

Architecting Dynamic Systems Using Containment Units

Leon J. Osterweil, Alexander Wise,
Jamieson M. Cobleigh, Lori A. Clarke
Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts
Amherst, MA 01002
USA
{ljo, wise, jcobleig, clarke}@cs.umass.edu

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

Abstract

Software modification can require as much time, human effort, and expense as the original development, so considerable software engineering research has been directed toward identifying ways in which software can be developed to facilitate subsequent change. One highly successful approach is to develop software using modules, or objects, each of which seals within itself decisions or secrets that are key to successfully addressing its requirements. When one of these requirements changes, it should be expected that the need to meet this changed requirement is to be satisfied by making appropriate changes to the module intended to satisfy that requirement. These changes are usually done manually and off-line. In our work we are exploring an approach where software systems make changes to themselves while executing.. Our approach is based on Containment Units, which are modules able to self-diagnose the need for changes based on their operational characteristics and then to make a limited set of changes aimed at meeting these needs.

1. Introduction

One of the most fundamental changes in attitude about software has been the realization that software must be expected to change over time. Accordingly approaches to facilitating software change are of great interest in contemporary software engineering. The current situation is that a software modification can require as much time, effort, and expense as the original development—and perhaps even more. Further, software modification is often done hastily and poorly, rendering subsequent modifications increasingly difficult and costly. Considerable software engineering research has been

directed towards identifying ways in which software can be developed to facilitate subsequent change.

One highly successful approach is to develop software using modules, or objects, each of which seals within itself decisions or secrets that are key to successfully addressing its requirements. When one of these requirements changes, it should be expected that the need to meet this changed requirement is to be satisfied by making appropriate changes to the module intended to satisfy that requirement. In the past such changes to modules were typically carried out by humans, who either modified affected modules, or replaced them outright with alternatives deemed better suited to meeting the new requirements. Having humans make these changes “offline” generally takes a period of days, weeks, months, or even years. But, the world has become a very fast-paced place that demands more rapid changes.

In our work we are exploring an approach to supporting the ability of software systems to make changes to themselves while they are still executing. The approach suggests how to develop software systems that can automatically diagnose the need for certain types of changes, hypothesize how to make some limited kinds of changes, evaluate the hypothesized changes, and then effect those changes, all while the software is continuing to execute. While ultimately we intend to use our approach to support more radical modifications, in the near term the changes we envisage are of a relatively modest sort, being based upon module substitution and resource reassignment, all within existing architectures that would not change during execution. We refer to this type of change as adaptation, suggesting that the software is to be responsible for making alterations to itself within a well-circumscribed range in order to render itself more suitable.

Our approach is based on *Containment Units*, which are modules able to self-diagnose the need for changes in their operational characteristics and also able to make a

