Containment Units: A Hierarchically Composable Architecture for Adaptive Systems

Jamieson M. Cobleigh,
Leon J. Osterweil,
and Alexander Wise
Laboratory for Advanced Software Engineering
Research
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
jcobleig, ljo, wise@cs.umass.edu

Barbara Staudt Lerner
Department of Computer Science
Williams College
Williamstown, MA 01267
lerner@cs.williams.edu

ABSTRACT

Software is increasingly expected to run in a variety of environments. The environments themselves are often dynamically changing when using mobile computers or embedded systems, for example. Network bandwidth, available power, or other physical conditions may change, necessitating the use of alternative algorithms within the software, and changing resource mixes to support the software. We present Containment Units as a software architecture useful for recognizing environmental changes and dynamically reconfiguring software and resource allocations to adapt to those changes. We present examples of Containment Units used within robotics along with the results of actual executions, and the application of static analysis to obtain assurances that those Containment Units can be expected to demonstrate the robustness for which they were designed.

Categories and Subject Descriptors

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

General Terms

Design, Reliability, Verification

Keywords

Adaptive systems, Containment Units, self-healing systems

1. INTRODUCTION

In today's world, software is becoming increasingly pervasive. Not only is software invisibly embedded in the cars we drive, the elevators we take, and the machines we use, but the Internet has made software increasingly visible in a wide variety of operational contexts. Increasingly, people expect access to Web sites and programs from desktop computers, cell phones, handheld devices, etc. For an application to be available and effective across such a range of platforms and in a variety of challenging operational contexts requires careful customization. Examination of various parameters of the usage environment, including such factors as memory, screen size, processor speed, ambient conditions, and communication speed is necessary in order to support the selection of the proper software components and other resources needed to assure satisfactory functioning. Adding to the complexity of this problem is the fact that runtime conditions will often change. Thus, perhaps the communication link speed or reliability changes, perhaps the available memory changes due to behavior of concurrent tasks, or perhaps the processor speed changes to conserve the battery. It is clearly desirable that overall system operation continue in spite of such changes. Often system operation can be continued with the help of resource reallocation or software component modification or substitution. It is clearly desirable that these adaptations take place as quickly and automatically as possible, but only with clear assurances that these automatic adaptations will be correct and effective.

In general, runtime adaptation is frequently needed in applications that are expected to work in a wide range of environments but where customization is required for those different environments. Robotics offers an excellent example. Imagine a team of robots whose task is to rescue people from a fire. The robots may be faced with a wide range of environmental conditions: smoke and heat to name two. While their task remains the same, their selection of sensors and how to interpret the data reported by the sensors vary depending upon the environmental conditions. The environment can change as the robot moves from one room to another and thus dynamic adaptation to this changing environment is essential to success of the mission.

Previously [15], we have introduced an approach to enable software systems to select customized components and satisfactory resource mixes that are suitable for the current environment, to monitor those components and the environment for satisfactory performance, and to adapt the software if the environment changes. We call our realization of this approach *Containment Units*, which are modules able to self-diagnose the need for changes in their operational characteristics and also able to make a limited set of changes aimed at meeting these needs. A Containment Unit is intended to guarantee that it will maintain its capabilities in the face of a range
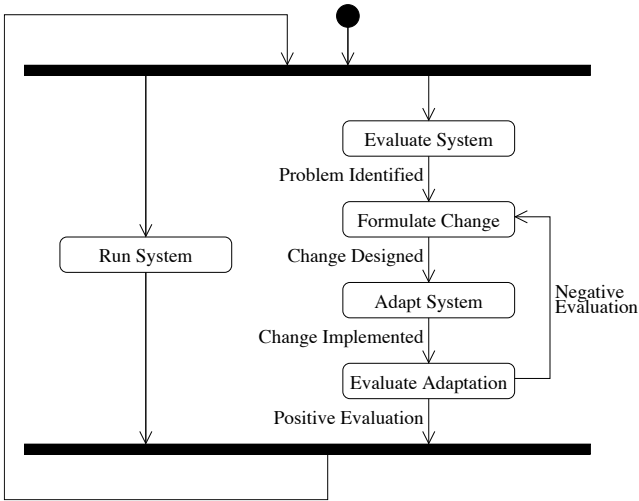
Figure 1: Phases of Software Modification

