Using Little-JIL to Define Containment Units

Barbara Staudt Lerner
Williams College
Computer Science Department
Williamstown, MA 01267
+1-413-597-4215
lerner@cs.williams.edu

Jamieson M. Cobleigh, Leon J. Osterweil, Alexander Wise
University of Massachusetts, Amherst
Computer Science Department
Amherst, MA 01003
+1-413-545-2186
jcobleig,ljo,wise@cs.umass.edu

ABSTRACT

Self-healing systems must be able to adapt to errors and changing resource environments without human intervention. We propose an architectural style, called Containment Units, particularly intended for self-healing systems. Containment Units feature the use of operational, evaluator, and change agent modules to encapsulate different activities required in self-healing systems. We present this architectural style along with the use of Little-JIL, a visual coordination language, to describe the high-level interactions among the modules of a containment unit.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture – *domain-specific architectures, languages.*

General Terms

Design, Languages.

Keywords

Adaptive systems, Containment Units, self-healing systems.

1. INTRODUCTION

Experience has shown that software development is highly error-prone, particularly if poor software engineering practices are used. Self-healing systems are likely to be even more difficult to build well due to the variety of situations in which they will be expected to operate without human intervention. Our approach to self-healing systems focuses on describing and reasoning about adaptation at the architectural level. Thus, our work is in the same spirit as recent work on architectural adaptation, such as [3, 5, 6, 7]. Our particular goal is to explore strategies to build self-healing systems that are amenable to analysis so that developers can gain confidence in the correctness of these complex systems prior to deploying them.

Our approach builds upon two well-accepted software engineering principles: modularization and static analysis. Specifically, we propose designing self-healing systems using an architectural style that we call Containment Units and then using static analysis to demonstrate that the Containment Unit satisfies specified desirable properties.

A **Containment Unit** consists of three types of modules: operational components, evaluators, and a change agent. An **operational component** implements the functionality of the Containment Unit. A Containment Unit may have more than one operational component. These components are functionally redundant with each other, but are designed to operate in different resource environments. Collectively, they enable a Containment Unit to operate in a range of situations, such as different network connectivity, different memory availability, different CPU availability, etc. At any time at most one operational component will be in use.

An **evaluator** monitors the behavior of the active operational component to ensure that it is behaving appropriately. Additional evaluators may also monitor the resource environment to ensure that the environment is still appropriate for the use of the current operational component.

If an evaluator determines that the operational component is not behaving satisfactorily, it notifies the **change agent**. It is the responsibility of the change agent to determine how to handle the current situation, often by activating a new operational component.

The purpose of a Containment Unit is to prevent certain faults from spreading outside its control. The idea is that there are collections of faults an operational component cannot handle itself but that can be handled by replacing the active operational component with another. These faults are thus "contained" by the Containment Unit. One of the chief uses of static analysis in the context of Containment Units is to verify that those faults are indeed managed within the Containment Unit and do not spread to other parts of the system.