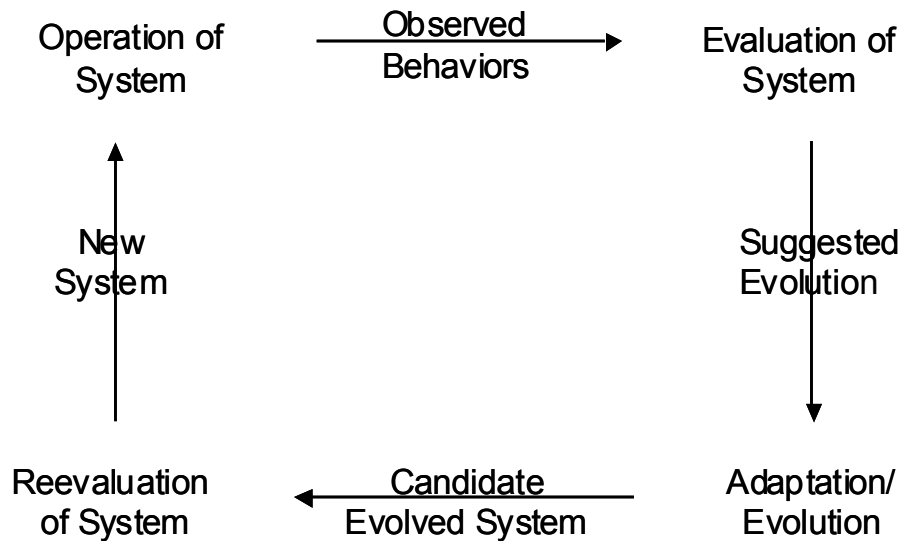


Figure 1: Phases of software modification

limited set of changes aimed at meeting these needs. In an important sense, a containment unit is intended to guarantee that it will maintain its capabilities in the face of a range of changes in operational conditions by automatically making internal adjustments. We envisage that, by composing systems out of configurations of containment units, we should be able to construct more adaptable systems that should need less human involvement in making relatively modest modifications. Such systems should be adaptable to important types of changes in operational environment in seconds or minutes, rather than days or months.

Initially we expect that each containment unit will support only a limited number of adaptations. But systems composed from a number of containment units should support a number of adaptations that is the product of the number of adaptations of its different containment units. This number could easily surpass the number of different variants of a system that might be built and evaluated in advance of deployment. Thus, this approach would thereby offer the additional advantage of increased system flexibility.

2. Our Approach

To understand the ideas underlying the containment unit concept it seems useful to begin with some observations about the general notion of software modification.

2.1. Some General Architectural Features

Most fundamentally, we observe that software modification is a process. Figure 1 is a very high level data flow diagram conceptualizing the four main phases of a software modification process. Note that 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. 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.

One key to understanding this process is to recognize that the process could itself be software and, like all types of software, is intended to satisfy specific requirements. In this case, the requirements with which we are primarily concerned involve improvements of some kind. Thus, for example, increasing reaction speed, adding facilities for handling new cases or contingencies, or incorporating more effective response to failure, are all examples of possible objectives of a modification process. Although the specific modifications made will vary from one modification to another, these different modifications share a common process.

