An Externalized Infrastructure for Self-Healing Systems

David S. Wile and Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina del Rey, CA 90292, USA
{dwile,aegyed}@teknowledge.com

Abstract

Software architecture descriptions can play a wide variety of roles in the software lifecycle, from requirements specification, to logical design, to implementation architectures. In addition, execution architectures can be used both to constrain and enhance the functionality of running systems, e.g. security architectures and debugging architectures. Along with others from DARPA's DASADA program we proposed an execution infrastructure for so-called self-healing, self-adaptive systems – systems that maintain a particular level of healthiness or quality of service (QoS). This ***externalized infrastructure*** *does not entail any modification of the target system – whose health is to be maintained. It is driven by a reflective model of the target system's operation to determine what aspects can be changed to effect repair. Herein we present that infrastructure along with an example implemented in accord with it.*[1]

1. Introduction

While the use of architectural models in the requirements specification to implementation phases is becoming more common, research into *dynamic* software architecture models is beginning to extend the utility of software architecture specification into the execution phase of the software lifecycle. Three approaches dominate their use here. First, together with others [7,9], we advocate the definition of a covering architecture that encompasses the entire possible locus of the running system's architectural elements [13].

Another approach is to adhere to a specific architectural style (e.g. C2 [11]) that facilitates the dynamic incorporation of new components while guaranteeing interaction constraints remain invariant.

The third approach, examined here, involves the use of a specific architectural "harness" to facilitate dynamic system reconfiguration, as required for so-called "self-healing" or "self-adaptive" systems – systems that maintain a particular level of healthiness or quality of service (QoS). Figure 1 illustrates an architectural diagram of an *externalized infrastructure* that can be used to monitor, interpret, analyze, and reconfigure running systems. Essentially a layer of probes is installed into a system just before it starts to run, probes that report significant data on a probe event-bus. Gauges translate and interpret this data with respect to models that abstract from the implementation, often using an architectural model of the target system to locate logical events. The information from these gauges is transmitted onto the gauge bus where other gauges can react and control decisions can be made. A layer of "effectors" is then invoked to effect changes in the target system, either by adapting existing components – perhaps by tweaking parameters – or by reconfiguring the system itself.

This infrastructure was proposed by several members of DARPA's DASADA community [7,12,14] where it was only partially developed before funding was dropped; however, the probe and gauge interaction protocols were standardized [3,8] based on Siena event-broadcasting middleware [4].

2. Architectural Modeling

The *Target Architectural Model* (modeling the "cloud" in Figure 1) plays a key role in the infrastructure. The architectural model essentially establishes the structural vocabulary; each layer can rely on the model for understanding its role in the system and the information it is responsible for. To see how events in the physical architecture map to events in the logical, target

[1] This work was sponsored by DARPA contract number F30602-00-C-0200.

architecture model consider the following scenarios, both ensuing after probes and gauges have been placed by the control layer:

2.1. Static Architecture Scenario

- Probes emit implementation-level events like “process D006 opened file ‘C:\Program Files\log.txt’ for write” or “process E001 used 2021.”
- Gauges provide interpretations of these events by first determining what logical architectural entities are being referred to – here, perhaps a logical application, WinZIP (D006), and another logical application, MS PowerPoint (E001), for example. This mapping from implementation terms (process ids) to logical architectural components must be established in the architectural model by the processes that originally set up the system and probes. The gauges additionally interpret implicit information from the probes; for example, perhaps 2021 means port 2021.
- The gauges are then “read” by the control layer to see if any action should be taken. For example, assume that the ILE for E001 is interpreted as “MS PowerPoint (E001) is attempting to access port 2021.” Furthermore, assume that the control layer has knowledge of good, suspicious, and bad events. For example, it is known that “access to ports 1000-3000 is suspicious,” e.g., because normal application operation does not require such access. In such a case, the control layer may decide to ask the user of the application to authorize or deny access to port 2021. The control layer may then communicate the user response to authorize or deny the access to the effector layer through an adaptation event (AE).
- Then the effector layer will use the architectural model to determine that process E001 needs to be adapted – requiring the inverse translation from before, now from logical architecture to physical architecture – and determine what implementation-level response corresponds to authorize or deny events, e.g., raise an implementation-level “port access failed exception” in the latter case.