Figure 2: Containment Unit Architecture

of changes in operational conditions by automatically making internal adjustments. By composing systems out of configurations of Containment Units, we are able to construct more adaptable systems that should need less human involvement in making relatively modest modifications and should be adaptable to important types of changes in their operational environment in minutes or even seconds. A key aspect of our work is its emphasis on automated analysis aimed at deriving assurances that Containment Units can be relied upon to always provide the robustness for which they are designed.

In this paper, we present architectural details of our approach to self-adaptation in software. We present the structure of our key notion of a Containment Unit, discuss the separation of the coordination aspects of a Containment Unit from its operational aspects, and then describe some of our experiences developing Containment Units, and analyzing key properties such as correct adaptation. Finally we compare our approach to other work in this area, and then conclude, suggesting future directions for this work.

2. OUR APPROACH

To explain the ideas underlying the Containment Unit concept, we begin with some observations about the general notion of software modification.

2.1 Some General Architectural Features

Software modification is a process with which there is much experience. Figure 1 is a very high level activity diagram conceptualizing the four main phases of a software modification process. Modification begins by evaluating the behavior of the currently deployed executing system. Not too surprisingly, evaluation often indicates the need for change. At that point, the formulation of a system modification takes place, followed by some alteration of the system, reevaluation of the alteration to determine if it is effective, and the utilization of the modified system if the alteration seems effective. If the alteration is not effective, a new change is formulated, implemented, and evaluated until a solution to the problem is found. The modified system then becomes the subject of a new round of observation, evaluation, and alteration. This process is presumably iterated continuously throughout the lifetime of the system.
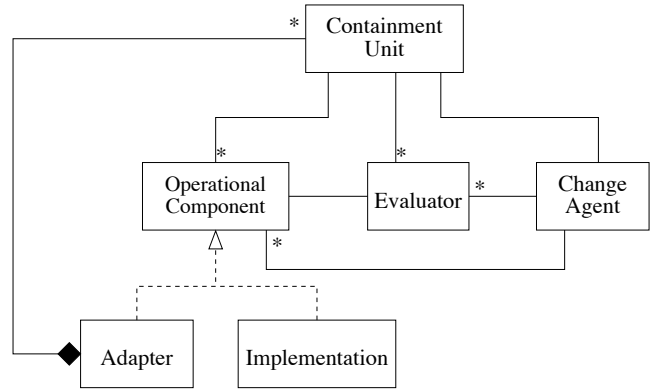
The purpose of modification is to improve system behavior. Thus, increasing execution speed, adding facilities for handling new cases or contingencies, and incorporating more effective response to failure are all examples of possible objectives of modification. Both the details of the modification and the actual modifications performed will vary for different circumstances. Nevertheless, these different modification processes share a common architecture.

Examination of Figure 1 suggests that there are two distinct types of activities entailed in software system modification, namely evaluation and alteration. We propose that each of these capabilities be assigned as the specific responsibility of a different modification component, as these two capabilities correspond to separate concerns. We thus arrive at a very high level representation of the architecture of a generic modification process in which the deployed, operational system is one component, an evaluator is a second component, and a change agent is a third component.

Generally, when we consider software modification, we think of evaluation as being done using principally automated analysis and testing activities while adaptation generally entails a great deal more emphasis on human involvement. To support self-adaptation, we suggest that the same modification architecture applies, but that it entails the automation of the adaptation activity. We refer to an architecture that incorporates these three components as a Containment Unit.

