The approach we are proposing in this paper somehow shares with her the prophecy capability, being it able to predict in the near future the possible occurrences of a system failure caused by the incorrect *integration* of the system composing components. Hoping that no curse will be placed on CASSANDRA, this information can be used in order to avoid and block the sequence of component interactions leading to the predicted failure.

[1]http://www.osgi.org

The adopted prediction technique is roughly inspired by model checking techniques where an operational model is explored to check if a desirable temporal property is satisfied or violated by the model itself. Nevertheless, differently from a traditional model checking approach, CASSANDRA explores design-time system models driven by information and events observed at run-time. The exploration strategy, starting from the current system state, looks k steps ahead and checks if there is a property violation in any path shorter than k steps and originating in the current state. Given its characteristics CASSANDRA can only detect violation of safety properties and bounded liveness properties where the look-ahead value is greater than the bounding value.

Given its characteristics, the approach is suitable for being applied to run-time check dynamically evolving component-based/service-based systems. For such kind of systems, static verification techniques present some limitations mainly related to the impossibility to precisely predict the characteristics of the run-time environment.

The CASSANDRA approach stands on the following assumptions:

- Each component to be integrated is augmented with a model of the interface suitable to be used for run-time checking;
- It is possible to map observed run-time events with the ones included in the formal model used for the run-time checking;
- The execution of one exploration step by the run-time checking algorithm is faster than the execution of the corresponding step by the real system. This assumption is necessary to keep the exploration synchronized with the real execution.

The algorithm we have implemented relies on a specification of the component/service behaviour based on the interface automata formalism [6]. This is a light-weight formalism to be used for describing the temporal interface behaviors of software components.

Definition 1: An *interface automaton* $P =< V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P >$ consists of the following elements:

- V_P is a set of states.
- V_P^{init} is a set of initial states. It is required that V_P^{init} contains at most one state. If $V_P^{init} = 0$, then p is called empty.
- A_P^I, A_P^O, and A_P^H are mutually disjoint sets of input, output and internal actions. The set of all actions are denoted by $A_P = A_P^I \cup A_P^O \cup A_P^H$.
- $T_P \subseteq V_P \times A_P \times V_P$ is the set of transitions.

If $a \in A_P^I$ (resp. $a \in A_P^O$, $a \in A_P^H$), then (v, a, v') is called an input (resp. output, internal) step. We denote by T_P^I (resp. T_P^O, T_P^H) the set of input (resp. output, internal) steps. The interface automaton p is *closed* if it has only internal actions, that is, $A_P^I = A_P^O = \emptyset$; otherwise, we say that p is *open*. An action $a \in A_P$ is enabled at a state $v \in V_P$ if there is a step $(v, a, v') \in T_P$ for some $v' \in V_P$. We denote with $A_P^I(v)$, $A_P^O(v)$, $A_P^H(v)$ the subsets of input, output, and internal actions that are enabled at the state v, and we let $A_P(v) = A_P^I(v) \cup A_P^O(v) \cup A_P^H(v)$.

Definition 2: An *execution fragment* of an interface automaton p is a finite alternating sequence of states and actions v_0, a_0, v_1, a_1,...,v_n such that $(v_i, a_i, v_{i+1}) \in T_P$ for all $0 \le i < n$. Given two states $v, u \in V_P$, we say that u is reachable from v if there is an execution fragment whose first state is v, and whose last state is u. The state u is reachable in p if there exists an initial state $v \in V_P^{init}$ such that u is reachable from v.

Since interface automata will interact through actions synchronization, the formal theory on how they compose needs to be defined. In particular the resulting automata can be calculated in linear time.

Definition 3: Two interface automata P and Q are *composable* if:

- $A_P^H \cap A_Q = A_Q^H \cap A_P = \emptyset$
- $A_P^I \cap A_Q^I = A_P^O \cap A_Q^O = \emptyset$

If P and Q are composable, then $shared(P, A) = A_P \cap A_Q = (A_P^I \cap A_Q^O) \cup (A_Q^I \cap A_I^O)$.