We believe that building a self-healing system using the modular constructs of Containment Units makes the Containment Units easier to build correctly, to understand, and to analyze. In this paper we focus on the use of Little-JIL, a coordination language, to express the architectural style of Containment Units in a manner that makes it easy to see the high-level modularization of the Containment Units. In a separate paper we discuss the Containment Unit architectural style more fully and our preliminary results in performing static analysis on Containment Units [4].
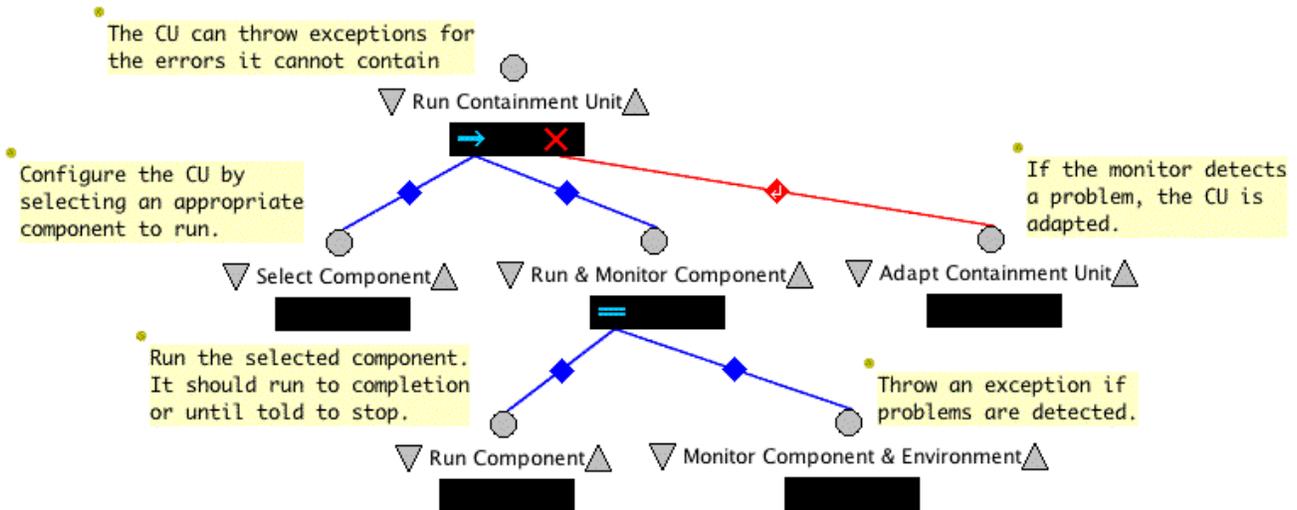
The CU can throw exceptions for the errors it cannot contain

Configure the CU by selecting an appropriate component to run.

If the monitor detects a problem, the CU is adapted.

Run Containment Unit

Select Component

Run & Monitor Component

Adapt Containment Unit

Run the selected component. It should run to completion or until told to stop.

Throw an exception if problems are detected.

Run Component

Monitor Component & Environment

Figure 1: Generic Containment Unit

2. LITTLE-JIL – A VISUAL COORDINATION LANGUAGE

Little-JIL is a visual agent coordination language. The graphical syntax of Little-JIL represents a system as a collection of hierarchically decomposed steps. The steps are the nodes in the graph while edges represent connections to substeps, exception handlers, and event handlers. An operator within the step icon represents control flow and concurrency among the steps. One can thus easily see the control flow within a system, using the hierarchical decomposition to view a system at varying levels of abstraction. Dataflow information and resource needs are visible through annotations on the graph.

The details of the Little-JIL language are provided in other papers [2, 8] and space does not permit us to repeat them here. Instead we use Figure 1, a Little-JIL specification of a generic Containment Unit, both to emphasize some of our points about the structure of a Containment Unit, and Little-JIL's suitability for representing it.

The Little-JIL step is depicted as a solid black bar with accompanying badges. Figure 1 depicts the step, "Run Containment Unit", which is decomposed into the structure of substeps shown below it. The arrow in the "Run Containment Unit" step indicates that it is a sequential step, where the substeps are performed in order from left to right. In this case, there are two substeps: "Select Component" and "Run & Monitor Component". "Select Component" initializes the Containment Unit and selects the first operational component.

"Run & Monitor Component" is a parallel step, indicated by the parallel lines in the step icon. This means that the substeps "Run Component" and "Monitor Component & Environment" are done concurrently. "Run Component" is the operational component. "Monitor Component & Environment" represents the monitors. Most likely, in a complete system, "Monitor Component & Environment" would be further decomposed into a collection of monitors executing in parallel, each with one monitoring responsibility.

For example, one might ensure that the running component is meeting its deadlines, a second that it is obeying its memory constraints, a third that it is producing results of sufficient quality, etc. If either the running component itself or the monitor detects a problem, it throws an exception. This starts the exception handler "Adapt Containment Unit" (the change agent), attached to the ✗ in the "Run Containment Unit" step. The edge connecting the exception handler to the step has a ↻ on the edge. This indicates that the "Run Containment Unit" step should restart, causing the Containment Unit to continue running after the adaptation.

