2012 19th Asia-Pacific Software Engineering Conference

Resynchronizing Model-based Self-adaptive Systems with Environments

†Linghao Zhang, *Chang Xu•, *Xiaoxing Ma, †Tianxiao Gu,† Xuezhi Hong, *Chun Cao, *Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
†{zlh.nju,tianxiao.gu,qiyingrunhua}@gmail.com, *{changxu,xxm,caochun,lj}@nju.edu.cn

***Abstract*—Self-adaptive systems are attractive due to their ability of adapting to changeable environments automatically. However, such systems may be subject to runtime failures when all environmental dynamics cannot be adequately considered at design time. When such failures occur at runtime, a system's internal adaptation logic usually has become inconsistent with its environment, according to our observation. We call this inconsistency sync-loss error. From our project experiences, we empirically identified a strong correlation between sync-loss error and system failure. This motivated us to fix sync-loss error in order to reduce failure for self-adaptive systems. In this paper, we formulate the problem of detecting sync-loss error, and present a framework ReSync to automatically fix sync-loss errors by resynchronizing a system with its environment. We experimentally evaluated ReSync on real robot cars with 20 different system versions. The evaluation reported promising results that ReSync can automatically recover our robot car systems from sync-loss errors, and significantly reduce the failure rate from 90.9% to 11.7-28.8%.**

***Keywords*-self-adaptive system, sync-loss error, resynchronization.**

I. Introduction

Self-adaptive systems are featured with the ability of dynamically adapting their behavior in response to environmental changes [1]. Such systems are increasingly deployed in practical environments, using sensors and actuators to interact with our physical world. Their interaction loop typically consists of four phases, namely, monitoring, detecting, deciding, and acting [2]. These phases intuitively explain how a self-adaptive system perceives the changes of its environmental conditions and decides to react to them.

In this paper, we focus on model-based self-adaptive systems, which are typically constructed using states and transition rules. Many popular example systems exist, including PhoneAdapter [3], GSM-oriented audio streaming system [4], intolerant message communication system [5], and SONY AIBO [6]. Such system resides at one state at a time, in which it executes its own business logic with a specific configuration. Its transition rules specify when this state is no longer suitable and how the system transits to another state, where it applies another configuration. This transition corresponds to an adaptation process, which may take multiple steps. Consider the PhoneAdapter application [3] for example. When a phone's user walks from his home to car, his phone is supposed to transit from a state of *Home* to another state of *Driving*. This is an adaptation process, which consists of two natural steps. First, PhoneAdapter detects that the user has left his home by its GPS sensor, and therefore changes the phone's profile to *Outdoor*. Then, PhoneAdapter detects a car's hand-free system by its Bluetooth sensor (implying that the user has entered his car), and therefore changes the phone's profile to *Driving*. However, such a simple two-step adaptation process can easily go wrong if the environmental changes are perceived in a way beyond the system's original anticipation. Consider that the user starts driving so quickly that PhoneAdapter detects the high speed of this user before it detects the car's hand-free system. Then, PhoneAdapter may think that the user is jogging (since he is moving quickly), and therefore changes the phone's profile to *Jogging* instead. After this wrong step of adaptation, PhoneAdapter may not work correctly as it now allows incoming calls at *Jogging*, which are actually not allowed at *Driving*. When the phone receives a call during the user's driving, it would accept it, leading to a system failure.

Such failures can be very common as a self-adaptive system can hardly consider all environmental dynamics adequately at design time. Therefore, preventing such failures at runtime is necessary. One promising approach is to detect system symptoms with which failures can easily occur, and fix these symptoms in order to prevent the failures. Then, the question becomes: How to define such symptoms and how to effectively detect and fix them? From our project experiences of several years on developing robot car systems, we empirically identified one major symptom with which failures can easily occur for self-adaptive systems. It is the loss of synchronization between a self-adaptive system and its environment. We call it sync-loss error for short. When a system tries to adapt to its environmental changes, it typically undertakes a multi-step adaptation process. These steps reflect the system's anticipation on how its environment changes and how the system reacts to such changes accordingly. Therefore, the system should keep its perception of its environment (i.e., internal state) always consistent with its actual environment (i.e., environmental conditions). We call this property synchronized. If the environment changes in a way beyond the system' original anticipation, the synchronization may be lost. In this case, we say that a sync-loss error occurs and the system may work incorrectly. Consider the aforementioned PhoneAdapter application. When the phone wrongly transits to a state of *Jogging* instead of *Driving* (i.e., synchronization is lost), any incoming call would be accepted, leading to a system failure (the user's safety is now threatened). Besides this example, we will also give an example of our robot car systems later in Section II.

•Corresponding author: Chang Xu.

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.62

To investigate the correlation between sync-loss error and failure in self-adaptive systems, we conducted a study on our real robot cars with 20 different system versions developed in the past two years. We encountered numerous failures (e.g., a car getting stuck or bumping into obstacles) when running these systems physically. From our experiments, we observed that 90.9% sync-loss errors led to system failure, and that 92.7% failures were accompanied with sync-loss error. This suggests a strong correlation between sync-loss error and system failure. Many pieces of existing work [3] [5] [6] [7] analyze for bugs inside a system's adaptation logic statically or in a way isolated from its environment. Since sync-loss errors occur at runtime, triggered by unanticipated environmental dynamics, such errors cannot be effectively detected by these existing techniques. It could be argued that a system can enhance each step of its adaptation process with various assertion checks to detect such errors. However, developers may be overwhelmed by this extra complexity and unable to focus on designing normal adaptation logic, leading to bad software engineering practice. In fact, even if one is willing to do so, exploring all environmental dynamics at design time may not always be possible.

We are thus motivated to dynamically fix sync-loss errors in order to prevent runtime failures for self-adaptive systems. These errors are to be fixed by resynchronizing a system with its environment again. We propose two mechanisms, namely, backward resynchronization and forward resynchronization. The former means that when a sync-loss error occurs, a system undoes the actions that have been taken in its current adaptation and goes back to an earlier stable state, where this sync-loss error has not occurred. Then the system can restart this adaptation process. Forward resynchronization means that when a sync-loss error occurs, the system goes forward to find a stable state in its future adaptation path, where this sync-loss error does not exist. This is to skip the current, problematic adaptation and recover its adaptability for new environmental changes.