2.2 Containment Units

To be more precise, a *Containment Unit* is a module that, like other modules, encapsulates some functionality and implements that functionality in such a way as to meet specific nonfunctional requirements. Specifically, we represent a Containment Unit interface, CU_{INT}, with a tuple (F, R, CP, FC). F represents the functionality of the Containment Unit. R represents the resource requirements including time, memory, and other shared physical resources such as special processors, sensors, and actuators. CP represents the communication protocol defining the input expected, output produced, and faults reported by the Containment Unit. Finally, FC represents the faults reported by the operational components that are handled by the Containment Unit, that is, situations that the Containment Unit guarantees it can handle internally.

As shown in Figure 2, a Containment Unit implementation, CU_{IMP}, is made up of four major components: $(Top, Op, Eval, Change)$. Top is the top level component that is responsible for initializing the Containment Unit, and managing the communication protocol. Op is a set of operational compo-

nents, $Op = \{op_i\}$, each of which provides the functionality of the Containment Unit. $Eval$ is a set of evaluators, $Eval = \{eval_i\}$, that dynamically monitor the performance of the operational components to ensure that the Containment Unit interface is being satisfied. There may be one or more evaluators. For example, there may be one evaluator to monitor execution speed, another to monitor memory usage, and still another to evaluate the quality of the functional results. The change agent, $Change$, provides a capability for adaptation in the event that one of the evaluators determines that the Containment Unit is not operating satisfactorily.

The purpose of an operational component is to provide the Containment Unit's functionality within the specified resource limitations, as stated in R. Each operational component within a Containment Unit has a specification that is consistent with the specification of its encompassing Containment Unit. In particular, the functionality provided by an operational component must be at least as comprehensive as that provided by the Containment Unit itself. Each operational component must require no more time, memory, and other resources than the Containment Unit as a whole. By doing this, we can be certain that any operational component will be able to satisfy the Containment Unit's functional requirements. But each operational component will probably have environmental constraints that constrain it to be effective in only a subset of the operational environments supported by the overall Containment Unit. In particular, the environmental constraints of a Containment Unit are generally a disjunction of the environmental constraints of the enclosed operational components. This allows the change agent to use information about current environmental conditions to select an appropriate operational component. Operational components may be implemented using other Containment Units to support their hierarchical composition. Such Containment Units are usually constructed through the use of the Adapter pattern [6] to implement the higher-level Containment Units functionality out of lower-level Containment Units.

As mentioned above, each operational component is not required to contain the faults that the enclosing Containment Unit contains. Instead, the roles of the evaluators and the change agent are to ensure that, should a fault arise, the Containment Unit will adapt either by running an alternative operational component or by changing resource allocations so that the fault is handled within the Containment Unit.

The relationship between the operational component specifications and the Containment Unit specification is captured below:

1. $\forall op \in Op_{CU}, F_{op} \geq F_{CU}$ (Each operational component provides at least the functionality required by the Containment Unit). More formally, we define $\geq$ to mean that for all I in the domain of F_{CU}, $F_{CU}(I) = F_{op}(I)$.

2. $\forall op \in Op_{CU}, R_{op} \subseteq R_{CU}$ (Each operational component does not use more resources than the Containment Unit specifies.)

3. $\forall op \in Op_{CU}, INPUT_{op} \equiv INPUT_{CU} \wedge OUTPUT_{op} \equiv OUTPUT_{CU}$ (Each operational component has exactly the same input and output as the Containment Unit.)

4. $\bigwedge_{op} FaultsReported_{op} - FaultsReported_{CU} \equiv FC_{CU} \equiv \bigwedge_{op} FC_{op}$ (The faults contained by a Containment Unit are the difference between the faults reported by the collection of operational components and the faults reported by the Containment Unit. The faults contained by a Containment Unit are also the disjunction over the faults contained by the collection of operational components. That

is, if one operational component reports a fault, but the Containment Unit contains it, it must be the case that a different operational component handles that fault.)