Neither Containment Units nor Little-JIL steps need to form a strict tree structured hierarchy. Lower level steps (and Containment Units) may be contained as part of more than one higher level step (or Containment Unit). In addition, Little-JIL steps include (optional) prerequisites and postrequisites. Both can be used to specify where evaluation activities are to occur and, as both of these structures are full Little-JIL steps, the structures may be used to specify to arbitrary levels of detail just how the evaluation is to take place

Clearly Little-JIL is a graphical language, and our experience suggests that its iconic representations help to make the high-level control flow of systems clearer. In addition, however, we believe that additional clarity is attributable to Little-JIL's uses of higher level semantic notions as the basis for its specifications. This clarity seems to be one of the key benefits to using Little-JIL, rather than a lower level programming language for representing Containment Units. Another key feature is that Little-JIL is an executable language, making it more powerful than typical modeling languages. Little-JIL system specifications are interpreted by the Juliette process interpreter [1]. Juliette enables distributed execution of systems, managing the necessary synchronization and communication specified in Little-JIL. Juliette also supports late binding of operational components and evaluators to steps, thereby allowing for the continuous incorporation of new Containment Units without the need to modify the Little-JIL specification. In order to explain this we now introduce the Little-JIL resource management capability.
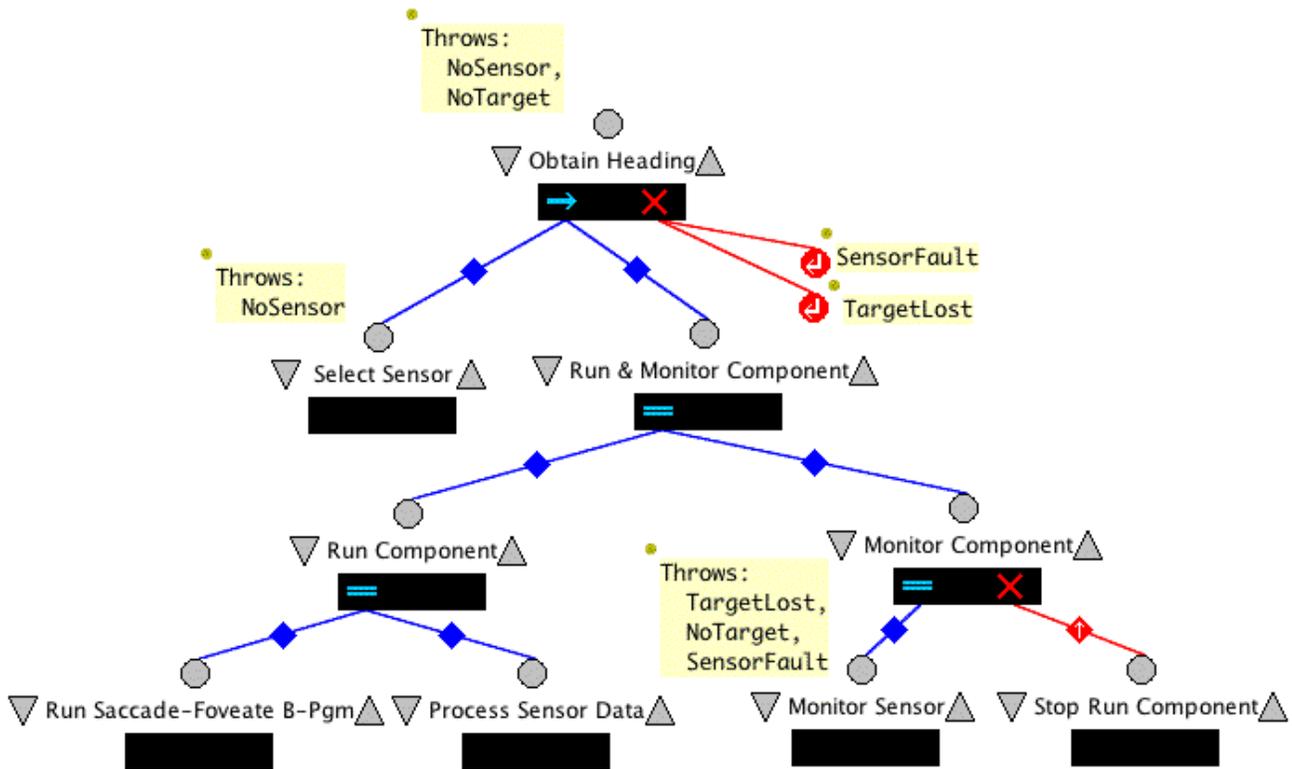
Figure 2: Obtain Heading Containment Unit