However, system resynchronization is non-trivial. First, sync-loss error is new and has not been discussed and formally defined in existing work. Second, a self-adaptive system may involve actions related to its hardware devices, and there is no guarantee for these actions to complete successfully as required in the resynchronization. To address these challenges, we propose a novel resynchronization framework ReSync. ReSync includes three parts: (1) An extended A-FSM model (as compared to the original A-FSM model by Sama et al. [3]), which enables the formal definition of sync-loss error and its effective detection; (2) A static analysis algorithm for deciding whether a system owns the ability of resynchronizing itself with its environment (or called resync-ability for short); (3) A runtime support that aids a system to automatically detect and recover from sync-loss errors. We evaluated ReSync using real robot cars with 20 different software versions and two different hardware platforms. We obtained preliminary, yet promising results that: (1) ReSync could support automatic resynchronization for a system with its environment when any sync-loss error occurs; (2) By doing so, the system's failure rate was greatly reduced from 90.9% to 11.7% and 28.8% for backward resynchronization and forward resynchronization, respectively.

The rest of this paper is organized as follows. Section II introduces background knowledge and explains the modeling of self-adaptive systems. Section III presents the concepts and modeling extensions used in our ReSync framework. Section IV introduces our static resync-ability analysis algorithm and runtime support for automatically detecting and recovering from sync-loss errors. Section V evaluates ReSync using our robot car systems and discusses experimental results. Finally, Section VI discusses and compares our work to related work, and Section VII concludes this paper.

II. Preliminary

In this section, we introduce the modeling of self-adaptive systems. We use our robot car systems as illustrative examples, but the modeling approach is applicable to other self-adaptive systems.

A. Robot Car Systems

We consider two types of robot cars, namely, tank car and tricycle car. The detail of robot car hardware information can be found at our project site [8]. Both cars are equipped with actuators (e.g., motors) and sensors (e.g., ultrasonic sensors). For each orientation (i.e., front, back, left, and right), a set of ultrasonic sensors are installed for measuring the distance between the car and the obstacle ahead at this orientation. A digital compass is installed for measuring the car's walking orientation. Besides, there are two optical sensors installed at a car's two motors for measuring how far the car has walked. The tricycle car has only one motor to control its walking, and its controllability is weaker than that of the tank car. For three typical actions (i.e., *MovingForward*, *TurningLeft*, and *TurningRight*), the tricycle car has to conduct them unstably. This may affect the effective completion of these actions, as discussed later.

B. Running Example

We use our robot car systems as a running example to explain the modeling of self-adaptive systems, as well as sync-loss errors that may encounter at runtime.

A robot car considers exploring an unknown area without bumping into any obstacle. Now it encounters a door obstacle ahead when it walks along a wall. When the door obstacle is detected by the car's sensors, the car has to bypass it in a way as illustrated in Fig. 1. This process can be seen as a multi-step adaptation (from Step i to Step v).

Step iii is, however, a challenging one. After a *TurningRight* action, the car goes from Step ii to Step iii. It goes straight ahead, expecting detecting and then bypassing the door obstacle on its right side. When its measured obstacle-distance data on the right side experiences a fluctuation (i.e., from a large to small and then to large value again), the car knows that it

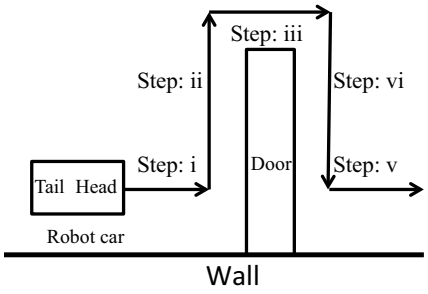

Fig. 1. Illustration of a five-step adaptation for a robot car to avoid bumping into a door obstacle.

should have bypassed the door. Then it can turn right and go to Step iv.

The above process is intuitive, but it, unfortunately, does not always work as expected. Sometimes, the car may fail to detect the door-bypassing event. As a result, the car would keep walking straight ahead, still expecting detecting and then bypassing the door (although it is already bypassed). What deserves noting is that since the car still resides at Step iii (expecting going to Step iv), it is not supposed to detect other obstacles. As such, it may bump into other obstacles ahead if any. It looks like that the car has lost its adaptability (unable to automatically avoid obstacles). We give some recorded videos for such scenarios at our project site [8] for readers' reference. There can be several reasons for this result: (1) The car walks too fast such that it misses detecting the door between its two consecutive sensor readings; (2) The door is too thin to be effectively detected; (3) Sensor noises prevent the detection of this door-bypassing event; (4) The door is suddenly closed.

Despite different reasons for this failure to occur, the underlying problem is the same. That is, when the car is trying to look for the door at its right side, the door can never be found (since it is already bypassed), the car actually gets stuck in Step iii. Its expectation on its environment becomes inconsistent with its actual environment. In this case, we say that the car has lost the synchronization with its environment, and a *sync-loss* error occurs.

To recover a system suffering from a sync-loss error, the system needs to synchronize with its environment again. We call this process resynchronization. For our robot car, it can be done by undoing its current adaptation and going back to an earlier stable point (e.g., the start point of Step iii or even prior to Step ii). Then the car restarts its obstacle-avoiding adaptation. Or, it may also choose to ignore the current, problematic adaptation and quickly go forward to a future, stable point after this adaptation (assuming that it has already completed this adaptation). By doing so, the car can have a chance to recover its adaptability for avoiding new obstacles encountered later.

C. Modeling Self-adaptive Systems

Various approaches have been proposed for modeling self-adaptive systems, such as Petri Net, Hybrid Automata, UML, and FSM. For those systems that are built on top of states and transition rules, we follow a popular A-FSM approach [3] proposed in recent years, and extend it for defining, detecting, and fixing sync-loss errors.

Given a self-adaptive system, its model consists of a set of states $\mathcal{S}$ and a set of adaptation rules $\mathcal{R}$ (i.e., transition rules). The system resides at one state at a time, and it can transit to another state through the triggering of certain adaptation rules (e.g., some rules may specify a state-transiting action). An adaptation rule is trigged when this rule's associated condition is satisfied (i.e., its truth value is evaluated to true). Among all states in $\mathcal{S}$, one state $s_0 \in \mathcal{S}$ is the initial state at which the system resides when it starts running.

