Contemporary Engineering Sciences, Vol. 7, 2014, no. 13, 617 - 628
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2014.4668

GRAS: Non-Programming Approach to Generate an Adaptation Strategy in Runtime

Donghyeon Kim

Department of Computer Science & Engineering
Woojung College of Information and Communications, Korea University
5-ka, Anam-dong, Sungbuk-ku, Seoul, Korea

Hoh Peter In

Department of Computer Science & Engineering
Woojung College of Information and Communications, Korea University
5-ka, Anam-dong, Sungbuk-ku, Seoul, Korea

Abstract

Increasingly complex and rapidly changing software requirements have advanced the principles of adaptive software development. However, defining every potential problem scenario during the design phase that will require the system to adapt is difficult and time consuming. According to current research, defining new adaptation strategies for problems undefined during the design phase frequently requires engineers to modify the system's programs, and occasionally to modify programs in the target system. This can include scenarios when the problem is not especially critical to the system. To avoid this situation in the future, we suggest a novel approach that enables Human Actors to create an adaptation strategy through our framework, GRAS (Generator for Runtime Adaptation Strategy). GRAS, which does not require programming, provides an interface and knowledge base enabling Human Actors to participate in the adaptation process and define new strategies. Each problem scenario the system encounters is defined as a Situation to resolve ambiguity, and managed by the applicable Situation Handler, which can be created in runtime with the aid of a Human Actor. With GRAS, a Human Actor with the proper domain knowledge for a system can define additional adaptation strategies, and is not required to be a skilled programmer.

Keywords: Software engineering, Adaptive system, Situation-based, Human interface

1 Introduction

As modern software systems become increasingly complex, the costs incurred to reconfigure, update and maintain the systems increase as well. As a solution for this issue, the concept of adaptive software is suggested [1]. An adaptive software system adjusts various artifacts or attributes in response to changes in the software itself, and also adjusts to changes in the system's context at runtime to accommodate requirement changes. In recent years, considerable research have been performed to study adaptability [2], and various methodologies and frameworks have been suggested to enhance adaptive software development efforts [2][3][4][5][6][8][9][10][11][12].

Although a considerable amount of adaptive software research has been completed, building an effective and efficient adaptive system is an arduous task, because it is difficult to predict and define every problem that the system may be required to handle. For the problems omitted in the design phase, new adaptation strategies must be defined. However, it can be difficult to add functionality to software which has already been through the design, coding and testing processes. Current research efforts attempt to overcome this problem by referencing script code which defines adaptation strategies, or reconfiguring and linking the system to external adaptation engines. However, these approaches require a certain level of additional coding, which must be completed by programmers. Furthermore, current research suggests that implementing adaptation strategies to address previously undefined problems will require the system to be shut down and subsequently restarted, after the engineers complete the strategy's implementation. This includes scenarios where the previously undefined problem is not sufficiently critical to force the system to abort. This process may require expensive wait time to terminate and restart the running system; in addition, it requires engineers to acquire domain knowledge for the system and learn the programming language for the script code. As a result, system engineers are required to be both domain experts and skilled programmers to successfully navigate this process.

To solve this problem, we suggest a new software engineering methodology for implementing new adaptation strategies into a system, while minimizing the effort required from system engineers. The system automatically executes the adaptation strategy for situations which are defined at design phase. Human Actors define new adaptation strategies through the interface of the proposed framework, for situations which could not have been anticipated by developers during the design phase. We suggest a process that can implement a system adaptation as a "Situation," based on a set of attributes from the system environment. Subsequently, the adaptation is executed by 'Situation handlers' which are implemented as a class. If the system encounters a new, unknown situation, it utilizes a Situation Definer to notify a Human Actor of the current situation. The notification includes summarized information, and provides a list of operations which has been refined to assist the decision making process of the Human Actor. Specifically, a Human Actor can apply the summarized information from the notification to define a new strategy through the Controller interface, without actual programming. Applying this model, a Human Actor is not required to be a skilled programmer; they can define the adaptation strategy mainly by utilizing their domain knowledge for the system, thus minimizing their system management burden. We defined the framework for this approach as GRAS, Generator for Adaptation Strategy.

In section 2, an overview of the GRAS framework is provided. Section 3 explains the adaptation process of GRAS utilizing a smart lecture room case study. In section 4, assumptions and the current limitations for this research are described. Section 5 introduces other works related to our research domain, and Section 6 includes the conclusion for this paper and a discussion of future research projects.