Evaluators are connected to operational components via the Observer pattern [6]. The purpose of the evaluators is to guarantee that the Containment Unit specification is satisfied by dynamically monitoring the behavior of the active operational component. Should the result quality or performance of the active operational component fall outside the Containment Unit guarantees, the evaluator signals an error to the change agent.

The change agent's job is to turn off the current operational component and select an alternative component better suited to the current environment or an alternative allocation of resources to the active component and then to continue. Because the operational components provide a great deal of information about their resource and environmental constraints, we expect that adaptations will lead to improved system behavior as the environment changes.

A Containment Unit configuration CU_{CONFIG} represents a Containment Unit at runtime and consists of $(Op_{CUR}, Eval, Change)$. The key point here is that at runtime, only a single operational component of the implementation will be active at a time. The evaluators and change agent are bound to the Containment Unit, rather than individual operational components. As a result, these remain active independent of which operational component is active.

One particularly important attribute of this architecture is that the signals sent by the evaluators and the structure adaptation procedures in the change agent are defined separately from the implementation details of the operational components. We believe this separation of the coordination aspects of the containment unit architecture will allow the static analysis to obtain assurances of the robustnesses of Containment Units and the safe addition of new operational components.

3. EXPERIENCES WITH CONTAINMENT UNITS

We now describe our experiences in defining, executing, and analyzing Containment Units. While we have exploited our agent coordination language Little-JIL [22, 23] to describe the coordination within a Containment Unit and use our Little-JIL runtime, Juliette [2], as the basis for our execution experiments, the following discussion does not require understanding of Little-JIL.

Our example Containment Units are based on a robot search and rescue example we have been developing. In particular, we focus on an important element of a search-and-rescue task, namely the ability to track a target moving through a room with a set of fixed sensors [11, 1]. We begin by presenting the top levels of a Containment Unit hierarchy, then skip down to one of its lowest levels.

3.1 Obtain Heading Containment Unit

First we will discuss a high level Containment Unit that is designed to obtain a heading towards a target under a wide range of operational situations. This is intended to be used in a search and rescue task, to determine where a target is in a room and to track it as it moves, despite the fact that the room may be smoky, littered with debris, etc.

The Obtain Heading Containment Unit, shown in Figure 3, takes as a resource a set of sensors that it will use to track the target. It begins by selecting an initial sensor to use and then begins reading sensor values from the sensor and writing them into a global space for use by other components, and monitoring the active sensor's
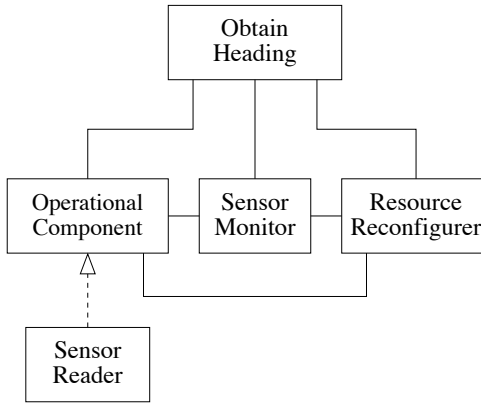
Figure 3: Obtain Heading Containment Unit

performance. The sensor interface allows the sensor to report three exceptional conditions:

- "No Target" if the sensor is unable to detect a target within its observation area
- "Target Lost" if the sensor is tracking a target and then loses it; the goes behind a wall for example, and
- "Sensor Fault" which reports internal diagnostic failures of the sensor.

When the change agent detects one of these failure modalities, it stops reading values from the current sensor and attempts to reconfigure the Containment Unit by selecting an alternate sensor. If there is no target in the observation area or no applicable sensors ("No Sensor"), the Containment Unit terminates.