Let $\mathcal{C}$ be the set of context variables that take sensory data values from the environment. For example, our robot car system may include context variables *Front.Distance* and *Right.Distance*, indicating the distance between the car and its obstacle at its front side and right side, respectively. Let $\mathcal{P}$ be the set of propositional predicates defined upon these context variables and connected by conjunction, disjunction, and/or negation operators. For example, our robot car system may define a predicate *FrontObstacleDetected* as being true when context variable *Front.Distance*'s value is less than 30 (i.e., *Front.Distance* < 30).

The set of adaptation rules $\mathcal{R}$ is defined as: $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S} \times \mathcal{A}^*$, where $\mathcal{A}$ is the set of actions. These actions include physical ones like *MovingForward*, *TurningLeft*, and *TurningRight*, and digital ones like transiting to a new state.An adaptation rule is defined as $r = (s, p, s', act^*)$, where $s, s' \in \mathcal{S}$, $p \in \mathcal{P}$, and $act^* \in \mathcal{A}^*$ ("$*$" means zero or more than zero). Predicate p is the triggering condition of this rule.

Fig. 2 illustrates the A-FSM model for our robot car system. There are a total of six states (1, 2, 3.1, 3.2, 4.1, and 4.2). Each state represents not only a specific configuration under which a self-adaptive system works but also the system's perception of its environment. For example, States 3.1 and 3.2 correspond to Step iii, one for the situation that the door has not been detected yet and the other for the situation that the door has been detected. Arrows in Fig. 2 represent state transitions (source and target states can be the same). Labels on these arrows informally describe the triggering conditions of these transition rules as well as their associated actions for taking when these conditions are satisfied.

With this model, the car system runs in a loop. It starts with the initial State 1. When its environmental conditions change (e.g., an obstacle is detected), related context variables are updated accordingly based on newly collected sensory data. Then all rules associated with the car's current state are evaluated to check whether any one is triggered. For ease of presentation, we assume that only one rule can be triggered at a time. We note that the multi-triggering issue does exist but it is out of the scope of this paper. An extension with priority can address it [3] [9], and therefore we omit its discussion in this paper. When a rule is triggered, the system executes its associated actions.

This A-FSM modeling approach is expressible. We base our work on this approach for generality. However, we note that

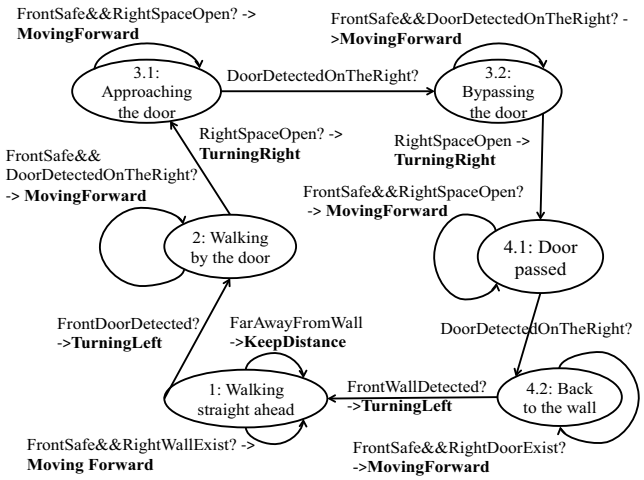

Fig. 2. An FSM model for robotcar (State 1 is the initial state).

such model is subject to sync-loss errors at runtime and has no ability of detecting and recovering from such errors. We in the following explain how to extend this model to enable detecting sync-loss errors at runtime and resynchronizing a system with its environment automatically.

III. ReSync Methodology

Our ReSync methodology consists of three parts. The first part is an extension to the existing A-FSM modeling approach, which includes three features to be presented in this section. The other two parts are a static analysis tool and a runtime support to be presented in the next section.

A. Sync-loss Error and Guard Condition

The first feature is the use of guard condition for defining and detecting sync-loss error. Sync-loss means that a system's perception of its environment is no longer consistent with its actual environment. Formally defining this inconsistency is necessary for detecting when a sync-loss error has occurred. We propose the notion of sync-loss based on our previous work on context inconsistency [10] [11]. We extend the earlier A-FSM modeling approach by associating a guard condition with each state in the model. Given a state, its guard condition specifies this state's expectation on the environment in terms of a predicate over context variables. When a self-adaptive system resides at this state, the state's guard condition should keep holding (i.e., always evaluated to true). Then, we consider this system always synchronized with its environment. Otherwise, we say that a sync-loss error has occurred.

Let $Guard(s)$ be the guard condition associated with a state s: $Guard(s) \in \mathcal{P}$. For example, consider State 3.1 in Fig. 2 for our robot car system. At this state, the car just turns right and starts walking straight ahead, expecting detecting and bypassing a door obstacle on its right side. We can specify a guard condition for this state as *FrontSafe && LeftSafe && NotWalkingTooFar*. It means that there should be no obstacle ahead and also on its left side, and the robot has not walking too far since the car starts walking straight ahead at this state. This is based on the expectation that the car should not take too long for detecting the door obstacle on its right side (since it is supposed to be there). If anything bad occurs, the guard condition may become violated, indicating that the environment is actually no longer as expected, i.e., a sync-loss error occurs. This gives the system a signal to resynchronize it with its environment before anything worse occurs (e.g., directly bumping into other obstacles ahead).

B. Compensable and Retriable Attributes

The second feature is the use of compensable and retriable attributes for a system's resynchronization with its environment. Resynchronizing a system with its environment concerns undoing the actions in its current adaptation and retrying them (i.e., backward resynchronization), or going forward to the completion of its current adaptation quickly (i.e., forward resynchronization). It is clear that such resynchronization relies on whether the actions involved in the current adaptation can be undone or retried. We formulate the answers to these two questions as "compensable" and "retriable" attributes. The names of these two attributes have been inspired from existing work on transactional work-flows [12], [13], [14], [15].