2 Overview of GRAS framework

In this section, we present an overview of the GRAS framework. In this approach, each problem scenario to which the system should adapt is defined as a Situation. The system monitors data from its environment utilizing its sensor components, and obtains ***Attributes*** from the data. A ***Situation*** represents a meaningful combination of data which the system should recognize and manage. Each Situation is composed of Attributes which are monitored by sensor units. Situations are divided into Known Situations and Unknown Situations, based on the system's adaptation capability. A ***Known Situation*** is a Situation that is identified and defined during the design phase, and provides the system with an adaptation strategy. Accordingly, the system can handle a Known Situation automatically during runtime. However, a system may encounter problem scenarios which are not defined as Known Situations. An ***Unknown Situation*** is a Situation omitted by the developers during the design phase; the system will not have a proper strategy for adaptation to the Situation.

To handle each Situation, GRAS utilizes a Situation Manager and Situation Handlers. A ***Situation Handler*** is a unit class that handles a single Situation detected by sensor units. Each Situation Handler contains a Situation name, Trigger operation, and Response operation. ***Situation name*** is an attribute which describes the name of the Situation to be handled. The ***Trigger operation*** checks a set of monitored Attributes utilizing defined conditional expressions. If all Attributes are true, it returns TRUE to the Situation Manager, indicating that the Situation Handler is the proper one for the encountered Situation. Otherwise, it returns FALSE. The ***Response operation*** executes the defined combination of functional unit operations. It implements the system's actual response behavior to handle the Situation.

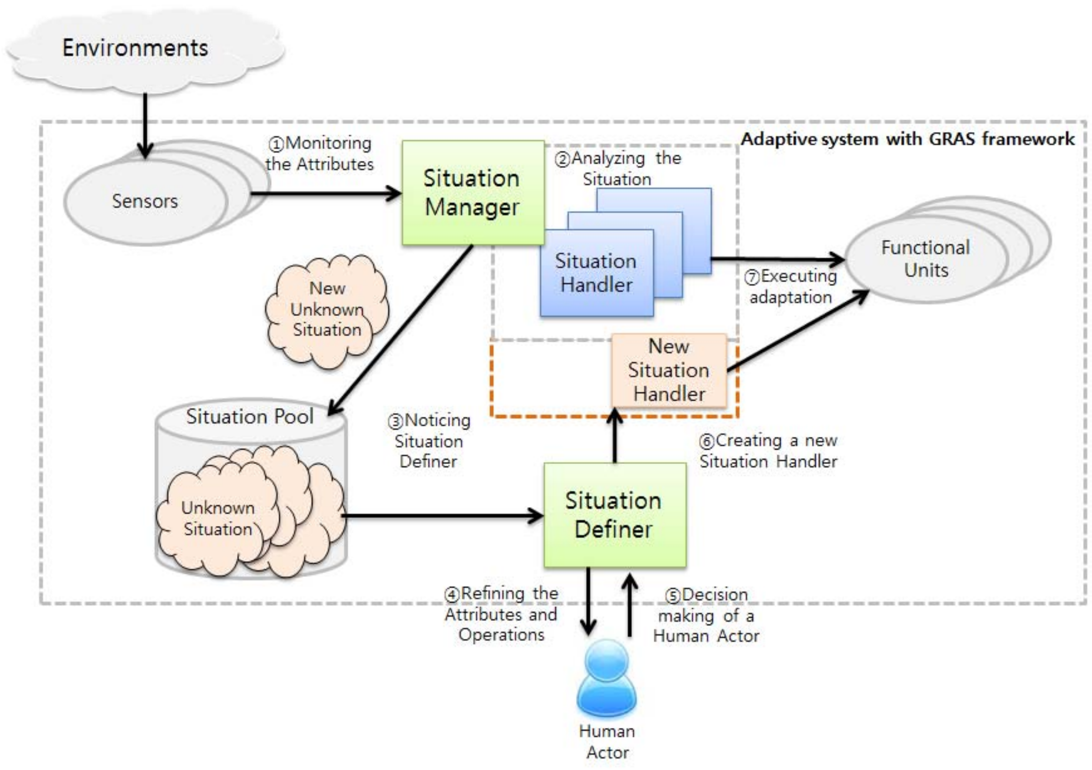


Figure 1. Overview of GRAS Framework

Every Situation Handler is registered to its manager class, Situation Manager. ***Situation Manager*** is a system component which stores Situation Handlers as a list, and manages the adaptation process of the system. It can add a newly defined Situation Handler to its list. When an Attribute change is detected, it executes a ***Validation loop*** which subsequently executes the Trigger operation of every registered Situation Handler utilizing the current set of Attributes, to locate the proper Situation Handler.