Definition 4: Let P and Q be two composable interface automata. Their product $P \otimes Q$ is the interface automata we define below:

$$V_{P\otimes Q} = V_P \times V_Q$$
$$V_{P\otimes Q}^{init} = V_P^{init} \times V_Q^{init}$$
$$A_{P\otimes Q}^I = (A_P^I \cup A_Q^I) \backslash shared(P, Q)$$
$$A_{P\otimes Q}^O = (A_P^O \cup A_Q^O) \backslash shared(P, Q)$$
$$A_{P\otimes Q}^H = A_P^H \cup A_Q^H \cup shared(P, Q)$$
$$\begin{aligned} &T_{P\otimes Q} = \\ &\{(v,u), a, (v',u)) \mid a \in A_P^H \cup (A_P^O \backslash shared(P,Q)) \wedge \\ &\quad (v, a, v') \in T_P\} \quad \cup \\ &\{(v,u), a, (v,u')) \mid a \in A_Q^H \cup (A_Q^O \backslash shared(P,Q)) \wedge \\ &\quad (u, a, u') \in T_Q\} \quad \cup \\ &\{(v,u), a, (v',u')) \mid a \in shared(P,Q) \wedge (v,a,v') \in T_P \wedge \\ &\quad (u, a, u') \in T_Q\} \end{aligned}$$

CASSANDRA is then based on a run-time checking algorithm that takes in input the interface automata for the various components, the look-ahead value k, the temporal property that should not be violated by the system and performs the following steps

1) The interface automata of the various components are composed on the fly to construct the tree of possible paths of length smaller than k,
2) When a system interaction event is detected the exploration is moved ahead of one step. In particular:
   a) All the subtrees rooted in a next state linked by an event different from the one detected are discarded;
   b) For each leaf in the subtree rooted in the next state linked by the detected event, the exploration is pushed ahead of one step;

c) In case a property is violated an alert event is generated specifying the trace leading to the property violation.

III. OSGi CASSANDRA Customization

While the theory outlined in Section II has a general idea in principle, we customize it into the OSGi component model and framework.

In OSGi, components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. For this purpose, each bundle is loaded with a different class loader and the "Hot-Deployment feature" enables to add new bundles to the platform without affecting the running services, that feature is very interesting. While there are a number of implementations of the OSGi Specification, we decided to use Equinox[2], published by the Eclipse Foundation, since it is a well-proven implementation which offers all necessary features of our approach.

In order to get run-time interaction information among OSGi bundles, we make use of the Equinox Aspects[3] project for the Eclipse platform that integrates aspect weaving, using AspectJ, into the Equinox OSGi implementation. It is possible to define and to deploy aspects inside of bundles independently of other bundles. To realize this, Equinox Aspects integrate the load-time aspect weaving functionality of AspectJ into the OSGi run-time itself. It takes care of finding aspects inside bundles and weaving these aspects into other bundles at load-time. Every time a class is loaded into the JVM, the applicable aspects are woven into it.

Figure 1 shows the main scenario on how to customize CASSANDRA into a OSGi-based dynamic component-based system. The OSGi bundle developers provide the architecture with a set of wrappers of bundles (each wrapper is equipped with source code, the corresponding interface automata model and the Aspect codes). The developers also provide a set of wanted/unwanted properties represented through the Property Sequence Chart (PSC) formalism [1]. Then PSCs are translated into Büchi automata for proactive monitoring. In order to make our monitoring approach have prediction ability, during run-time we take k steps looking ahead at the interface automata models to construct the temporary Global State Machine (GSM) according to the current run-time information. We on-the-fly construct the GSM with k steps looking ahead. Then, the approach can proactively check at step x whether possible failures could happen in the future $x+k$ steps. Consequently, the application users can take measures to avoid these failures.

The implementation is divided into two stages (design-time and run-time), with several steps each.