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

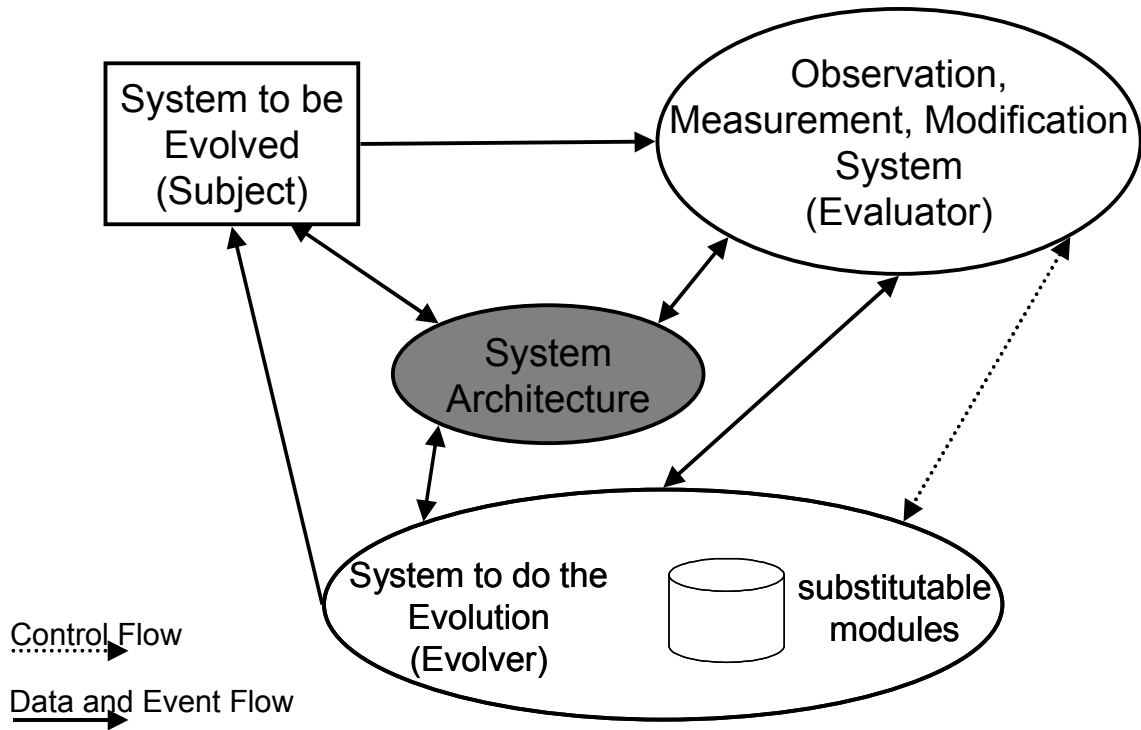


Figure 2: Self-Adaptive System: Data and Control Flows

specific responsibility of a different modification process component, as these two capabilities correspond to separate concerns. We thus arrive at a high level representation of the architecture of a generic modification process. This architecture is shown in Figure 2. In this architecture, there are evaluator and alterer components that use and perhaps modify the deployed, operational system and its architecture.

2.2 Containment Units

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 requirements. Specifically, we represent a containment unit interface, CU_{INT} , with a tuple (F, R, CP, ENV, 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. ENV represents the environmental constraints under which the containment unit can operate. These will typically be related to the physical resources. For example, sensors may require a certain amount of light to operate well. Finally, FC represents the faults contained explicitly. These identify the exceptional situations that are contained by the containment unit, that is, situations

that the containment unit guarantees it can handle internally.

A containment unit implementation, CU_{IMP} is represented as a tuple, $(Op, Eval, Adapt)$. Op is a set of operational components, $Op = \{opi\}$, which provide the functionality of the containment unit. Generally Op will contain several operational components that can be selected alternatively depending upon the environmental conditions. In many cases, an opi may itself be a containment unit, which we refer to as a *subcontainment unit*. This is the principal way in which to create a hierarchy of containment units. $Eval$ is a set of evaluators, $Eval = \{evali\}$, 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, it would be common to have one evaluator monitor execution speed and another monitor memory usage, and still another evaluate the quality of the functional results. The adapter, $Adapt$, is a capability for adaptation in the event that one of the evaluators identifies that the containment unit is not operating satisfactorily. The adaptation process generally consists either of putting into execution a different opi as a substitution for the currently executing opi , or alternatively by reallocating the resources that are available to the containment unit. The purpose of the operational components is to provide the containment unit's functionality within the specified time, memory, and resource limitations. Each operational component within a

containment unit has a specification that is not inconsistent 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. On the other hand, the time, memory, and resource requirements might well be expected to be less stringent. 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 adapter to use information about current environmental conditions to select an appropriate operational component. Due to this similarity between operational component specifications and containment unit specifications, it is possible to compose containment units hierarchically.

As mentioned above, each operational component is not required to contain the faults that the enclosing containment unit contains. Instead, the role of the evaluator and adapter is 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. Should the containment unit be unable to deal with the fault it must then signal a fault that can, in turn potentially be handled by higher level containment units.

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 adapter. The adapter'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 of the semantic richness associated with each operational component we expect that the containment unit should be effective in making modifications across the full range of semantics addressed by the interface specification. The containment unit architecture described above is undergoing evaluation. Our expectation is that it will work well in environments in which alternative algorithms and/or resources are required in different environmental situations, but where the environmental situations either cannot be predicted in advance or are susceptible to change during execution. In these cases, we believe the rich interface specifications,

the continuous monitoring of operational behavior, and the ability to dynamically adapt to the changing circumstances will all be of value in supporting the design of more resilient systems. Example domains for which this will be applicable include robotics and smart home applications.

3. Experiences with Containment Units

One of our early demonstrations provides a useful example of the value of our ideas about containment units. In this early demonstration we are building a software system to use sensors, vision analysis software, and electrical socket controls to provide continuing illumination of a human subject as the subject moves around a room. We assume that the room has a set of electric lights all under computer control, and that the room is instrumented with a variety of sensors. It is our intention to show that containment units are useful in developing a software system that can continue to keep the human subject illuminated as the human moves from place to place, despite such changes in operational environment as variation in illumination levels, changes in ambient conditions in the room (e.g. presence of smoke), and the failure of various devices (e.g. a light bulb).