Unknown Situations represent any problem scenario that cannot be handled by currently registered Situation Handlers. To handle an Unknown Situation, Human Actors must define a new adaptation strategy utilizing the Situation Definer, which creates a new Situation Handler. A Situation Handler can be created during runtime, thus providing the system with the capability to adopt a new adaptation strategy while processing. Every unhandled Unknown Situation is temporarily stored in the ***Situation Pool***, and then transmitted to the Situation Definer. The ***Situation Definer*** is the main component enabling Human Actors to define a new adaptation strategy. The Situation Definer stores a database for refining raw Attributes and operations in the ***Attribute Interpreter*** and ***Operation Interpreter***. The ***Controller*** interface provides the Interpreted data to a Human Actor, who analyzes the Interpreted data and enters a new strategy into the interface. The ***Handler Creator*** defines a Trigger Operation and a Response Operation utilizing the input provided by the Human Actor and creates a new Situation Handler..

3 Application of GRAS utilizing an example scenario

The GRAS adaptation process is divided into seven steps as described in Figure 1. In this section, we explain the GRAS process in detail, illustrated with example scenarios. Section 3.1 describes the abstract model of the smart lecture room, with two Situation scenarios that depict the GRAS process. Utilizing those examples, in Section 3.2 we explain each step of the adaptation and strategy creation process, to address problems that were not defined during the design phase.

3.1. Example model: Smart lecture room

To explain the adaptation process of GRAS, we modeled a smart lecture room system. Our smart lecture room manages the lecture room's illumination, to attain the most comfortable lighting conditions for the room. By monitoring the status of the room's computer, projector and light sensor, the system determines the lecture is in progress but the room is too bright to see the projector screen clearly. Accordingly, the system utilizes the window blinds and both front and rear lights to change the illumination. In another example, suppose the light sensor is inoperable and returns an abnormal value of -1 to the system. The system detects a change in an Attribute, but it cannot determine how to handle this Situation. In the next section, we explain the GRAS process utilizing these two scenarios.

3.2. Adaptation process of GRAS

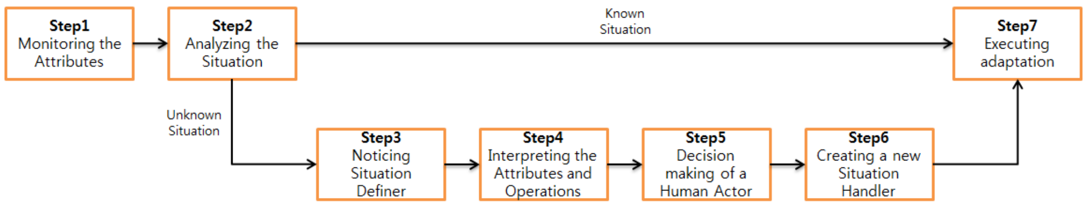

Step1
Monitoring the Attributes
Step2
Analyzing the Situation
Known Situation
Step7
Executing adaptation
Unknown Situation
Step3
Noticing Situation Definer
Step4
Interpreting the Attributes and Operations
Step5
Decision making of a Human Actor
Step6
Creating a new Situation Handler