Notice that nothing about the architecture itself changed during this scenario; no modules or connections were created or destroyed. Moreover, the repair was effected by a simple parameter change to a running module; no new resources were brought to bear. A similar scenario might require dynamic target architecture changes:

2.2. Dynamic Architecture Scenario

- Probes emit architecturally significant implementation-level events, such as “process D006 spawned new process F008 of type MS Word.”
- Gauges interpret these events and modify the corresponding physical and logical architectural models. Here, perhaps, because F008 was spawned by D006, the system knows that there must be a new logical application, MS Word (F008) now and that the logical application WinZIP (D006) is the creator (parent) of MS Word (F008). We call this process *identification* of physical models with pre-defined, logical architecture models. That is, with dynamic architectures, the whole range of possible architectures is pre-specified in a covering architecture [10,13]; those elements of the architecture that have been identified with the physical architecture are kept track of. Hence, at any given time, only the identified modules and connectors constitute the actual logical architecture.
- Imagine that some time later an event similar to the one above, “MS Word (F011) is attempting to access port 2021” is transmitted by the probes and reported by the gauges. The control layer at this point could issue a user request to authorize/deny this attempt or it could change the system’s running architecture by issuing a *reconfiguration* event to the effector layer. This time perhaps the command issued would be to “replace the MS Word process (F011) with another physical application (e.g., MS WordPad, which can read MS Word documents but is less subject to exploitation by viruses).