The hardware devices we are using as resources consist of a set of effectors, which are devices that are capable of switching electrical power on and off, and a set of sensors capable of detecting the presence of a human. The effectors implement the commercial X10 standard for receiving signals and using them to either turn on or turn off the power coming from electrical sockets. In our demonstration, these sockets contain light bulbs intended to provide illumination for a sector of a room. The sensors are both optical and pyroelectric sensors, each capable of detecting the presence of humans. In the case of the optical sensors, this is done by generating images that then undergo analysis by any of a variety of vision analysis software modules. In the case of the pyroelectric sensors, this is done by detecting the heat generated by a human body and analyzing the heat spectrum received. The sensors are directional, and can be turned on axis to locate subjects. Once a subject has been detected, the sensors can then report the angular heading at which the subject was detected, and this information can then be used to identify nearby lighting sources, which can then be turned on to keep the subject illuminated. Lighting sources that are no longer in the vicinity of the human are then to be turned off.

We are seeking to assure that our system remains effective even in the face of a variety of complicating factors. For example, the sensors are assumed to remain in fixed positions, while the human subject is free to move around. Thus, different sensors must be used as the human

moves from place to place. Some sensors are more effective than others in low lighting conditions. Some (e.g. the pyroelectric sensors) do not require light at all. Some of the vision analysis software to be used to identify humans is faster than some other software. Some software works more effectively in the presence of smoke, and in low light conditions, etc. We also assume that some light sources may fail (e.g. lights may burn out), and some X10 signals may not be received due to interference on the electrical lines. We seek to develop a software system that will be robust in the face of all of these difficulties.

A principal vehicle for assuring this robustness is the provision of ample resources to allow for flexibility in pursuing the human tracking activity. Thus, sensors may overlap in the areas that they cover, and in the way in which they work. Vision analysis software components may have overlapping capabilities, but may have different degrees of effectiveness under different operating conditions. And different illumination sources will be able to provide illumination to some of the same areas. By providing these redundant and overlapping resources it should be possible to provide substantial robustness. Clearly we would like to assure that more lavish supplies of resources lead to more robustness. But programming a software system to assure all of this remains a challenge. It is to meet such challenges that we have designed the notion of a containment unit.

To achieve the robustness just described we are building a hierarchy of containment units. At the top of the hierarchy is a containment unit called *Track_Human* that is assigned a variety of resources and has a considerable amount of flexibility in assigning them to subcontainment units in order to continue doing its job.

Thus, we are defining *Track_Human* to be the tuple (F, R, CP, ENV, FC), where:

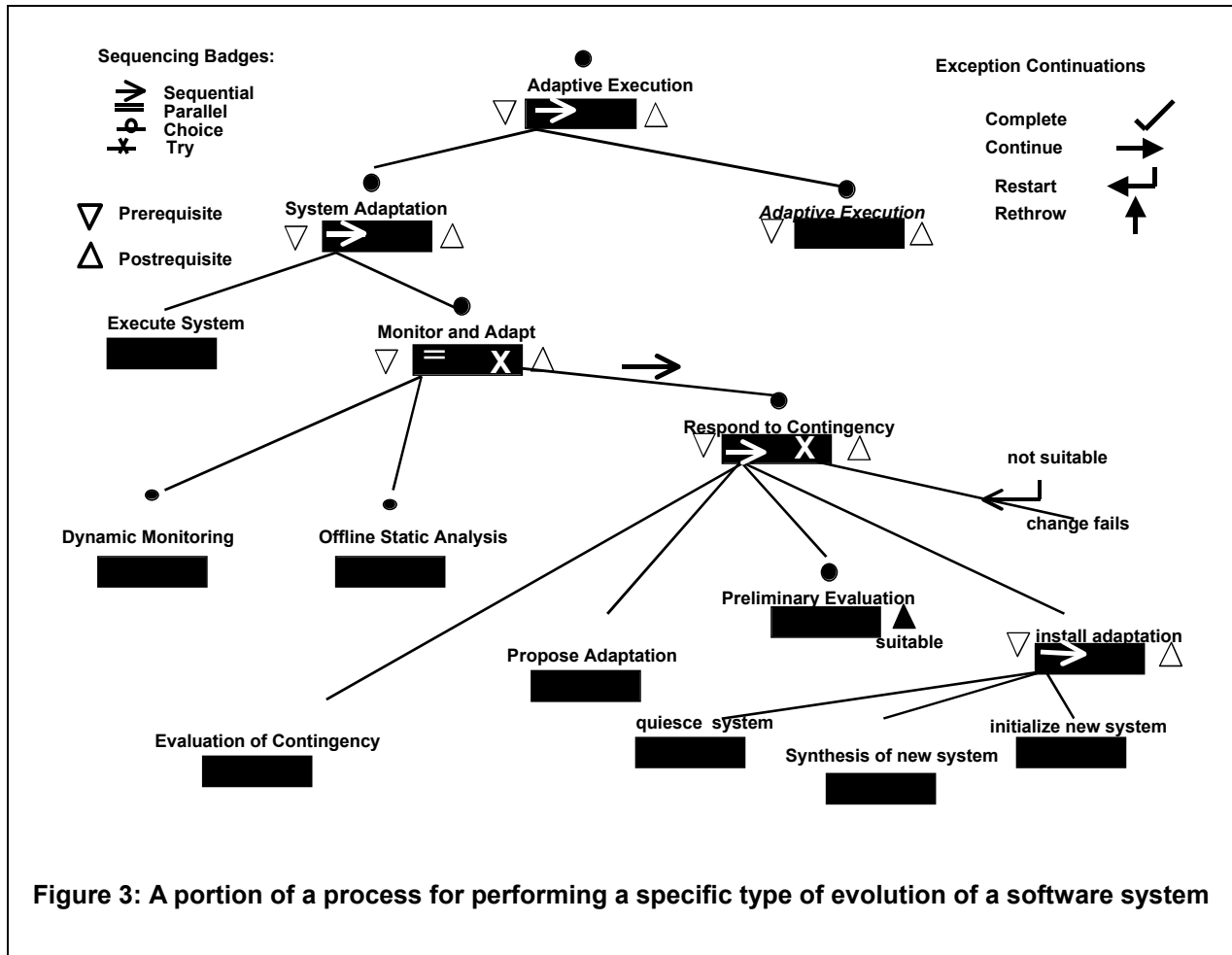
- F is a function that continues to report the (x,y) location of a human as the human continues to move around the room;
- R consists of an array of illumination sources placed redundantly around the room, as well as an array of both vision and pyroelectric sensors also placed redundantly around the room;
- CP consists of a definition of the stream of output locations generated by this containment unit, as well as a definition of the faults reportable (presumably that the human was no longer being tracked), as well as an indication of the conditions that combined to cause this fault (for example human moved too quickly in low light conditions);
- ENV consists of a specification of the maximum speeds at which humans can move under various illumination conditions: and finally
- FC represents such faults as low light, smoky room conditions, and single illumination source failure.

Encapsulated within *Track_Human* is a collection of subcontainment units, each of which is able to track a human, but each under somewhat different circumstances, and each with a somewhat different complement of resources. Thus, for example, one of these subcontainment units, *Track_Human_Pyro*, is able to track a human using only a pyroelectric sensor, but can do so in the dark. This sensor, however, suffers from a relative lack of precision, and is relatively poor in tracking humans who move relatively rapidly. The tuple defining *Track_Human_Pyro* is quite similar to the tuple defining *Track_Human*. For example, its F and CP components would be the same. But R, its resource complement, would contain only the pyroelectric sensors, ENV would be a specification of the maximum speed at which the pyroelectric sensors can track humans, and FC represents the fault that occurs when humans move faster than the speed at which the pyroelectric sensors are effective.

The adaptor component of *Track_Human* is programmed to switch to this subcontainment unit when another subcontainment unit throws an exception indicating that it is unable to track a human because of insufficient light. That exception might be thrown, for example, by *Track_Human_Panos*, which is another subcontainment unit. The subcontainment unit uses two redundant panoramic sensors, each of which is able to track a human using vision software components. This containment unit is relatively robust, but fails when the subject is not illuminated at all. With sufficient illumination, however, one of the panoramic vision sensors can track a human rapidly and accurately. If this panoramic sensor fails, then the adaptor component of this containment unit is programmed to switch over to a second panoramic sensor, which is being held in reserve as a backup resource by this containment unit.