Figure 2. Process view of GRAS

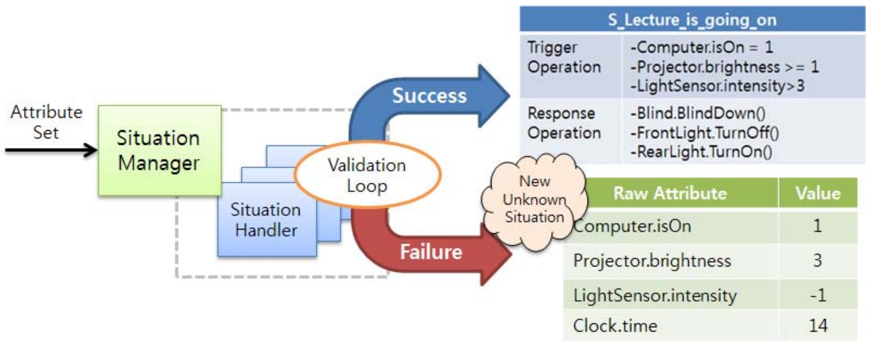
In this subsection, each step of the GRAS adaptation process described in Figure 1 is explained in detail. Figure 2 displays the execution flow of the process. Step 1 and Step 2 are common monitoring and analyzing processes for both Known Situations and Unknown Situations. Based on the analysis results, if the newly encountered Situation is a Known Situation, the process proceeds directly to Step 7 and the adaptation is executed. However, for an Unknown Situation, the system proceeds with Steps 3 through 7 to adopt a new adaptation strategy during runtime.

Step 1. Detecting the Situation

When the sensors detect a change in the Attributes they monitor, the system combines current Attributes into a structure instance. This Attribute set is transmitted to the Situation Manager, which is invoked to analyze it.

Step 2. Analyzing the Situation

After receiving the Attribute set, the Situation Manager executes the Validation loop for each registered Situation Handler. If one of the registered Situation Handlers returns TRUE from its Trigger operation, this indicates a matching Handler exists for the Situation, and the Validation loop returns that Situation Handler. The encountered Situation is defined as a Known Situation, thus the system jumps from Step 2 to Step 7 to execute an adaptation utilizing the matching Handler.



S_Lecture_is_going_on	
Trigger Operation	-Computer.isOn = 1 -Projector.brightness >= 1 -LightSensor.intensity>3
Response Operation	-Blind.BlindDown() -FrontLight.TurnOff() -RearLight.TurnOn()

Raw Attribute	Value
Computer.isOn	1
Projector.brightness	3
LightSensor.intensity	-1
Clock.time	14

Figure 3. Analyzing the Situation

However, if none of the registered Situation Handlers return TRUE, the Validation loop fails, and the system defines the Situation as an Unknown Situation. Because we have narrowed the scope of Unknown Situations to represent non-critical problems which do not cause a system shutdown, the system maintains the status by utilizing a system status snapshot, which includes Attributes and the previously running Situation Handler. This status is maintained until the new Situation Handler is defined, or the system detects another Situation. Accordingly, the system proceeds with Steps 3 through 6 to define a new Situation Handler. In the first example, the S_Lecture_is_going_on Situation Handler returns TRUE for the Situation, and is subsequently handled by its Response operation. In the second example, however, there is no Situation Handler having a Trigger operation that returns TRUE for Lightsensor.intensity = -1; therefore, the Situation is considered to be an Unknown Situation. Figure 3 displays the example process.

Step 3. Noticing Situation Definer

The system stores the Attribute set in the Situation Pool and notifies the Situation Definer that a new Unknown Situation has occurred. If multiple Unknown Situations are currently in the Situation Pool, it applies the following algorithm to determine priority for the Unknown Situations.

-The Situation that occurred the most recently – this is the default and currently used

-The Situation that waited the longest time

-The Situation which has the lowest similarity to Known Situations.

-The Situation which has the highest similarity to Known Situations.

Step 4. Refining the Attributes and operations

The Situation Definer receives the highest priority Unknown Situation from the Situation Pool, and refines the provided Attributes and operations into a human readable form with a description. This refining process is performed by the Attribute Refiner and Operation Refiner components in the Situation Definer. The knowledge base for the refining process should be defined during the design phase's requirements engineering process and provided to the components. Figure 4 describes the Attribute refining process based on the example scenario.