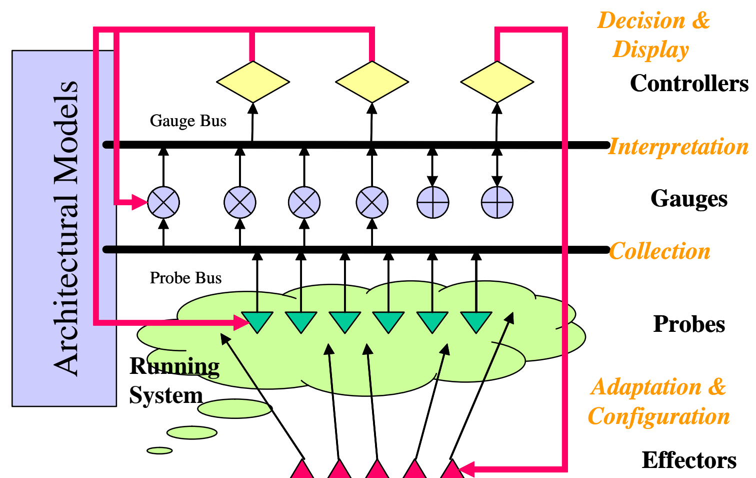

Figure 1. Infrastructure Architecture

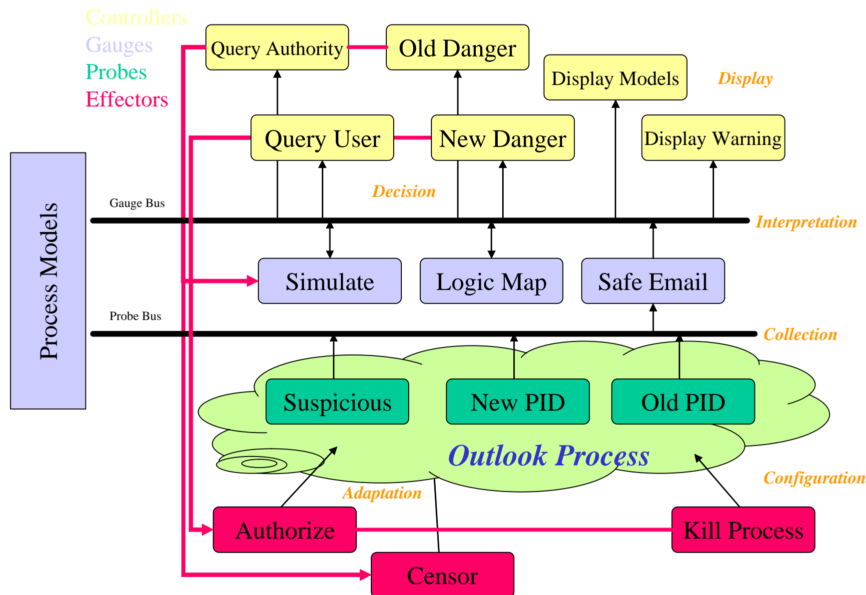

Figure 2. Idealized Safe Email Infrastructure Architecture

- The effector layer again has to map the logical MS Word component into the physical process F011 and it also has to understand how to remove that component and substitute a new one of type MS WordPad, a rather tricky activity in any event.

So there are two separable dynamic architecture activities here: modeling the dynamic architecture as it evolves and reconfiguring the architecture via the control layer.

3. An Example: Safe EMail

Although the entire externalized infrastructure architecture was never realized, we did attempt to reformulate a program that itself constitutes a self-healing harness for repairing the errant behavior of processes spawned by email attachments, our "Safe Email" program [1].

Figure 2 depicts the instantiation of the Safe Email self-healing infrastructure (Figure 1) for the email application Microsoft Outlook. Its purpose is to detect and prevent malicious behavior caused by viruses received through email. Email viruses are either embedded in the email itself to exploit security weaknesses in MS Outlook (e.g., macro viruses) or they are unleashed by attachments opened by unsuspecting users.

To counter the security threat posed by viruses, three kinds of probes are required, probes for observing: 1) the occurrence of events that could be considered "suspicious," 2) the creation of new processes (New PID), and 3) the destruction of existing processes (Old PID). All three probes are based on wrapper technology, where calls to PC Windows-based platforms' Dynamic Link Libraries are intercepted and our code is invoked before (conditionally) invoking the original code [2], reporting suspicious activity via the probe bus [3].

If a virus exploits a weakness in MS Outlook, then it will engage in suspicious activities that are observed through the first type of probe. The other probes maintain the target architectural model to coordinate faults with spawned processes. When attachments are opened, new processes are created to view/execute these attachments (e.g., MS Word or a Web Browser). Therefore, if a virus is embedded in an attachment, the new process, and not MS Outlook, will then engage in suspicious activities. Observing "suspicious" activities is thus extended to processes spawned by MS Outlook (New PID) until they are destroyed (Old PID).

The "Safe Email" gauge acts as a mediator to collect and translate probe information broadcast on the probe bus. It may combine multiple implementation-level events to produce architecture-level events that abstract from the implementation. It may also translate observed information with the help of the "Logic Map" gauge to interpret how implementation-level data relates to target architectural elements and to record the creation hierarchy of applications. MS Outlook sits at the top of this hierarchy. When it spawns a process by opening an attachment, a "child application" is created to represent the new process along with its type and id. Since a spawned process may spawn yet other processes, the target architecture model supports a tree hierarchy of "parent" applications and their "children."

Gauges cannot judge whether suspicious activities, caused by MS Outlook or any of its child processes, are truly malicious or not. If a suspicious event is observed the first time ("New Danger") then the control layer displays a warning message ("Display Warning") to let the user decide about the maliciousness of the event ("Query User"). The user may allow the activity, deny it, or kill the application. The user may also reconfigure the control layer to

ignore similar events ("Old Danger") in the future (equivalent to an automatic allow).

Probes may reside in different processes and on different machines, so the infrastructure can be used to monitor multiple email users. A so-called "Authority" is given access to a GUI showing the target architectures with processes decorated by border colors indicating how well behaved processes are with respect to producing suspicious activities that the users deny, e.g., red for malicious.

In addition, the architectural elements are simulated in the "Simulate" gauge to determine the level of trust of individual applications, based on users' responses to warnings of suspicious activities and to "guilt assessment" imposed on parents of misbehaving child processes [6]. This information is also visualized for each process.