At the bottom of this hierarchy of containment units are some low level containment units that are designed to be robust with respect only to a small number of very narrowly defined failures. Thus, for example, one of these low level containment units, *Illuminate_xyz*, is designed only to assure that an illumination source actually provides the requested illumination. For this containment unit:

- F represents an illumination level above specified threshold level at location (x,y,z);
- R consists of two redundant illumination sources, each of which is capable of providing a satisfactory level of illumination at (x,y,z);
- CP consists of a specification of how (x,y,z) is to be delivered as input, how the illumination level observed at (x,y,z) is to be delivered as output, and a specification of the fault that sufficient illumination was not deliverable;



- ENV specifies that the proper working of the illumination detector is required; and
- FC specifies that the failure of a single illumination source (but not two illumination sources) is contained by this containment unit.

While we have begun to define several of these containment units, we have to date implemented relatively few of them. We have implemented a preliminary version of *Illuminate_xyz*, for example. Recall that a containment unit implementation, CU_{IMP} , is represented as a tuple, $(Op, Eval, Adapt)$, where Op is a set of operational components, $Op = \{opi\}$, which provide the functionality of the containment unit. $Eval$ is a set of evaluators $Eval = \{evali\}$ that dynamically monitor the performance of the operational components to ensure that the containment unit interface is being satisfied. And $Adapt$, is a capability for adaptation in the event that one of the evaluators identifies that the containment unit is not operating satisfactorily.

In our initial implementation of *Illuminate_xyz*:

- $Eval$ consists of a component for monitoring to assure that the light level at (x,y,z) is sufficiently high. This procedure makes use of a photoelectric cell (which is part of the resources allocated to this containment unit) that will report whether the command to turn on the illumination source actually resulted in sufficient illumination. Failure of the light to illuminate will cause $Eval$ to throw an exception that triggers $Adapt$.
- Op consists of two different X10 units controlling the flow of electricity to the two different light sources that are nearest to (x,y,z) . Each is capable of modifying the level of illumination in the subject area.
- $Adapt$ responds to notification from $Eval$ that there is insufficient illumination by moving the flow of electricity from the primary (i.e. Closest) X10 device to the device that is next closest to (x,y,z) . $Adapt$ responds to notification that this has been inadequate by signaling a fault for this containment unit as a whole. Presumably there is a higher level, containing, containment unit that will take this fault

and pursue further remedial actions (e.g. illuminating another light, notifying a human, or attempting some other kind of repair).

As our work proceeds we are implementing increasingly complex containment units, with the goal of completing the implementation of the full *Track_Human* structure. It should be emphasized, however, that this nest of containment units is designed to be robust only with respect to specific contingencies. As the need to deal with additional contingencies arises (e.g. the sudden arrival in the room of additional humans) additional containment units and structural complexity will be needed. Understanding how to deal with this growing complexity is one of the key goals of this research.

4. Using Little-JIL to Define Containment Units

Our Little-JIL language has turned out to be very effective in defining containment units. The details of the Little-JIL language are provided in other papers [1, 19], and space does not permit us to repeat them here. Instead we use Figure 3, 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 construct, depicted as a solid black bar with accompanying badges, is the central feature of the language, and is particularly appropriate as a vehicle for representing containment units.

The Little-JIL step synthesizes notions of proactive control, reactive control, resource specification, concurrency, artifact flow, real time specification, and exception management in ways that provide the power needed to specify a containment unit. Little-JIL steps are hierarchical compositions of lower level steps. Thus, for example Figure 3 depicts the step, "Adaptive Execution", which is decomposed into the structure of the substeps shown below it. The second level steps represent the fact that "Adaptive Execution" is called iteratively after each "System Adaptation" activity has taken place, thereby representing the unceasing nature of adaptation. "System Adaptation" itself is composed of two parallel activities, namely "Execute System" and "Monitor and Adapt", which represents the fact that monitoring goes on continually during execution of the subject system. "Monitor and Adapt" in turn consists of two parallel types of monitoring activities (static and dynamic), and a programmed response to the identification of the need to adapt the system. This response is shown as being exception driven, rather than proactive. At the bottom of this hierarchy are abstract representations of functions implemented as executable code. As in the case of containment units, Little-JIL steps need not 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. Thus, note that in Figure 3 the "Preliminary Evaluation" step has a postrequisite step, "Suitable," which represents a potentially complex process of deciding if the proposed reconfiguration is going to be suitable.