The interface to the Obtain Heading Containment Unit is (F, R, CP, FC) where:

- F is a function that returns a heading to a target
- R is one or more sensors that can be used to track a target
- CP states that this Containment Unit writes headings into a global space, and reports "No Sensor" and "No Target" as faults
- FC is the faults "Sensor Fault" and "Target Lost"

The implementation of this Containment Unit is a tuple $(Top, Op, Eval, Change)$ where:

- Top selects an initial sensor for the Containment Unit
- Op is a component that reads a sensor value and writes it into the global space
- $Eval$ is a monitoring component for the sensor that detects the various failure modalities
- $Change$ handles the faults "Sensor Fault" and "Target Lost" by reconfiguring the resources for Op

At the conclusion of this section we describe our experiences in using automated static analysis to verify that this Containment Unit performs as intended.

3.2   Track Heading Containment Unit

While the Obtain Heading Containment Unit can determine the heading of a target with respect to a given sensor, it cannot determine the exact position of a target in a room. All that can be said is that the target lies on a given line that runs through the sensor. With two sensors reporting headings, there are two lines that the target lies on. The position of the target can be determined by identifying
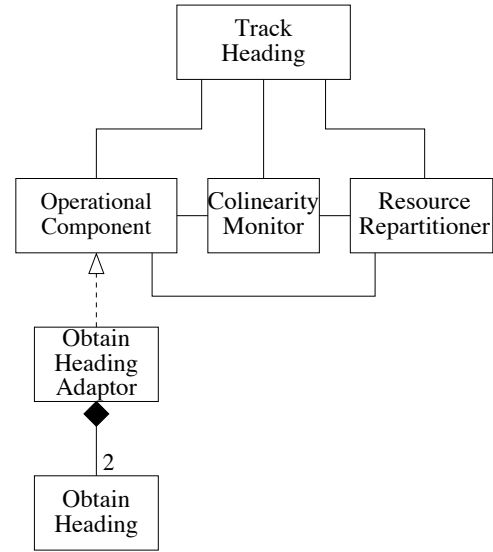

Figure 4: Track Heading Containment Unit

the intersection point of the two lines. This fails, however, if the target is on the line that runs through both sensors. We call this a Colinearity Fault. Thus, to determine the position of a target, we use two Obtain Heading Containment Units and their respective resources. An evaluator is responsible for identifying and reporting colinearity faults.

The Track Heading Containment Unit, shown in Figure 4 takes as input a set of sensors. It begins by dividing the sensors into two disjoint subsets. 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 the partitioning fails, signaling "No Partition," and the Containment Unit terminates. If a partitioning is possible, then an operational component that serves as an adapter between two instances of the Obtain Heading Containment Unit and this Containment Unit by computing the heading based on their outputs.

The computation of a position from two headings introduces a new failure modality: as the target gets closer to a line drawn through the two sensors, the accuracy of the triangulation decreases. The Containment Unit has a monitor that detects this condition and signals a "Colinearity Fault." It is important to note that a 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 a colinearity occurs.

When the change agent is notified of a colinearity fault, or "No Sensor" from one of the nested Containment Units, it responds by repartitioning the resource pool so that there are appropriate resources for each of the two containment units. Due to the hierarchy, this Containment Unit does not have to handle the faults contained by the Obtain Heading Containment Unit. If Obtain Heading Containment Unit cannot locate a target ("No Target") the Containment Unit terminates.

The interface to the Track Heading Containment Unit is (F, R, CP, FC) where:

- F is the (set of) function(s) that return(s) the position of a target
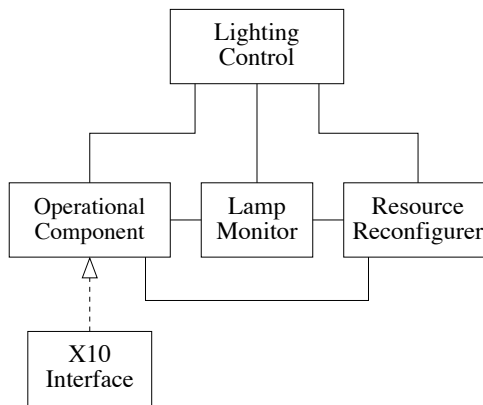
Figure 5: Lighting Control Containment Unit