The authority is allowed to determine that specific processes are misbehaving for enough users that subsequent attempts to invoke the suspicious actions should *automatically* be denied. "Query Authority" uses MailIDs to ensure that previously denied events are denied automatically again if they originated from the same email, e.g., even for different users.

The layer of effectors is invoked to effect changes in the running system. Effectors may "authorize" or "censor" (deny) suspicious events, or may even kill processes.

4. Conclusions and Future Work

Although we noticed problems in applying the infrastructure to the Safe Email example, we think this approach is a feasible way to decouple self-healing aspects of a target system from its functionality. Problems with externalization arise when there is a coupling between an effector that corrects a problem and a sensor that detects it, e.g., a sensor detects danger and suspends itself, awaiting a decision about how to proceed. The effector that allows it to proceed is strongly coupled, something that cannot be indicated with the infrastructure as it stands. Similar problems concern how to model the user and administrator.

Nonetheless, we feel the future of this infrastructure will best be to serve as "a template" for imposing self-healing systems on applications, as suggested by Jeff Magee [5]. In fact, we are designing an architectural style that is consistent with the infrastructure that allows refined descriptions of the relationships of the sensors, gauges, controllers, and effectors. The choice of which architecture description method to use – infrastructure or style – will depend on the volatility of the infrastructure itself.

5. References

[1] Balzer, R.: "Assuring the Safety of Opening Email Attachments," Proceedings of the DARPA DISCEX Conference , Anaheim, California, June 2001, pp.1257.

[2] Balzer, R. and Goldman, N.: "Mediating Connectors: A Non-ByPassable Process Wrapping Technology," Proceedings of the DARPA DISCEX Conference, South Carolina, January 2000, pp.361-368.

[3] Balzer, R.: "The DASADA Probe Infrastructure," Technical Report, Teknowledge Corp. (available from authors), 2003.

[4] Carzaniga A., Rosenblum D. S., and Wolf A. L.: Achieving scalability and expressiveness in an Internet-scale event notification service. ACM Transactions on Computer Systems (TOCS) 19(3), 2001, 332-383.

[5] Crane, S., Dulay, N., Fossa, H., Kramer, J., Magee, J., Sloman, M., and Twidle, K.: "Configuration Mangagement for Distributed Systems ," Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management, Santa Barbara, California, 1995.

[6] Egyed, A. and Wile, D.: "Statechart Simulator for Modeling Architectural Dynamics," Proceedings of the 2nd Working International Conference on Software Architecture (WICSA), August 2001, pp.87-96.

[7] Garlan, D. and Schmerl, B.: "Model-based Adaptation for Self-Healing Systems," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS), South Carolina, November 2002, pp.27-32.

[8] Garlan, D., Schmerl, B., and Chang, J.: "Using gauges for architecture-based monitoring and adaptation," Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 2001.

[9] Mikic-Rakic, M., Mehta, N. R., and Medvidovic, N.: "Architectural style requirements for self-healing systems," Proceedings of the First Workshop on Self-Healing Systems (WOSS), Charleston, South Carolina, November 2002.

[10] Monroe, R.: "Capturing software architecture design expertise with Armani," Technical Report CMU-CS-98-163, Carnegie Mellon University, 1998.

[11] Oreizy, P., Medvidovic, N., and Taylor, R.: "Architecture-Based Runtime Software Evolution," Proceedings of the 20th International Conference on Software Engineering (ICSE), Kyoto, Japan, 1998, pp.177-186.

[12] Valleto, G. and Kaiser, G.: "A Case Study in Software Adaptation," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), Charleston, South Carolina, November 2002, pp.73-78.

[13] Wile D. S.: Modeling Architecture Description Languages Using AML. Automated Software Engineering Journal 8(1), 2001, 63-88.

[14] Wile, D. S.: "Towards a Synthesis of Dynamic Architecture Event Languages," Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS), Charleston, South Carolina, November 2002, pp.79-84.