It is important to note that steps such as "Preliminary Evaluation" and "Offline Static Analysis", which effect evaluation, are indeed specifications of procedures and methods belonging to Eval, the evaluator module of the containment unit, even though one ("Preliminary Evaluation") is shown being used to support the modification activity. The Little-JIL step decomposition structure is used to indicate which actions are to be carried out at what times and in what ways. Clearly some of these actions at times both invoke, and are invoked by Adapt, that adaptor module. Conversely, as noted, adaptor module actions are invoked, through the Little-JIL contingency handling capability, from evaluator actions. Our early experiences are suggesting that Little-JIL language structures are quite useful in specifying intricate control and artifact flow in containment units that are complex syntheses of diverse evaluation and adaptation capabilities. Figure 3 is a very high level representation of a generic reconfiguration process. Little-JIL supports the incremental addition of further detail quite nicely. Thus, for example, it is possible to use Little-JIL to elaborate on the "Offline Static Analysis" step, to provide details of the intricate interplay between dynamic analysis and static analysis. Some indications of the nature of this interplay are provided in [11, 12].

Clearly Little-JIL is a graphical language, and our experience suggests that its carefully chosen iconic representations help to make complex process specifications 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. Process specifications are interpreted by the Juliette process interpreter, a distributed system described in [2]. Juliette enables support for the late binding of containment units to adaptation processes, thereby allowing for the continuous incorporation of new containment units without the need to modify Little-JIL adaptation processes. In order to explain this we now introduce the Little-JIL resource management capability.

Every Little-JIL step may (optionally) 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 above. If a resource is available, it is bound to the requesting step and execution proceeds.

The adaptor module of a containment unit uses this facility, in that the pool of resources and alternative functional modules available for substitution in Op, the operational system, are considered to be resources. Thus, those methods and functions of the adaptor that must deal with this pool of resources, their characteristics, and ontologies, would require the ability to access information about this pool. The specification of this pool would be a resource specification for the steps representing those methods and functions. It is most important to note that, as the resource pool is a separate module, it is quite free to evolve itself, independently of the reconfiguration of the operational software. Thus, we anticipate that new operational modules and resources might very well be added to the resource pool continuously as they become available. Because resources are bound to Little-JIL steps at runtime, modifier steps needing operational modules or new resources would be able to utilize newly added modules or resources as soon as they are made available through the resource manager. This assures that systems implemented as containment unit structures using Little-JIL will be able to evolve in new ways, as soon as new operational modules and resources are created and made available.

5. Related Work

Perhaps the earliest work that has addressed adaptation to faults was the work of Randall on recovery blocks [17]. 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. This approach seems insufficient to achieve the kind of flexible and rapid adaptation to fluid situations that is required in modern systems. The sort of late-binding approach

described here seems more appropriate to these demanding requirements.

Earlier work with real time systems has some relationship to this project as well. The work of [16], [9], [8], and [5], 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 [6, 15, 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., [4, 13, 14]). Our specific approach to module interchange is similar to that suggested by [10], and [3] who propose the use of a defined architecture as the framework within which different components can be interchanged, although we believe that the range of reconfiguration issues that we address in our work is broader than in these earlier efforts. Another closely related project is Chamelon [7], which has been used to develop ARMORS that add fault tolerance by wrapping existing software. It might be possible to build containment units uses ARMORS.

6. Future Directoins

There are ongoing joint efforts between software engineering, robotics, and computer vision researchers to encapsulate various combinations of sensors into containment units capable of tracking humans in a room. In the intermediate future we expect to complete work on prototype capabilities in this area and to integrate them with smart illumination containment units to fully implement the containment unit structure described earlier in this paper. This work will help us to validate our notions of containment units. In particular the development of ambitious structures of containment units

will help us to determine how well these ideas scale to address the need for highly dynamic self-adaptation in response to broad ranges of contingencies.