- R is two or more sensors that can be used in the Obtain Heading Containment Unit
- CP states that this Containment Unit writes positions into a global space, and reports "No Partition" and "No Target" as faults
- FC is the faults "Colinearity Fault" and "No Sensor"

The implementation of this Containment Unit is a tuple $(Top, Op, Eval, Change)$ where:

- Top selects an initial partitioning of the resource set
- Op is an adapter that invokes two Obtain Heading Containment Units and performs the triangulation on their output
- $Eval$ is a monitoring component that detects colinearity between the active sensors and the target
- $Change$ handles "Colinearity Fault" and "No Sensor" by repartitioning the resources for Op

Later in this section we describe the analyses we attempted on this Containment Unit.

3.3 The Lighting Control Containment Unit

As a very basic evaluation of our Containment Unit concepts we used Little-JIL to define a Containment Unit, and then used Juliette to execute the Containment Unit, demonstrating its ability to dynamically reallocate resources to assure desired behavior in the face of failure.

The Lighting Control Containment Unit, shown in Figure 5, is a lamp controller that might be used to provide illumination for use with vision sensors as part of a search and rescue task. Our lamp controller Containment Unit uses the commercially available X10 home automation system. This system includes inexpensive switched electrical outlets, light and motion sensors, and an interface that allows a computer to send and receive messages that use the X10 protocol.

The lamp controller provides illumination when requested by activating the switched outlet attached to a lamp, monitoring the state of the lamp using the light sensors, and in the event that the monitor determines that the lamp has failed, switching to an alternate switched outlet. While very simple, this demonstrates the basic structure and operational components of a Containment Unit.

The interface to the Lighting Control Containment Unit is (F, R, CP, FC) where:

- F is the function providing illumination in a requested area
- R contains the primary and alternate switched outlets, and light sensors for monitoring the associated lamps

- CP states that this Containment Unit reports if illumination cannot be provided
- FC is the fault "Illumination Failure"

The implementation of this Containment Unit is a tuple $(Top, Op, Eval, Change)$ where:

- Top selects a primary and backup lamp, and the appropriate sensor for monitoring the primary lamp
- Op is a component that controls a lamp using the X10 protocol
- $Eval$ is a component that monitors the lamp and reports "Illumination Failure"
- $Change$ responds to "Illumination Failure" by activating the backup lamp

Having defined this Containment Unit using Little-JIL, we were then successful in using Juliette to support running demonstrations showing the automatic switching over to a backup working lamp in response to the failure of an initially selected lamp.

3.4 Analyzing Containment Units

As indicated earlier in this paper, we believe it is particularly important to be able to reason about Containment Units, specifically to be able to demonstrate that they have been successfully defined to assure that their desired robustness properties must always be achieved. Thus a key feature of our work is the application of automated analyzers to verify that defined configurations and resource reallocations do indeed assure desired robustness.

To address this key goal we employed FLAVERS (**FL**ow **A**nalysis for **VER**ification of **S**ystems), a static analysis tool that can verify user specified properties of sequential and concurrent systems [5]. FLAVERS requires that a property to be checked be represented as a Finite State Automaton (FSA). FLAVERS uses an annotated graph model called a *Trace Flow Graph* (TFG), derived from Control Flow Graphs, that captures an overestimate of all possible system executions. The FLAVERS model is highly abstracted so that it is as small as possible. This makes the analysis more tractable but comes at a cost in precision. FLAVERS results are conservative, therefore if FLAVERS determines that a property holds, then it guarantees that the property holds on all possible system executions. If FLAVERS determines that a property does not hold, this can either be because there is a fault in the system or because the property is violated on an infeasible path through the model, a path that does not correspond to any possible execution of the system and result from the imprecision of the model. FLAVERS uses *Feasibilty constraints*, also represented as FSAs, to improve the precision of the model and eliminate some infeasible paths from consideration. FLAVERS uses an efficient state propagation algorithm to verify the property that has worst-case complexity that is $\mathcal{O}\left(N^2 \cdot |S|\right)$, where N is the number of nodes in the TFG, and $|S|$ is the product of the number of states in the property and all constraints.