A compensable action can always be undone successfully. If a system decides to undo the current adaptation, then all actions that have been taken in the current adaptation should be undone. They can be undone successfully as long as they are all compensable. If an action is a digital action like a state-transiting action, it is certainly compensable. However, if the action is a physical one like *TurningLeft* for a robot car system, it may not always be compensable. For our tank car, this action is compensable as the car can always bring itself back to its last position prior to taking the action *TurningLeft*. However, for our tricycle car, this action is not compensable as the car has a weak control on its motor and it can easily fail to undo this action.

A retriable action can be eventually completed no matter what exception occurs during its trying process. In other words, this action may be retried in different ways but its completion is guaranteed. For a digital action, it is always retriable. For a physical action, it depends. For example, our robot cars may not conduct actions *TurningLeft* and *TurningRight* precisely. A car's resulting orientation after taking these actions may differ from its supposed orientation by some degrees or even wrong due to hardware limitation or physical exception. However, when equipped with a digital compass, our tank car can measure the actually rotated degrees and therefore guarantee to complete these actions successfully eventually. As such, these two actions are retriable for our tank car. However, for our tricycle car, they are not retriable due to it weak control on its motor.

In our extended A-FSM modeling approach, we allow each used action to be denoted with two attributes, showing whether or not this action is compensable and retriable. For each action $act \in \mathcal{A}$, the attributes of this action are denoted as: $Attributes(act) \in 2^{\{compensable, retriable\}}$. We assume the compensation action act for a compensable action act to be

available and supported by the underlying middleware infrastructure for the development practice on it. Similarly, for a retriable action act, we assume that the underlying middleware infrastructure can complete it eventually by retrying it many times or via different ways. With these two types of denotation, whether or not a given A-FSM model can be guaranteed to resynchronize with its environment becomes decidable statically. We discuss how to do it later.

C. Stable State and Transient State

The third feature is a refined classification of the states used in an A-FSM model. According to the nature of each self-adaptive system, we allow the states of a system to be categorized into two types: *stable state* and *transient state*. When a system resides at a stable state, it means that the system considers its current configuration suitable for its environment and there is no need to adapt itself. When the environment changes (e.g., a door obstacle appears ahead), the system needs to adapt itself to cope with this change (e.g., trying to bypass this door obstacle). This can be a multi-step adaptation, which corresponds to a series of transient states. At each transient state, the system adapts itself a little bit and checks for its environment for the next step of adaptation (e.g., a robot car takes five steps to bypass a door obstacle). These series of transient states connect two stable states. When the system finishes this multi-step adaptation, it arrives at the target stable state, at which it considers its current configuration suitable again for its environment (e.g., after bypassing the door obstacle, the robot car walks along its right-side wall again).

Let $\mathcal{S}_s$ be the set of stable states and $\mathcal{S}_t$ be the set of transient states. We have $\mathcal{S} = \mathcal{S}_s \cup \mathcal{S}_t$ and $\mathcal{S}_s \cap \mathcal{S}_t = \emptyset$. In our ReSync framework, stable states can be used as the target states for backward or forward resynchronization, as explained later.

As a summary, we extend the existing A-FSM modeling approach with three features: (1) Each state is associated with a guard condition for defining and detecting sync-loss error; (2) Each state is associated with a type, showing whether it is a stable state or transient state; (3) Each action is associated with two attributes, showing whether or not it is compensable and retriable. All these new features will be used in our ReSync framework to conduct automated resynchronization between a system with its environment.

D. ReSync Methodology

We now present the methodology used in our ReSync framework. Given a self-adaptive system specified by our extended A-FSM model, we define an adaptation path as a sequence of states connected by adaptation rules. Formally, the set of all adaptation paths $\mathcal{PATH}$ is defined as follows:

$$\mathcal{PATH} = \{s_0 \overset{r_0}{\rightarrow} s_1 \overset{r_1}{\rightarrow} ...s_i \overset{r_i}{\rightarrow} ... \overset{r_{n-1}}{\rightarrow} s_n | \\ r_i \in \mathcal{R}, s_i \in \mathcal{S}, n \geqslant 0\}$$

Generally, a self-adaptive system starts from its initial state, which is usually a stable state. When its environment changes, the system may need to adapt itself according to its triggered adaptation rules, and then arrives at another stable state. The execution trace of the system can thus be described as an adaptation path. When any sync-loss error occurs on this path, the system should be resynchronized with its environment via either backward resynchronization or forward resynchronization. In our ReSync methodology, when any sync-loss error occurs, the system is being resident in the process of a multi-step adaptation, i.e., residing at some transient state. Therefore, its resynchronization goal should be to resynchronize this system to an earlier stable state in its past adaptation path, or to a new stable state to appear in its future adaptation path.

We note that whether or not a system can conduct resynchronization depends on the state at which it resides when a sync-loss error occurs. A state can be *backward resync-able* (i.e., able to go back to an earlier stable state) or *forward resync-able* (i.e., able to go forward to a future stable state) or none (i.e., neither is supported) or both (i.e., both are supported). Such a property is called the *resync-ability* of this state. A state's resync-ability is decided based on whether its associated adaptation path is compensable or retriable, which is further decided by whether the involved actions on this path are compensable or retriable. We explain in detail below.

Rule's compensability and retriability. If all actions associated with an adaptation rule are compensable (retriable, or both), we say that this rule is compensable (retriable, or both).

Path's compensability and retriability. If all rules on an adaptation path are compensable (retriable, or both), we say that this path is compensable (retriable, or both).

State's backward resync-ability. Given a state s, for each stable state $s_i \in \mathcal{S}_s (s_i \neq s)$ such that s is reachable from s_i, if each possible adaptation path from s_i to s is compensable, we say that state s is backward resync-able.

State's forward resync-ability. Given a state s, if there exists a stable state $s_j \in \mathcal{S}_s (s_j \neq s)$ and s_j is reachable from s such that there exists one adaptation path from s to s_j and this path is retriable, we say that state s is forward resync-able.

The state's resync-ability is critical for deciding whether a self-adaptive system can be resynchronized with its environment and how it can be done. When a system resides at a backward resync-able state and a sync-loss error occurs, it can always compensate all its conducted actions in the current adaptation and go back to an earlier stable state, where it is expected to synchronize with its environment again. Since this state is backward resync-able state, it means that no matter what state from which the system has adapted to the current state, this adaptation path must be compensable. Therefore, the system can always go back (i.e., roll back) to this earlier stable state.