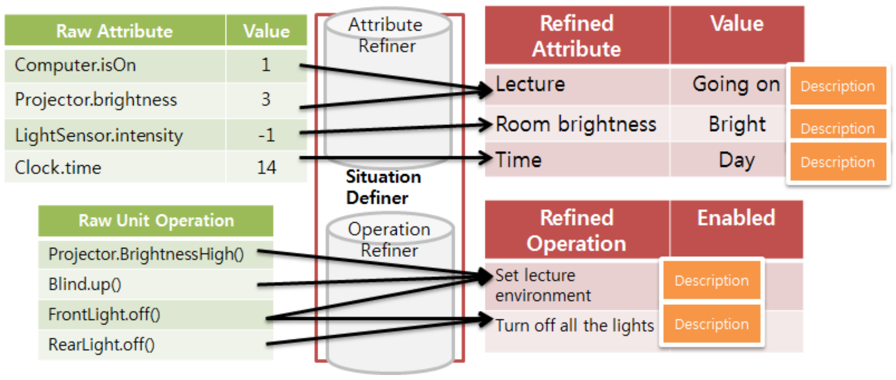


Figure 4. Refining the Attributes and operations

Step 5. Decision making by a Human Actor

The Situation Definer notifies a Human Actor regarding the Unknown Situation through the Controller interface and provides the Refined Attributes; in addition, it provides a list of refined operations to support strategy definition. A Human Actor can access both the refined interface for ease of understanding, and the raw interface for the detailed solution. The Human Actor determines the range or the value of the Attribute set to define a Situation, and selects available operations from a list to create a strategy for the new Situation. In the example scenario, the Human Actor considers the intensity of the Light sensor and the time from the Clock to define a new adaptation strategy. If the Light sensor is not working properly, and the current time of day is during daylight hours, the system should turn off all the lights and draw up the window blinds.

Step 6. Creating a new Situation Handler

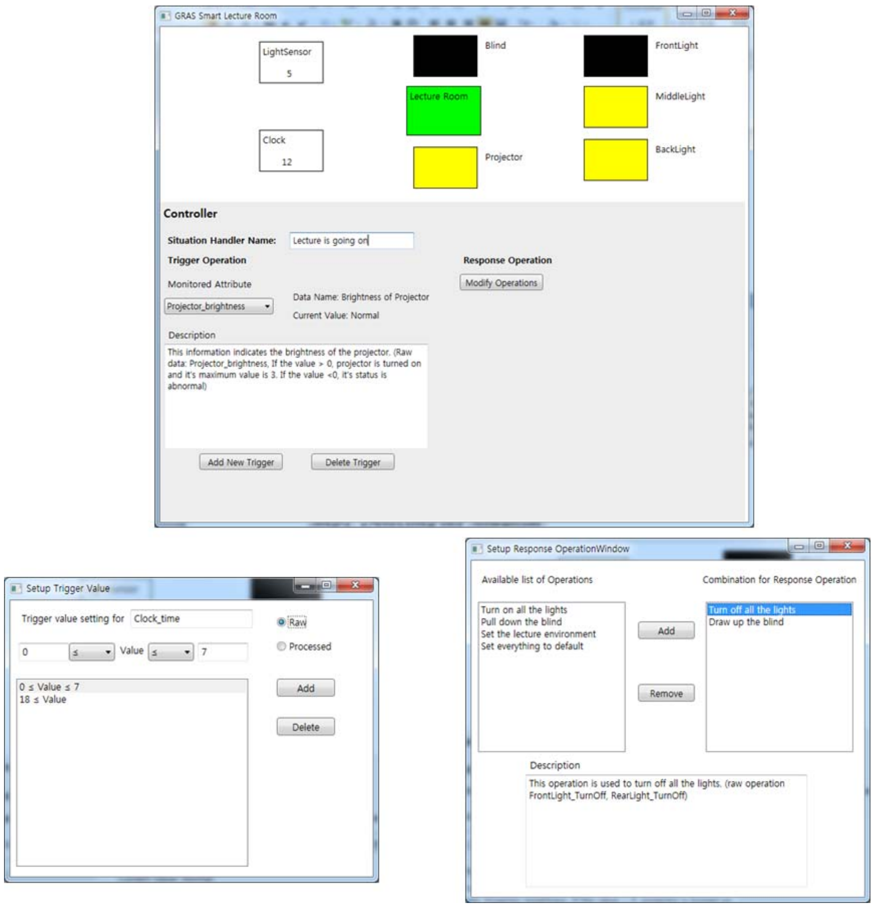
Employing the input received from the Controller, the Situation Definer creates a new Situation Handler utilizing the Handler Creator component. It defines the Situation Handler's Trigger operation utilizing the Attribute set, and defines the

Response operation from the combination of input operations. The Situation Definer registers the new Situation Handler with the Situation Manager. The Situation Manager determines if there is conflict between the new Handler and existing Handlers by checking for overlap in Trigger operation conditions; if no conflict is identified, it executes the Validation loop with the newly registered Situation Handler.