A. Design-Time

Step 1 - Architectural Specification

As the first step, the developer is required to describe the initial architecture of the evolvable OSGi application, in terms of the initial components the system will run. Any architecture

[2]http://www.eclipse.org/equinox

[3]http://www.eclipse.org/equinox/incubator/aspects/

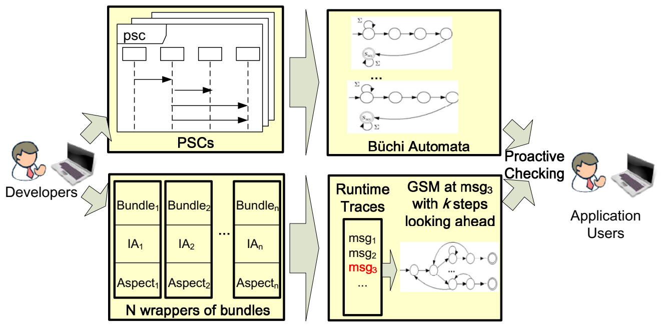

Fig. 1. Overview of OSGI CASSANDRA customization

description language that can describe components, their interfaces, and bindings with other components can be used in this step. We can use the tool Charmy [12] to help the developer to construct the system architecture.

Step 2 - Interface Formal Specification

The pre-condition of this step is that the interface automata must be consistent with the source codes. We have already defined the mapping between the source codes and the interface automata. A private method of source codes is just mapped into an internal action in the interface automata model while a public or protect method is mapped into an input or out action in the interface automata model. Then, the developers provide a wrapper of each basic OSGi bundle. The wrapper permits to add the original OSGi bundles with the representation of the interface automata, in XML format. The wrapper contains also aspect-based source codes that are necessary to instrument the bundle at run-time with mechanisms for notifying the invocations the bundle will receive, and to expose the interface automata so that can be retrieved by an external analyzer.

Step 3 - Properties Specification

A set of wanted/unwanted properties referring to several different OSGi bundles is provided at this step. The properties to be verified can be of various nature and can refer to a single component or to several components. This work focusses on interaction temporal properties, that are functional and temporal properties assigned to interacting components called property sequence charts(PSCs). PSCs are an extended graphical notation of a subset of UML 2.0 sequence diagrams, which is proposed in [1] to specify temporal properties. PSCs have two basic message types: *ArrowMSGs* and *IntraMSGs*. The *ArrowMSGs* The messages are classified into *Regular*, *Required* and *Fail* to define possible, mandatory and undesired interactions. *IntraMSGs* are used to describe *constraints* that restrict the future and past exchange of messages (*arrowMSGs*). *Constraints* are classified into *unwanted message constraints* and *chain constraints*. PSCs have four operators: *Strict*, *Parallel*, *Loop* and *Alt*. The *Strict* operator explicitly specifies a strict ordering between a pair of messages; no other message is allowed in between. The *Parallel*, *Loop* and *Alt* operators specify parallel merging (i.e., interleaving), iteration and alternative behavior, respectively.

B. Run-Time

Step 4 - Bundle Weaving

In this step, we will run the OSGi applications with aspects to get the run-time trace. Aspect-oriented computing [9] is continuing to increase in popularity. The modularity inherent in OSGi and Eclipse offers unique opportunities for managing and applying aspects by supplying them in bundles and directing their application to particular sets of bundles. The goal of EquinoxAspect is to allow developers to use the Equinox together with AspectJ by combining the benefits of both worlds. Using a load-time weaving extension it is possible to add AspectJ aspects to a bundle-based system just by putting them into general OSGi bundles.

Step 5 - Failure Prediction via Proactive Monitoring

A novel proactive run-time monitoring algorithm implemented the idea in Section II is proposed to carry on this step. The input of this algorithm is the updated run-time trace, a set of interface automata for the OSGi bundles, an interaction property represented by a PSC, and the k steps looking ahead. The output of this algorithm is yes or possible errors detected within the explored k steps ahead. The future k steps global state space is represented by all the possible future execution traces which are checked against the Büchi automaton translated by PSC. If the Büchi automaton goes to the final state, an error trace will be shown to the developers.

Step 6 - Bundles Run-Time Management

The OSGi bundles can be added or removed at run-time without rebooting. Please note that when a new bundle is added into the system, its corresponding interface automata and the new related properties are also synchronized at run-time. The dynamics nature of OSGi bundles are supported by Equinox Aspects as well. Equinox Aspects triggers update of woven bundles automatically when aspect bundles are installed, updated or uninstalled.