Every Little-JIL step may specify a set of resources that it requires in order to perform its assigned task. These specifications may take the form of requests for specific resources, but are more commonly requests for types of resources. At runtime, the Juliette interpreter passes these requests on to a separate resource allocation module. This resource allocation module has the responsibility for maintaining complete information about which resources are allocated to support execution of the process, and which are currently available. If requested resources are not available, the resource manager indicates this, and the requesting step fails and throws an exception that is handled as indicated previously. If a resource is available, it is bound to the requesting step and execution proceeds.

The change agent of a Containment Unit is expected to use this facility, in that the operational components available for substitution in the operational system are modeled as resources. Their entries in the resource model further indicate what other resources they depend upon. The change agent can use this information to select the most appropriate operational component given the currently available resources. It is important to note that, as the resource model is separate from the Containment Unit, it is free to evolve itself, independently of the Containment Unit. Thus, we anticipate that new operational components would be added to the resource model as they become available. The change agent will thus have a larger collection of components to select from at later adaptation points. This assures that Containment Units implemented through Little-JIL will be able to evolve in new ways, as soon as new operational modules and their evaluators are created and made available.

3. SAMPLE LITTLE-JIL CONTAINMENT UNITS

In this section we present Little-JIL descriptions of three Containment Units. The first two are designed for a robotic sensor system. In this system, a room is equipped with a collection of sensors. The sensors are placed in different parts of a room. The room itself contains obstructions, such as furniture and office partitions so that no sensor can "see" the entire room. The goal of this system is to track the motion of a person through the room, using a minimal number of sensors for the task and switching sensors when necessary. These were experiments in expressing Containment Units in Little-JIL, but were never actually executed with sensors.

The third Containment Unit is for a simpler domain, controlling a collection of lights, ensuring that an area stays lit even if one or more lights fail. While the domain is simpler, this example demonstrates that executing Containment Units written in Little-JIL is feasible.