On the other hand, if a self-adaptive system resides at a forward resync-able state and a sync-loss error occurs, it can always go forward by quickly conducting the actions that have not been conducted in the current adaptation and arrive at a

```
Algorithm: Backward resync-ablity analysis for states.
Input: An extended A-FSM model M.
Output: Backward resync-ablity analysis results.
1.  Enum[] property;
2.  State[] states = M.ReadStates();
3.  Rule[] rules =M.ReadRules();
4.  for each state s_i in states
5.      Rule[] rule_s = GetRulesTargetAt(s_i);
6.      if (each rule in rule_s is compensable) then
7.          property[i] = "backward resync-able";
8.      else
9.          property[i] = "not backward resync-able";
10.     end if
11. end for
12. while (true)
13.     boolean modify = false
14.     for each rule_i from s_u to s_v in rules
15.       if (property[u] == "not backward resync-able") then
16.           property[v] = "not backward resync-able"; modify = true;
17.       end if
18.     end for
19.     if (modify == false) then
20.         return property;
21.     end if
22. end while
```

Fig. 3. Algorithm to check whether a state is backward Resync-able.

future stable state, where it is expected to synchronize with its environment again. Since at least one retriable path exists from this forward resync-able state to another stable state, the system can always proceed to that stable state.

Therefore, our ReSync methodology is able to decide whether or not a given self-adaptive system model can resynchronize itself with its environment when any sync-loss error occurs, as well as suggesting how it can be done. A rigorous static analysis algorithm of deciding the resync-ability for a self-adaptive system and a dynamic runtime support for guiding how the resynchronization is done are presented in the next section.

IV. ReSync Framework

In this section, we present our ReSync framework. We assume that the software part of the system runs on top of a hardware controlling middleware, which wraps the details of controlling and communicating with the actuators and sensors in the system. The ReSync framework includes three parts: an extended A-FSM model, a static analysis tool, and a runtime support. User applications (i.e., software part) are expressed using our extended A-FSM models, which have been explained in the last section. The static analysis tool decides the resync-ability for all states in a self-adaptive system. Based on the analysis results, the runtime support dynamically monitors the system and resynchronizes it with the environment when any sync-loss error occurs. In the following, we elaborate on the resync-ability analysis and runtime support. The detail of our ReSync framework and runtime support can be found at our project site [8].

A. Resync-ability Analysis

The resync-ability of a self-adaptive system is analyzed for all states of this system. This includes both backward resync-ability and forward resync-ability analyses.

Backward resync-ability analysis. We give the state's backward resync-ability analysis algorithm in Fig. 3. We use an array of enumeration type to represent the resync-ability properties of all states (*property[i]* represents the property of the *i-th* state $s_i \in \mathcal{S}$). Method *ReadStates* at Line 2 returns all states from a given self-adaptive system's extended A-FSM model. Method *ReadRules* at Line 3 returns all adaptation rules except those adapting the system from one stable state to itself (i.e., no actual state transition). Method *GetRulesTargetAt*(s_i) at Line 5 returns those rules whose target state is exactly s_i. Give a state s_i, if all rules whose target state is this state are compensable, then "backward resync-able" is assigned to *property[i]* at Line 7. Otherwise, "not backward resync-able" is assigned at Line 9. After that, the algorithm executes a loop from Lines 12 to 22. In the loop, we repeat the following work until a fixed point is reached: if there exists a rule that transits the system from state s_u to state s_v, and state s_u has a "not backward resync-able" property, then this property is propagated to state s_v. Finally, the algorithm returns the property array as the backward resync-ability analysis results. Although the backward resync-ability of a state is defined on the backward resync-ability property of paths, our algorithm instead takes advantage of translation relations between backward resync-able states as shown from Line 14 to 17. This facilitates an easier computation.

Forward resync-ability analysis. To analyze whether a state s is forward resync-able, we apply a breadth-first search (BFS) algorithm to check whether or not there exists a retriable path from state s to another stable state in the extended A-FSM model. If such a path exists, then state s is forward resync-able and this path is recorded for later use when resynchronizing the system with its environment. Otherwise, state s is not forward resync-able. Since this algorithm is straightforward, we omit its details due to space limit.

Based on these two algorithms, we implemented a static tool for analyzing the resync-ability of a given self-adaptive system. The input of the tool is an extended A-FSM model of the system under analysis. Its output is the resync-ability analysis results for all states in the model (including both backward and forward resync-ability analysis results).

By using the tool, developers can learn whether or not their designed model for a self-adaptive system is safe in terms of resync-ability. If all states of a system model are backward resync-able, it means that no matter at which state the system encounters a sync-loss error, it can always conduct backward resynchronization to recover from this error, thus avoiding failure. Similarly, if all states are forward resync-able, the system can always conduct forward resynchronization no matter at which state it encounters a sync-loss error. As such, resync-ability greatly increases a self-adaptive system's dependability, and we make it statically decidable.

B. Runtime Support

The runtime support in our ReSync framework contains three components: trace recorder, state monitor, and recovery executor.

When a self-adaptive system starts running, the trace recorder keeps logging its adaptation path all along. The state monitor keeps monitoring the values of all context variables concerned in the system model. When any context variable is updated due to new sensory data from the environment, the state monitor evaluates the guard condition for the system's current state. If the guard condition is violated (i.e., evaluated to be false), a sync-loss error is detected. Then the recovery executor would start a certain resynchronization mechanism (backward or forward) according to earlier analyzed resync-ability results. A resynchronization is considered successful when its target stable state is reached and this state's guard condition is satisfied. If not successful, the system would invoke the resynchronization again from the new state.

V. Evaluation

In this section, we evaluate our ReSync framework using our real robot car systems. The evaluation aims to answer the following four research questions:

RQ1: Can it be possible for sync-loss error to be one major reason for the failures of self-adaptive systems?

RQ2: Is our static analysis tool useful for deciding the resync-ability of self-adaptive systems?

RQ3: Can ReSync successfully resynchronize a self-adaptive system with its environment when a sync-loss error occurs at runtime?