Step 7. Executing adaptation

The Situation Manager executes the Response operation of the Situation Handler returned by Validation loop from the previous Step (either Step 2 or Step 6). Operations of the functional units are executed in accordance to the Response operation, and the adaptation is completed. Utilizing the newly defined Situation Handler, the system can immediately execute an adaptation when it detects a comparable Situation in the future.

We have implemented the smart lecture room example utilizing a GRAS prototype. Figure 5 displays screenshots from the implementation. The prototype of the GRAS framework is currently implemented only in C#; however, except for the user interfaces (UI), the basic structure of GRAS can be implemented or refined in any Object-Oriented Programming (OOP) language.



LightSensor 5

Clock 12

Situation Handler Name: Lecture is going on

Monitored Attribute: Projector_brightness

Data Name: Brightness of Projector

Current Value: Normal

Description

This information indicates the brightness of the projector. (Raw data: Projector_brightness, If the value > 0, projector is turned on and it's maximum value is 3. If the value <0, it's status is abnormal)

Trigger value setting for Clock_time [✓ Raw] Processed

0 ≤ Value ≤ 7

0 ≤ Value ≤ 7
18 ≤ Value

Available list of Operations

Turn on all the lights
Pull down the blind
Set the lecture environment
Set everything to default

Combination for Response Operation

Turn off all the lights
Draw up the blind

Description

This operation is used to turn off all the lights. (raw operation FrontLight_TurnOff, RearLight_TurnOff)

Figure 5. Implementation utilizing a GRAS prototype

4 Assumptions and limitations

To apply GRAS's human-aided adaptation process, there are some conditions that should be met by both system designers and the human resources managing the system. In designing the system, the complexity of operations utilized by functional units should be minimized, because these operations must be refined and sent to the Situation Definer component, to enable Human Actors to define new adaption strategies. The Situation Definer's Controller interface, which allows Human Actors to access both the monitoring data and the available operation list, should be straightforward and easy to understand for the Human Actors. The Knowledge base utilized for refining the attributes and operations in the Situation Definer should be well defined in the requirement engineering process, to support the decision making of a Human Actor. A Human Actor who manages the system should be sufficiently educated about the system and its requirements. This system expertise will be required to understand the Controller interface and analyze the Unknown Situations encountered by the system.

Although all of these conditions may be met, there will be some limitations caused by human involvement. Human-aided processes require skilled engineers that possess system domain knowledge; therefore, costs will be incurred to educate and train these Human Actors. However, this requirement will reduce the time and funds available for training Human Actors to program scripts for implementing new strategies. Moreover, there is no guarantee that a Human Actor will always make the best decisions possible to manage the Unknown Situation. For this research, we operate under the assumption that the Human Actor who "understands the system well enough" through education can make "proper decisions" for the Situations encountered. Finally, if the monitoring data or available combination of operations is too complex, Human Actors, including those with the greatest amount of knowledge, may not be able to make a proper decision for an Unknown Situation. Excessive complexity may also result in reduced GRAS execution speed. To overcome these problems, the knowledge base in the Situation Definer should be well-defined, and Controller interface should be straightforward and well-organized.

5 Related works

Rainbow [2][3][4] is an architecture-based self-adaptive framework suggested in 2008; it remains widely researched by numerous adaptive software research groups. Rainbow utilizes probes and gauges to monitor target systems and utility functions to locate suitable adaptation strategies in its processes. It uses ACME to model the adaptive system architecture, and the Stitch language to script the adaptation strategy. MUSIC [5] is suggested for developing self-adaptive software in the mobile domain. It provides middleware support to configure target systems and utilizes modified UML for modeling. However, it has not been updated since

2010, and is currently not functioning. DiVA [12] is a requirement-based adaptation framework suggested in 2010. DiVA utilizes requirements documents to extract functional and non-functional requirement from the target system, and creates its own DiVA metamodel to construct adaptive systems. It provides DiVA Studio as an Eclipse plugin, but it has not been updated since 2011 and currently not functioning because its core component has been modified. Similar to DiVA, Souza et al. presented Zanshin [8][9][10][11] as a requirement-based adaptive framework in 2012. In Zanshin, initial requirements are the core of the adaptation process, and remain unchanged. They define two special requirements: the Awareness requirement and the Evolution requirement. The Awareness requirement is designed to monitor the process, and the Evolution requirement serves as an adaptation strategy in the process.