3.1 Obtain Heading Containment Unit

Figure 2 shows the Little-JIL description of the Obtain Heading Containment Unit. This Containment Unit takes as resources a set of sensors that it will use to track the target. It begins by selecting a sensor to use in the "Select Sensor" step. Once this is done, the "Run & Monitor Component" step is executed. This step has two substeps, "Run Component" and "Monitor Component", which can be executed in parallel. The "Run Component" step uses the sensor to obtain the heading in the "Run Saccade-Foveate B-Pgm" step and reads the heading in the "Process Sensor Data" step. This latter step
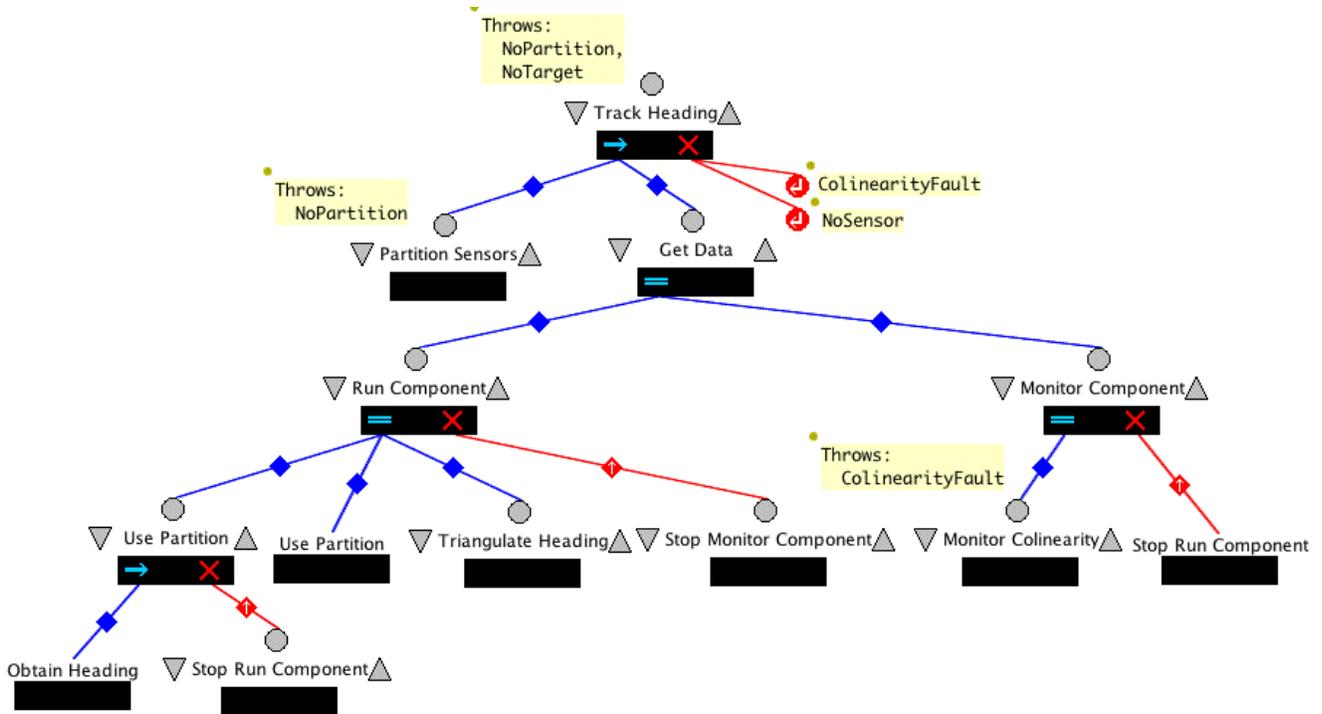
Figure 4: Track Heading Containment Unit

posts the heading read into a global data space so that the data is available for use by whomever needs it.

While these steps are executing, the "Monitor Sensor" step can execute and it also observes the sensor, but watches for the error messages that can be reported. There are three errors which are shown as the exceptions "Target Lost", "No Target", and "Sensor Fault". When an exception is thrown, it is first handled by the "Monitor Component" step, which sends a message so that the agents executing the "Run Saccade-Foveate B-Pgm" and "Process Sensor Data" steps know to stop. "Monitor Component" then rethrows the exception, which will reach the "Obtain Heading" step. If the problem was a "No Target" exception, the Containment Unit terminates, since we currently have no way to deal with this kind of fault. On a "Sensor Fault" a restart handler is encountered, causing the Containment Unit to begin again by selecting a sensor in the "Select Sensor" step. On a "Target Lost" exception, the Containment Unit knows that there is a target to be tracked, but it may be that the target is moving too fast to be tracked by the current sensor. To handle this, a restart handler is used to cause another, presumably faster, sensor to be selected. If "Select Sensor" cannot select a sensor, because none are working or a faster sensor is not available, the step throws a "No Sensor" exception, which causes the Containment Unit to terminate.