To apply FLAVERS to determine whether or not some of our Containment Units performed as intended, we manually transformed the Little-JIL descriptions of several Containment Units into Ada as described in [3], and then used FLAVERS to analyze these models. In future work we would hope to use automated translators to render Little-JIL defined Containment Units into graphs amenable to FLAVERS analysis.

In one such project, we modeled the Obtain Heading Containment Unit in Ada, and created the model so that the Containment Unit can receive up to four sensors[1]. This model required 549 lines of Ada code.

---

[1] We were constructing Containment Units while we were develop-

We were interested in ensuring the Containment Unit could successfully contain Sensor Faults, so we checked the property that the Containment Unit would not terminate if three sensor faults occurred. The TFG for this had 120 nodes and 1,592 edges. Proving this property required 11 feasibility constraints. Even with this large number of feasibility constraints, FLAVERS was able to prove the property in less than 1 second.

We also applied FLAVERS to the analysis of the Track Heading Containment Unit. The model for this Containment Unit required 1,873 lines of Ada code, and the model was written so that it could accept up to 4 sensors.

We were again interested in the robustness of this Containment Unit. We wanted to check that if given four sensors, that it could handle one of the sensors failing without requiring a repartitioning. Unfortunately, we were unable to prove this property on this Containment Unit. The number of feasibility constraints needed to prove this property was large enough that FLAVERS was unable to prove this property using 2GB of memory.

While none of the analyses we performed found any faults in any of the Containment Units, it has been shown that Little-JIL coordination specifications can contain subtle faults that can be detected by finite state verification [3], so we believe that applying analysis techniques to Containment Units is important to ensuring their correctness, and work continues on extending FLAVERS to allow it to verify properties of larger software systems.

4. RELATED WORK

Perhaps the earliest work that has addressed adaptation to faults was the work of Randall on recovery blocks [20]. In this work the suitability of a software function was evaluated, and when found to be inadequate, a recovery block was called to try to mitigate the effects of the inadequate code. This early work was quite static in nature, requiring that the conditions to be examined, and the recovery strategies be hard coded in advance. The representation of operational components as resources allows us more dynamism.

Work with real time systems has some relationship to this project as well. The work of [19], [13], [12], and [8], for example, suggest the use of a framework within which to describe operational components and the real time constraints on their performance. These approaches tend to use the real time constraints primarily to determine whether proposed module configurations would necessarily meet real time constraints. In this work, however, unacceptable configurations were often simply not deployed, or ad hoc responses were generated. Our work differs in that we use language constructs to define programmed strategies for dealing with such constraint violations. Like some of these authors we use module replacement as the basis of our work.

Our work is also related to earlier efforts in software reuse. This work, like ours, emphasized the importance of repositories of reusable modules, and the use of architectural frameworks within which to insert them. These approaches are presented in work such as [9, 21, 18]. Our work takes these approaches further in using explicit, rigorous process representations to effect the module reuse.

The work that this project most closely resembles, however, is work in the areas of software architecture and domain specific software. Numerous authors have suggested the use of architectures to guide the composition of software system out of components or modules (e.g., [16, 17, 7]). Our specific approach to module interchange is similar to that suggested by [14] and [4] who propose the use of a defined architecture as the framework within which dif-

ferent components can be interchanged. Containment Units extend this through the inclusion of mechanisms to detect when adaptation is required, and to automate this reconfiguration.