This work is also providing continuing validation of the Little-JIL process definition capabilities and our initial notions of the value and their use in implementation of containment units. We shall continue these activities in collaboration with our research colleagues in the other areas, eventually leading to prototype demonstrations of unusually adaptive robotic systems, and deeper understandings of the nature of reconfiguration and mechanisms for achieving it.

7. Acknowledgements

We wish to thank the dozens of colleagues who have participated in this project through their work in robotics, computer vision, real time programming and multiagent systems. We are particularly grateful to Aaron Cass for his many stimulating and insightful discussion of this work. In addition we would like to thank Krithi Ramamritham and Harikrishna Shrikumar for their work in designing and implementing the earliest containment units, Rod Grupen, Elizeth Araujo, and Patrick Deegan for their work in identifying robotic functions to be encapsulated, Ed Riseman, Alan Hanson, Deepak Karrupiah, and Zhigang Zhu for their work in identifying and helping to encapsulate computer visions functions, and Victor Lesser, Regis Vincent, Tom Wagner, Anita Raja, and Shelley Zhu for their work in establishing interfaces between our resource management system and their multiagent schedulers. Finally, the work of Rodion Podorozhny and Anoop George Ninan in developing our resource manager was of great value to this project.

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231 The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U. S. Army, or the U.S. Dept. of Defense.

Bibliography

- [1] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr., and A. Wise, "Little-JIL/Juliette: A Process Definition Language and Interpreter," presented at ICSE 2000, International Conference on Software Engineering, Limerick, Ireland, 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," University of Massachusetts, Computer Science Department, Amherst, MA, Technical Report UM-CS-1999-065, November 1999.
- [3] C. Dellarocas, M. Klein, and H. Shrobe, "An Architecture for Constructing Self-Evolving Software Systems," presented at Third International Software Architecture Workshop (ISAW-3), 1998.
- [4] D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Transaction on Software Engineering*, vol. 21, April 1995.
- [5] O. J. Gonzalez-Gomez, H. Shrikumar, K. Ramamritham, and J. A. Stankovic, "Adaptive Fault Tolerance and Graceful Degradation Under Dynamic Hard Real-Time Scheduling," presented at Eighteenth IEEE Real-Time Systems Symposium, 1997.
- [6] M. L. Griss and K. D. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory," presented at 1994 ACM Symposium on Applied Computing (SAC 94), New York, 1994.
- [7] 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*, vol. 10, pp. 560-579, 1999.
- [8] J. H. Lala, R. Harper, and A. L. A., "Design Approach for Ultrareliable Real-Time Systems," *IEEE Computer*, vol. 24, 1991.
- [9] J.-C. Laprie, J. Arlat, and C. Beounes, "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *IEEE Computer*, vol. 23, 1990.
- [10] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based Runtime Software Reconfiguration," presented at 20th International Conference of Software Engineering, Kyoto, Japan, 1998.

- [11] L. J. Osterweil, "A Strategy for Integrating Program Testing and Analysis," in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds. Amsterdam: North-Holland, 1981, pp. 187-229.
- [12] L. J. Osterweil, "Perpetually Testing Software," presented at 9th International Software Quality Week, 625 Third St. San Francisco CA, 1996.
- [13] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 128-138, 1979.
- [14] D. E. Perry and A. L. Wolf, "Foundations for Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40-52, 1992.
- [15] R. Prieto-Diaz, "Status Report: Software Reusability," *IEEE Software*, vol. 10, pp. 61--66, May 1993.
- [16] K. Ramamritham, J. A. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *Transactions on Parallel and Distributed Systems*, vol. 1, pp. 184-194, 1990.
- [17] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions On Software Engineering*, vol. 1, pp. 220-232, June 1975.
- [18] W. Tracz, *Confessions of a Used Program Salesman*: Addison-Wesley, 1995.
- [19] A. Wise, "Little-JIL 1.0 Language Report," Department of Computer Science, University of Massachusetts at Amherst, Technical Report 98-24, April 1998.