3.2 Track Heading Containment Unit

The Little-JIL description of the Track Heading Containment Unit is shown in Figure 3. This Containment Unit takes as input a set of sensors. It begins by dividing the sensors into two disjoint subsets in the "Partition Sensors" step. Each of these sensor sets will be used by an Obtain Heading Containment Unit to determine a heading towards a target. If no partitioning is possible (because there is only one sensor

available, for example), then this step throws a "No Partition" exception and the Containment Unit terminates. If a partitioning is possible, the "Get Data" step is started. This step tries to run two tasks in parallel. The first is the "Run Component" step, which invokes two instances of the Obtain Heading Containment Unit and a step to compute the Heading based on the outputs of the Obtain Heading Containment Units.

The second is the "Monitor Component" step, which uses the "Monitor Colinearity" step to detect if the target being tracked has become colinear with the two sensors and that the target's position cannot be determined. If this happens, this step throws a "Colinearity Fault". It is important to note that colinearity cannot be detected or handled in either of the Obtain Heading Containment Units, since each of these only has access to a single heading. The Track Heading Containment Unit has access to both headings and can determine when colinearity occurs. This exception gets propagated to the "Track Heading" step, which handles the exception by restarting the "Track Heading" step, causing a repartitioning of the sensors.

The Obtain Heading Containment Units may terminate with one of two exceptions: "No Sensor" and "No Target". If a "No Target" exception occurs, then the Track Heading Containment Unit will terminate. However, if a "No Sensor" exception occurs, then this gets propagated up to the "Track Heading" step, which invokes a restart handler, causing a repartitioning to occur. In this way, the Track Heading Containment Unit can handle a fault of one of the Obtain Heading Containment Units it uses to perform its task.
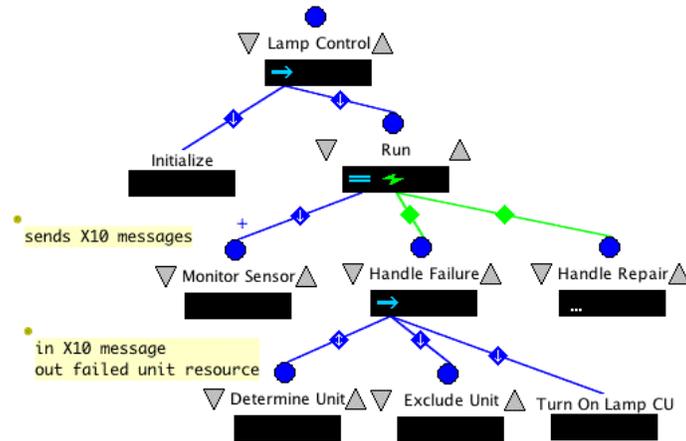
Figure 5 : Light Control Containment Unit

3.3 Lamp Control Containment Unit

The lamp control test bed is built using off-the-shelf X-10 home automation components. A Little-JIL agent serves as an interface between the X-10 network and the Juliette run-time system. The lights are actuated using control modules, and modified "dusk/dawn" sensors are used to monitor the states of the lamps. The resource manager keeps track of which lamps are functional, or have failed, and models the relationships between the sensors and the lamps. Resource queries are used to select functional lamps and to determine which lamp has failed in response to a report from a sensor.

The Little-JIL description of a Lamp Control Containment Unit is shown in Figure 4. This Containment Unit seeks to ensure that an area is illuminated by guaranteeing that one of a set of lamps is always illuminated. The Containment Unit takes as input the set of lamps and begins by initializing the configuration to ensure that one of the lamps in the set is turned on (the "Initialize" step), and then begins to monitor the sensors associated with the lamps. When a sensor detects that the illuminated lamp fails, the containment unit reacts by executing the "Handle Failure" step. This step determines which lamp failed by examining the failure message, removes the failed lamp from the available resources, and finally uses "Turn On Lamp CU" to turn on an alternate lamp from the available resources. "Turn On Lamp CU" is not shown here, but is similar to the Lighting Control Containment Unit presented in [4]. For simplicity, "Handle Repair" is elided in this example. It is similar to "Handle Failure" except that the repaired unit is added to the resource pool.