A particular system with a similar goal and approach is Chameleon. Chameleon is an infrastructure for adaptive fault tolerance [10]. The Chameleon system is based on ARMORs, which are components that control all operations in the Chameleon environment. An ARMOR can be thought of as a wrapper around a component or set of components. Each ARMOR provides a specific fault tolerance capability and the Chameleon architecture supports specific failure modes and recovery mechanisms. Some example ARMORs include the Heartbeat ARMOR, which can be used to query a component to see if it is up or down, the Checkpoint ARMOR, which saves the state of the component so it can be resumed from the checkpoint in the event of failure, and the Voter ARMOR, which implements n-version programming. Like Chameleon, Containment Units are designed to support the hierarchical composition of fault-tolerant components. We believe Containment Units are a more general mechanism because a Containment Unit can have a wider range of adaptations available to it than those provided by an individual ARMOR

5. FUTURE DIRECTIONS

In this paper, we present Containment Units – modules that provide the basis for building a self adaptive system and show how they can be analyzed to prove their correctness.

The current Containment Unit architecture supports the switching between different resource configurations and operational components at run-time as long as the transition between configurations is relatively simple. For the robotic search and rescue platform, components do not require detailed initial state to begin execution, and executions can overlap without interference. However, we recognize that this is not always the case. The Containment Unit architecture should include a mechanism for the orderly and safe transition from one configuration to another.

Also, one of the goals of the Containment Unit architecture is to allow new configurations and components to be added to running systems in order to allow them to adapt to new operational contexts. While we believe that the separation of operational components from coordination structure will greatly assist us in this goal, we have only a little experience to support this claim. The separation of resources in the Containment Unit model allows us to change the collection of resources that can be deployed at run-time, and we have some experience with Little-JIL programs adapting their behavior based on these change, but we do not yet have any experience replacing the operational components.

6. ACKNOWLEDGMENTS

ing the Containment Unit architecture. As a result, our analyses are based on older versions of the Containment Units.

7. REFERENCES

[1] E. G. Araujo, D. R. Karuppiah, Y. Yang, R. A. Grupen, P. A. Deegan, B. S. Lerner, E. M. Riseman, and Z. Zhu. Software mode changes for continuous motion tracking. In *International Workshop on Self Adaptive Software*, Apr. 2000.

[2] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. TR 99-65, University of Massachusetts, Department of Computer Science, Nov. 1999.

[3] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 96–101, Aug. 2000.

[4] C. Dellarocas, M. Klein, and H. Shrobe. An architecture for constructing self-evolving software systems. In *Proceedings of the Third International Software Architecture Workshop*, pages 29–32, Nov. 1998.

[5] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, Apr. 1995.

[8] O. González, H. Shrikumar, J. A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proceedings of the Eighteenth Real-Time Systems Symposium*, pages 79–89, Dec. 1997.

[9] M. Griss and K. Wentzel. Hybrid domain-specific kits for a flexible software factory. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 47–52, Mar. 1994.

[10] Z. T. Kalbarczyk, S. Bagchi, K. Whisnant, and R. K. Iyer. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.

[11] D. R. Karuppiah, Z. Zhu, P. Shenoy, and E. M. Riseman. A fault-tolerant distributed vision system architecture for object tracking in a smart room. In *International Workshop on Computer Vision Systems*, July 2001.

[12] J. H. Lala, R. E. Harper, and L. S. Alger. A design approach for ultrareliable real-time systems. *IEEE Computer*, 24(5):12–22, May 1991.

[13] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *IEEE Computer*, 23(7):39–51, July 1990.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the Twentieth International Conference on Software Engineering*, pages 177–186, Apr. 1998.

[15] L. J. Osterweil, A. Wise, J. M. Cobleigh, L. A. Clarke, and B. S. Lerner. Architecting dynamic systems using containment units. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, Dec. 2001.

[16] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–38, Mar. 1979.

[17] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[18] R. Prieto-Díaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, May 1993.

[19] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), Apr. 1990.

[20] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[21] W. Tracz. *Confessions of a Used Program Salesman*. Addison-Wesley, 1995.

[22] A. Wise. Little-JIL 1.0 language report. TR 98-24, University of Massachusetts, Department of Computer Science, 1998.

[23] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, pages 155–163, Sept. 2000.