All the research mentioned above can execute adaptations only for defined strategies in defined processes. To manage undefined problems, they require additional programming to implement new adaptation strategies; for example, the Stitch language in Rainbow and the Evolution Requirement in Zanshin require engineers to perform basic programming tasks. Furthermore, to adopt newly defined strategies, the running system has to be shut down and restarted. In contrast, GRAS can execute the adaptation process utilizing the Situation handling routine for Known Situations; for undefined problems, Human Actors can add new strategies easily utilizing the Situation Definer without shutting down the system. The Controller interface provides a Human Actor with monitoring attributes and available operations utilizing a GUI interface; therefore, a Human Actor can successfully manage Undefined Situations by applying their domain knowledge about the system, and does not need to be a skilled programmer.

6 Conclusions

Utilizing GRAS, we suggested a novel approach for implementing new adaptation strategies without requiring programming or shutting down the system. First, we defined the concept of Situations to describe the scenarios to which the system should adapt. We defined the adaptation process, and described how Situations are handled by a Situation Handler registered with the Situation Manager. By clearly defining problem scenarios and the system reaction process with the Situation concept, our approach minimizes ambiguity in the adaptation process. To handle an Unknown Situation, which is a problem undefined during the design phase, a Human Actor participates in the adaptation process to define a new strategy utilizing the Controller interface. The system provides the attributes and operations refined by the Situation Definer, employing the knowledge defined in the requirement engineering process, to support the decision making of the Human Actor. Compared to other adaptive systems, a Human Actor can define adaptation strategy straightforwardly utilizing the Controller from the Situation Definer. After the adaptation strategy is defined, the system can apply the new adaptation strategies to manage the undefined problem. Because the process for creating a new Situation Handler can be completed without programming, it is not necessary for the Human Actor to be an expert programmer, and mainly requires domain knowledge of the system.

We are currently implementing the complete version of the GRAS adaptation framework. In future work, we will verify the actual usefulness of our methodology through additional research. Furthermore, we will attempt to minimize the effort required of Human Actors and manage their limitations by devising a more effective interface for the Situation Definer.

Acknowledgements. This research was supported by the Next-Generation Information Computing Development and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planing(2012M3C4A7033345) and by the Ministry of Education, Science and Technology(2012R1A1A2009021), and by the R&BD Support Center of the Seoul Development Institute and the South Korean government (WR080951).

References

[1] B.H. Cheng, et al, Software engineering for self-adaptive systems: A research roadmap, Software engineering for self-adaptive systems, Springer Berlin Heidelberg (2009), 1-26.

[2] S.W. Cheng, Rainbow: cost-effective software architecture-based self-adaptation, ProQuest, 2008.

[3] S.W. Cheng, and D. Garlan, Stitch: A language for architecture-based self-adaptation, Journal of Systems and Software, 85 (2012), 2860-2875.

[4] D. Garlan, S.W. Cheng, A.C. Hauang, et al, Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, In IEEE Computer, 37(2004), 46-54.

[5] S. Hallsteinsen, et al, A development framework and methodology for self-adapting applications in ubiquitous computing environments, Journal of Systems and Software, 85 (2012), 2840-2859.

[6] R. Laddaga, P. Robertson, and H.E. Shrobe, Self-adaptive software, Proposer Information Pamphlet BAA (1997), 98-12.

[7] M. Salehie and L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 4 (2009): 14:1-42.

[8] V.E.S. Souza, and J. Mylopoulos, Requirements-based software system adaptation, Diss. PhD Thesis, University of Trento, Italy (2012).

[9] V.E.S. Souza, A. Lapouchnian, and J. Mylopoulos, (Requirement) evolution requirements for adaptive systems, Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on. IEEE (2012), 155-164.

[10] V.E.S. Souza, A. Lapouchnian, and J. Mylopoulos, Requirements-driven qualitative adaptation, On the Move to Meaningful Internet Systems: OTM 2012. Springer Berlin Heidelberg (2012), 342-361.
[11] L. Tahvildari, and K. Kontogiannis, Requirements driven software evolution, Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on. IEEE (2004), 258-259.
[12] URL: https://sites.google.com/site/divawebsite/home.

Received: May 1, 2014