RQ4: Does ReSync help reduce the failure rate for self-adaptive systems?

A. Experiment Subjects

Our experiments were conducted on our self-adaptive robot cars with 20 different system versions and two types of hardware platforms. These system versions (i.e., software part) have been built over more than two-year development. They were individual versions from different developers, which include research staffs, technicians, and graduate students. All these system versions aim to explore an unknown area with the ability of automatically avoiding detected obstacles. The avoidance strategy is decided based on the sensory data from the cars. Due to different development processes, these system versions have been built in very different ways. For example, their built-in states have a varying number from 2 to 9, and the number of transition rules ranges from 6 to 14, combined with various guard conditions, triggering conditions, and actions. Therefore, they are a good representative of different programs for our experimental purposes.

For the hardware part, we have two types of robot cars, namely, tank car and tricycle car, as explained earlier. When no physical exception occurs at runtime, both robot cars can complete actions like *MovingForward*, *TurningLeft*, and *TurningRight* successfully. When any physical exception occurs and destroys the current action, only tank car can retry and compensate this action. Then, all actions carried out by the tank car are retriable and compensable. For the tricycle car, action *MovingForward* is retriable but not compensable. Both actions *TurningLeft*, and *TurningRight* are neither retriable nor compensable. These differences have been caused by the physical controlling mechanism of the tricycle car, and would affect the resynchronization process in our experiments later.

B. Experiment Setup

We designed ten test scenarios for our robot car systems. Some of these scenarios are for testing normal functionalities like avoiding an obstacle detected ahead. Some scenarios introduce environmental dynamics, which may not have been considered at the design time of the 20 system versions. Consider the illustration in Fig. 1. We may suddenly close the door (i.e., obstacle disappeared suddenly) when a car is bypassing the door obstacle. For this car, the obstacle would look like "disappearing" suddenly or too thin to be detected effectively. We may also use a very thick obstacle to replace the original door obstacle. Then the car would have to walk longer than originally expected to bypass the obstacle.

To answer research question RQ1, we ran 20 system versions combined with two car types in ten test scenarios. We got a total of 400 runs. We recorded whether the car failed (e.g., getting stuck or bumping into an obstacle) and whether any sync-loss error occurred then. Then we analyzed the correlation between sync-loss error and system failure. This part of experiments is named as E1.

To answer research question RQ2, we applied our static analysis tool to 20 system versions combined with two car types. We thus obtained the resync-ability analysis results for all states in these 40 combinations. To evaluate the usefulness of these analyses, we ran the 40 combinations under ten test scenarios. We got a total of 400 runs. Among them, some runs under the same combinations encountered sync-loss errors at the same states, while in some runs sync-loss errors did not occur. Since we are interested in the resync-ability of states, we merged such runs when they encountered sync-loss errors at the same states for the same combination. Finally, we got a total of 80 runs. For them, we used ReSync to resynchronize the system with its environment with both backward resynchronization and forward resynchronization. So totally we had 160 runs in this part of experiments (named as E2). For experimental purposes, we randomly injected physical exceptions to car actions with a rate of 50%. We observe whether each of such runs can successfully resynchronize the system with its environment even if random exception may occur at runtime, and compare it to each run's associated state's resync-ability analysis result.

To answer research questions RQ3 and RQ4, we ran 20 system versions with our tank car on ten test scenarios. This is because all actions for our tank car are both compensable and retriable, and this facilitates us to study the usefulness of our resynchronization approach since it is our focus. We tested both backward resynchronization and forward resynchronization. These two parts of experiments are named as

	#Run	Failure rate (%)
Sync-loss error observed	222	90.9% (202/222)
Sync-loss error not observed	178	8.9% (16/178)

Fig. 4. Results of experiments E1.

E3 and E4. We observe whether ReSync is able to resynchronize a system with its environment when any sync-loss error occurs, and whether this resynchronization helps reduce the system failure rate accordingly.

C. Experimental Results and Analysis

We report and analyze experimental results for the four research questions in turn below.

RQ1: Can it be possible for sync-loss error to be one major reason for the failures of self-adaptive systems? We give the results of E1 in Fig 4. We obtained a total of 400 runs, and observed sync-loss error in 222 of them and failure in 202+16=218 of them. We found that when any sync-loss error occurs, there is 90.9% (202/222) probability that a system failure also occurs. On the other hand, when any system failure occurs, there is 92.7% (202/218) probability that a sync-loss error occurs as well. This suggests a strong correlation between sync-loss error and system failure. Therefore, sync-loss error can be one major reason explaining why system failure would occur. This implies that fixing sync-loss errors can be an effective way to prevent system failures.

RQ2: Is our static analysis tool useful for deciding the resync-ability of self-adaptive systems? We give the results of experiments E2 in Fig. 5. As explained earlier, we consider 20 system versions combined with two car types, i.e., a total of 40 combinations. Our static analysis tool analyzes the resync-ability for all states in these 40 combinations. Finally, we got a total of 160 runs for analyzing the usefulness of our resync-ability analysis results. Among these 160 runs, 80 are associated with the tank car, for which the concerned states are both backward resync-able and forward resync-able. The other 80 runs are associated with the tricycle car, for which the concerned states are neither backward resync-able or forward resync-able.

From the experiments, we observe that the tank car can always resynchronize itself with its environment via backward resynchronization (100% successful rate), and resynchronize with 87.5% successful rate via forward resynchronization. For the tricycle car, both rates are down 0% and 27.5%, respectively. This shows that our static resync-ability analysis results can give useful predictions on whether a certain combination of system version and car type is able to resynchronize with its environment in a backward or forward way. Still, we note that the successful rate of resynchronization depends on the property of the concerned state as well as the actual environment then. Some discussions are given later in the next section.

RQ3 & RQ4: Can ReSync successfully resynchronize a self-adaptive system with its environment when a sync-loss error occurs at runtime, and help reduce the failure rate for self-adaptive systems? We give the results of experiments E3

Car type	Backward resynchronization			Forward resynchronization		
	#runs	State's resync-ability	ReSync succ. rate	#runs	State's Resync-ability	ReSync succ. rate
Tank car	40	b/r	40 (100%)	40	f/r	35 (87.5%)
Tricycle car	40	not b/r	0 (0%)	40	not f/r	11 (27.5%)

Fig. 5. Results of experiments E2.
b/r: backward resync-able; f/r: forward resync-able; succ.: successful.