Step 7 - Proactive Monitoring Again

In this step, we will proactive re-monitor the system with the new property after OSGi bundles change. The systems do not need to restart, however, the proactive monitor approach needs to restart because new bundles are added into the system.

Please note when new bundles are added or old bundles are deleted from the systems, Steps 4-7 can be repeated to proactive monitor possible failures in the future.

IV. Conclusions and Future Work

This paper has proposed CASSANDRA, a novel proactive run-time monitoring approach for predicting failures in dynamic evolvable system. The approach combines design-time and run-time verification techniques as a way to predict possible failures that may happen in the near future. Eventually, the approach has been customized to the OSGi component framework.

A substantial list of future works for CASSANDRA is in our wish list. More specifically, we plan to: i) Apply CASSANDRA to real OSGi-based run-time evolving applications, ii) Measure the performance and overhead in large case study and further compare to our previous non-proactive monitoring tool [15], iii) take into consideration scenarios that require properties themselves to evolve during system evolution, iv) Study how much to look ahead (i.e., how to set-up the look ahead parameter properly), in order to derive a completely useful theory that could be used to guide how to monitor real systems, v) Manage time and probability in the interface automata, so to be able to predict more precise failures limited by timing and probabilistic properties, vi) Consider other types of run-time and parameterized properties which cannot be verified at design-time.

Acknowledgment

The authors of this work would like to thank Paola Inverardi for the inspiring discussions on combining design-time and run-time verification. The authors from Nanjing University are supported by the National Natural Science Foundation of China (No.90818022, No.61021062), the National Grand Fundamental Research 973 Program of China (No.2009CB320702), the Jiangsu Planned Projects for Post-doctoral Research Funds (Grant No.1101066C) and by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China (Grant No.11KJD520010). Andrea Polini's work is partially supported by the European Project FP7 IP 257178: CHOReOS.

References

[1] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical scenarios for specifying temporal properties: An automated approach," *ASE Jounral*, vol. 14, no. 3, pp. 293–340, 2007.

[2] A. Bertolino, D. Bianculli, I. Forgacs, and A. Polini, "Test framework specification and architecture," IST STREP PLASTIC Project, Tech. Rep. 4.1, 2007.

[3] E. Bodden and P. Lam, "Clara: Partially evaluating runtime monitors at compile time - tutorial supplement," in *RV*, 2010, pp. 74–88.

[4] S. Chaki, M. Clarke, Sharygina, and N. Sinha, "Verification of evolving software via component substitutability analysis," *Formal Methods in System Design*, vol. 32, no. 3, pp. 235–266, 2008.

[5] R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel, "Predictable dynamic plugin systems," in *FASE*, 2004, pp. 129–143.

[6] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC/SIGSOFT FSE*, 2001, pp. 109–120.

[7] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Trans. Software Eng*, vol. 30, no. 12, pp. 859–872, 2004.

[8] H. Goldsby, B. H. C. Cheng, and J. Zhang, "Amoeba-rt: Run-time verification of adaptive software," in *MoDELS Workshops*, 2007, pp. 212–224.

[9] G. Kiczales and et al, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242.

[10] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009.

[11] H. Muccini, A. Polini, F. Ricci, and A. Bertolino, "Monitoring architectural properties in dynamic component-based systems," in *CBSE*, ser. LNCS, vol. 4608, 2007, pp. 124–139.

[12] P. Pelliccione, P. Inverardi, and H. Muccini, "Charmy: A framework for designing and verifying architectural specifications," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 325–346, 2009.

[13] R. Purandare, M. B. Dwyer, and S. G. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010, pp. 270–285.

[14] J. Simmonds, S. Ben-David, and M. Chechik, "Guided recovery for web service applications," in *SIGSOFT FSE*, 2010, pp. 247–256.

[15] P. Zhang, Z. Su, Y. Zhu, W. Li, and B. Li, "WS-PSC Monitor: A tool chain for monitoring temporal and timing properties in composite service based on property sequence chart," in *RV*, 2010, pp. 485–489.