4. FUTURE WORK

There are several key directions in which we expect to take this work. A central focus of all of them is to be able to reason effectively about the ability of Containment Units to be relied upon to actually contain and respond to faults.

One such direction is to continue our work in analysis of Containment Units. In our previous work we have applied finite state verification tools and technologies to demonstrate that Containment Units do indeed respond as claimed to detected needs for adaptation [3]. This early work seems to us to have demonstrated the feasibility of such reasoning. But this work has succeeded in verifying only a modest set of

adaptation properties and responses. In future work we will investigate the ability to do more powerful reasoning about wider classes of properties. We expect that this work will indicate the need for different and more powerful reasoning tools and technologies.

Related to the above, we will also investigate extensions to the specification formalisms for Containment Units. One area in which this seems particularly important is the specification of resources. We are currently adding increasingly comprehensive resource specifications, focusing on specification of relations and aggregations among resources. We expect this to enable us to specify Containment Units more accurately, to select operational components more effectively, and to perform further reasoning about Containment Units.

Also, we expect to continue our focus on Containment Units designed to deal with physical resources. Our early work with robotics convinces us that this is a fertile area of investigation, as it deals with diverse devices having a rich range of failure modes. It is our belief that future systems will be interesting and complex synergies among software systems, hardware devices, and humans, with each having an important role to play in assuring adequate response to failures. We expect our Containment Units will need to be correspondingly complex and diverse. Thus our work will be broad enough to encompass responses to failures in all three domains.

Finally, this work is already suggesting that specification of Containment Units using Little-JIL indicates the need for changes in the language itself. Thus, as we continue this work we expect to find that Little-JIL itself will need to incorporate changed and new semantic features. One direction we will investigate is the ability to customize Little-JIL to support domain-specific abstractions such as Containment Units more directly.

5. ACKNOWLEDGMENTS

6. REFERENCES

[1] Cass, A. G., Lerner, B. S., McCall, E. K., Osterweil, L. J. and Wise, A. Logically central, physically distributed control in a process runtime environment, TR 99-65, University of Massachusetts, Amherst, Department of Computer Science, 1999, ftp://ftp.cs.umass.edu/pub/ techrept/techreport/1999/UM-CS-1999-065.ps

[2] Cass, A. G., Lerner, B. S., McCall, E. K., Osterweil, L. J., Sutton, Jr., S. M. and Wise, A. Little-JIL/Juliette: A process definition language and interpreter, in Proceedings of the 22nd International Conference on Software Engineering (Limerick, Ireland, June 2000), 754-757.

[3] Cheng, S., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B., and Steenkiste, P. Using architectural style as a basis for system self-repair, in Proceedings of the Working IEEE/IFIP Conference on Software Architecture 2002 (Montreal, August 2002).

[4] Cobleigh, J. M., Osterweil, L. J., Wise, A. and Lerner, B. S. Containment Units: A hierarchically composable architecture for adaptive systems, in Proceedings of the 10th International Symposium on the Foundations of Software Engineering (Charleston, South Carolina, November 2002).

[5] Dellarocas, C., Klein, M., and Shrobe, H., An architecture for constructing self-evolving software systems, in the Proceedings of the 3rd International Software Architecture Workshop (Orlando, Florida, November 1998).

[6] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Johnson, G., Medvidovic, N., Quillici, A., Rosenblum, D. S., and Wolf, A. L. An architecture-based approach to self-adaptive software, IEEE Intelligent Systems (May/June 1999), 54-62.

[7] Wermelinger, M., Lopes, A., Fiadeiro, J. L. A graph based architectural (re)configuration language, in Proceedings of the Joint 8th European Software Engineering Conference and the 9th Symposium on the Foundations of Software Engineering (Vienna, September 2001), 21-32.

[8] Wise, A., Little-JIL 1.0 language report, TR 98-24, University of Massachusetts, Amherst, Department of Computer Science, 1998, ftp://ftp.cs.umass.edu/pub/ techrept/techreport/1998/UM-CS-1998-024.ps