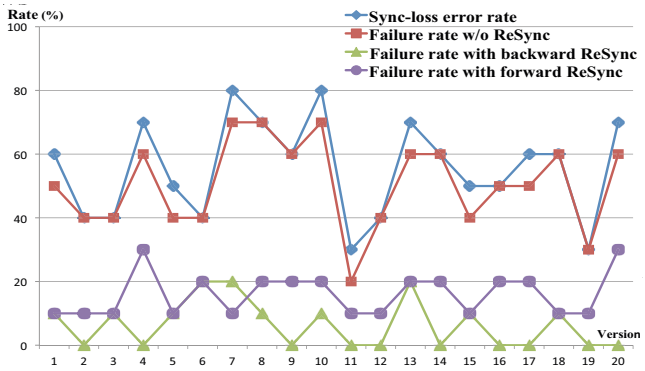

Fig. 6. Results of experiments E3 and E4.

and E4 in Fig. 6. All rates were calculated based on the ratio of the error/failure number observed in ten test scenarios against all. Our ReSync could automatically fix 100% sync-loss errors via backward resynchronization and 80.1% sync-loss errors via forward resynchronization on average. As a result, the car system's failure rate was accordingly reduced to 11.7% and 28.8%, respectively, on average. We can observe from Fig.6 a great reduction in the failure rates against their original values without using ReSync.

We observed that failures may still exist even if ReSync resynchronized a car system with its environment successfully. There are two reasons. First, we have relied much on the underlying hardware. In fact, inaccuracy always exists in measuring sensory data and controlling physical devices. Even if a sync-loss error is fixed by backward resynchronization, a robot car may not be able to precisely go back to its original location before the current adaptation. Such error would be accumulated and eventually affect the normal functioning of the car. Second, a car's adaptation logic may be too weak to handle environmental dynamics. Undoing and then redoing the current adaptation may not always work. Even for forward resynchronization, it tries to ignore the current error and aims to recover a system's adaptability for future obstacles. However, the system may itself be too weak to cope with normal situations, for which our ReSync cannot improve.

D. Threats to Validity

We briefly analyze potential threats to the validity of our experimental results. First, our selection of the ten test scenarios may include unknown bias. However, we note that although these scenarios may not cover all possible environmental dynamics, they do cover both normal functioning situations and those special situations where sync-loss errors occur. In addition, these sync-loss situations have been really

encountered in our project development, and therefore they are trustable. Second, we only used our two robot cars for evaluation. However, we note that the sync-loss problem concerns with model-based self-adaptive systems that build their adaptation logics on top of states and transition rules. Such a model is generic and its encountered sync-loss errors are also common to other self-adaptive systems. In addition, we used 20 different system versions and two hardware plat-forms to reduce possible bias. Indeed, validating whether our ReSync framework is applicable to more real systems is necessary and we are working along this line.

VI. Discussion and Related Work

In this section, we discuss several issues of our ReSync framework and its limitations. We then discuss and compare our ReSync to related work.

Discussions. First, we note that our resynchronization approach does not aim at fixing faulty adaptation logic originally designed for self-adaptive systems. Instead, it tolerates such faults and attempts to recover a system to its normal adaptation when such faults cause any sync-loss error. In many cases, such recovery helps a lot by allowing a system still able to correctly adapt to future environmental changes, as we observed in experiments.

Second, our resynchronization approach has a practical impact on developing dependable self-adaptive systems. Due to dynamic nature of physical environments, developers can hardly consider all situations adequately at design time. Our work releases developers from such burden by allowing a self-adaptive system enhanced by our ReSync framework to cope with unconsidered situations where inadequate design leads to sync-loss errors.

Third, in our resynchronization approach, stable states are used as target states to which our backward and forward resynchronization proceeds. This is based on the idea that a self-adaptive system is supposed to have a hierarchy of its adaptation logic, in which several layers take care of different levels of adaptation processes. In this work, we consider two levels, namely, stable state and transient state. In fact, the idea can be further extended to multi-level adaptation. When a sync-loss error occurs at one level, one can try to recover the system with its upper level as the target.

Fourth, when forward resynchronization is applied, a system goes along an adaptation path to its future stable state where there is no sync-loss error. Then, whether the actions on this adaptation path should be conducted is an open issue. In this work, we choose to conduct them. However, this may not be always suitable for other systems. Therefore, we suggest that these actions should be assigned with an additional attribute like “skippable” or “unskippable” to allow user customization. Such extension can better suit specific needs for different systems.

Limitations. Our work has some limitations. First, we note that the effectiveness of our ReSync framework may be subject to how guard conditions of states are specified. Too strong guard conditions may cause over-reporting of sync-loss errors that are not supposed to occur (i.e., false positives). Too weak guard conditions may miss reporting situations where sync-loss errors should have occurred (i.e., false negatives). In our experiments, we designed guard conditions on behalf of users based on their provided descriptions, and obtained confirmations from them. In practice, users need to specify guard conditions themselves. Whether users can specify useful guard conditions and whether this would affect the effectiveness of our ReSync framework needs further investigation. We are now conducting a study on this issue and further findings would be available in future.

Second, although our ReSync framework and its resynchronization methodology are general to model-based self-adaptive systems, we have evaluated ReSync only on our robot cars. How effective ReSync is to other self-adaptive systems still needs investigation. We have tested ReSync with 20 different software versions and two hardware plat-forms, and the results are promising. We expect to conduct more real experiments on other self-adaptive systems to further validate the usefulness of our ReSync framework.

Related works. Defining and detecting sync-loss errors and recover a system from them by resynchronizing it with its environment is one major contribution of this work. Although the notion of guard condition is not new, building the concept of sync-loss error based on guard conditions and using it for automated synchronization between a self-adaptive system with its environment is novel. In fact, sync-loss error can also be defined upon the inconsistency between a system and its environment. This allows a further relaxed recovery model, while inconsistency issues [7][11] have been extensively studied in our communities and many pieces of work support extensions to our work.

Various techniques [3], [4], [5], [6], [16], [17], [18] have been proposed for ensuring the dependability of software systems. They typically do not consider the synchronization between a system and its environment, or simply assume that such synchronization trivially holds. When such a system interacts with its environment and the synchronization between them is lost, those verified properties or well-tested units may work no longer as expected.

Some pieces of work are dedicated for self-adaptive or context-aware systems. For example, Sama et al. [3] [9] used three families of static verification techniques to detect faults in a system’s adaptation model. Instead, our work detects errors and recovers a system from such errors dynamically. Lu et al. [17] and Wang et al. [18] explored hidden data flows caused by context-awareness and proposed covering such data flows in software testing to better detect faults in applications. Their techniques work at a code level, not directly applicable to our problem. Besides, they focus on exposing problems, while our work fixes problems. Our previous work [7] tried to detect errors dynamically for self-adaptive systems and log error information for debugging. That work highlights extra challenges existing in developing such software systems. This work naturally follows our previous work by recovering such systems when these errors occur at runtime.

Our backward and forward resynchronization ideas have been inspired by existing work from service and workflow communities [12], [13], [14], [15]. What deserves noting is that our resynchronization not only recovers an error at a software level, but also makes the recovery meaningful to physical environments. This is because we consider conducting, compensating, and retrying both digital actions and physical actions. Although our extended A-FSM model looks similar to Samas work [3], we have incorporated three new features, which facilitate detecting sync-loss errors as well as recovering the system from such errors. Our previous work [19] also extended the A-FSM approach with event patterns to avoid some types of errors, but cannot prevent other errors from happening at runtime. The latter is exactly the focus of this work. Our previous work [7] [10] [11] [20] also tried to detect and resolve context inconsistencies at runtime. This resembles this work but aims at a data level, while this work targets at an application level. Therefore, they complement to each other.

Finally, some pieces of work focus on architectural support for self-adaptive systems. This support can provide abstract global views of such systems and specify system-level integrity constraints, so that self-adaptive systems can be analyzed and managed at an architectural level [21], [22], [23]. Our work focuses on the adaptation process, which can contain multiple steps. Therefore, these two perspectives are complementary to each other, together contributing to the dependability of self-adaptive systems.

VII. Conclusion

Model-based self-adaptive systems are light-weight applications that perceive environmental changes and adapt smartly. In this paper, we focus on the sync-loss error of such systems and show that by fixing such errors, these systems can recover their adaptability and reduce their runtime failures significantly. Our work complements to existing work on modeling self-adaptive systems in that it enables the modeling, detecting, and fixing of sync-loss errors for such systems. The work also complements to existing work on testing self-adaptive systems in that challenging faults may be hidden in programs that are hard to disclose, and our work provides a novel way to prevent such faults from causing runtime failures that would otherwise be inevitable. Besides, our work can be easily extended to log the conditions under which these sync-loss errors have occurred, facilitating later program debugging.

Our work is still preliminary at several aspects. Some issues about guard conditions, context inconsistency, and resynchronization strategy that have been discussed in the Section VI need further investigation. Besides, we plan to extend validating our ReSync framework with more self-adaptive systems for its general applicability.

Acknowledgement

This research was partially funded by the 973 Program (2009CB320702), 863 Program (2011AA010103), National Nature Science Foundation (61100038, 61021062 and 60973044) of China, and by Research Grants Council (grant 612210) of Hong Kong. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (NCET-10-0486).

References

[1] Betty H. Cheng et al. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[2] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[3] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian Elbaum. Model-based fault detection in context-aware adaptive applications. In *Proceedings of FSE*, pages 261–271, 2008.

[4] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of ICSE*, pages 371–380, 2006.

[5] Sandeep S. Kulkarni and Karun N. Biyani. Correctness of component-based adaptation. In *Proceeding of CBSE*, pages 48–58, 2004.

[6] Li Tan. Model-based self-adaptive embedded programs with temporal logic specifications. In *Proceedings of QSIC*, pages 151 –158, 2006.

[7] Chang Xu, S. C. Cheung, Xiaoxing Ma, Chun Cao, and Jian Lu. Adam: Identifying defects in context-aware adaptation. *The Journal of Systems & Software (JSS)*, 85(12):2812–2828, 2012.

[8] https://sites.google.com/site/njuselfadaptiverobotcar.

[9] Michele Sama et. al. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 36:644–661, 2010.

[10] Chang Xu, S. C. Cheung, W. K. Chan, and Chunyang Ye. Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Softw. Eng. Methodol.*, 19:9:1–9:61, 2010.

[11] Chang Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of ESEC/FSE-13*, pages 336–345, 2005.

[12] Heiko Schuldt, Gustavo Alonso, Catriel Beeri, and Hans-Jörg Schek. Atomicity and isolation for transactional processes. *ACM Trans. Database Syst.*, 27:63–116, March 2002.

[13] C. Hagen and G. Alonso. Exception handling in workflow management systems. *Software Engineering, IEEE Transactions on*, 26(10):943 –958, 2000.

[14] Paul Greenfield, Dean Kuo, Surya Nepal, and Alan Fekete. Consistency for web services applications. In *Proceedings of VLDB*, pages 1199–1203, 2005.

[15] Chunyang Ye, S.C. Cheung, W.K. Chan, and Chang Xu. Atomicity analysis of service composition across organizations. *Software Engineering, IEEE Transactions on*, 35(1):2 –28, 2009.

[16] T. H. Tse., Stephen S Yau, W. K. Chan, Heng Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of COMPSAC*, pages 458–466, 2004.

[17] Heng Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an rfid-based experimentation. In *Proceedings of FSE*, pages 242–252, 2006.

[18] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *Proceedings of ICSE*, pages 406–415, 2007.

[19] Shuchu Gao, Jun Wei, Chang Xu, and S.C. Cheung. Sequential event pattern based design of context-aware adaptive application. *International Journal of Software and Informatics (IJSI)*, 4(4):419–436, 2010.

[20] Chang Xu, S.C. Cheung, W.K. Chan, and Chunyang Ye. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In *Proceeding of ICDCS*, pages 713 –721, 2008.

[21] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proceeding of FOSE*, pages 259–268, 2007.

[22] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Proceeding of ICSE Companion*, pages 899–910, 2008.

[23] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46 – 54, 2004.