An Approach to Self-Adaptive Software based on Supervisory Control

Gabor Karsai, Akos Ledeczi, Janos Sztipanovits
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA
{gabor,akos,sztipaj}@vuse.vanderbilt.edu
Gabor Peceli, Gyula Simon, Tamas Kovacshazy
Department of Measurement and Information Systems
Technical University of Budapest, H-1521 Budapest, Hungary
{peceli,simon,khazy,}@mit.bme.hu

Abstract:

Self-adaptive software systems use observations of their own behavior, and that of their environment, to select and enact adaptations in accordance with some objective(s). This adaptation is a higher-level system function that performs optimizations, manages faults, or otherwise supports achieving an objective via changes in the running system. In this paper, we show how this capability can be realized using techniques found in hierarchical control systems, and we discuss interrelated issues of stability, assurance, and implementation.

Keywords: self-adaptive software, supervisory control, hierarchical control, fault-tolerance, reconfiguration

Introduction

Self-adaptive software seems to offer novel capabilities that are very hard to achieve using other methods. Software that adapts itself to momentary situations and requirements is envisioned as the vehicle for building complex applications that are robust and fault-tolerant, yet flexible and responsive. Robustness and fault-tolerance is typically achieved by software redundancy and exception management, although purely software-based fault tolerance is yet to be demonstrated in a practical situation. Flexibility and responsiveness is typically achieved by explicitly "designing in" all the alternatives in the system, and verifying that the system reacts properly in each situation. Unfortunately, the (somewhat) contradictory requirements of robustness and flexibility impose a big burden on the designer, as there is no established design approach and technique for self-adaptive software systems.

Self-adaptivity causes further problems. The complexity of today's systems makes it very difficult to explicitly enumerate (and verify) all states of a system, although we must do this for some applications [1]. If the system also exhibits self-adaptive behavior, the situation gets even worse as it is very hard to predict what a system will do if it can modify its own behavior and/or structure. To draw a parallel from control theory, it took about 20 years to prove a simple property: stability about adaptive controllers [2]. The point is that when a systems' behavior space is enriched by another dimension: adaptivity, it becomes extremely difficult to formally analyze that space.

We can recognize that unrestricted adaptivity may be just a new name for self-modifying code. However, restricted, well-designed, and engineered adaptivity is something worth considering, and it is a prime candidate for addressing the needs of robustness and flexibility mentioned above. According to our knowledge, the only engineering field that deals with engineered adaptivity is adaptive control theory [3]. In this paper, we show how techniques invented by control theorists and engineers can be applied to design and implement self-adaptive software, and what type of lessons control engineering teaches us.

Background

Control theory and engineering has been using adaptive techniques since the 1960s [3]. The prevailing principle for adaptive control is as follows. When a particular controller is designed for a system, the engineer makes certain assumptions about the dynamics of the plant. These assumptions may not hold over the lifetime of the deployed system, and the controller may not work optimally when circumstances change. Hence, an adaptation component is introduced that revises (re-tunes) the controller as the system operates by recalculating controller

parameters. There are three major techniques for adaptive control [4]: gain scheduling, model-reference adaptive control, and self-tuning control. In each of these cases, adaptation is parametric (i.e. non-structural). In adaptive controllers, there are two, interlinked feedback loops: one is the usual feedback loop between the plant and the controller, while the other loop contains the adaptation component, which receives data from the plant, and configures the main controller by setting its parameters. Figure 1 illustrates the generic architecture of adaptive control: the adaptation mechanism can use measurements of the plant as well as the control signals generated by the regulators. The typical technique for achieving stability and convergence in the adaptive system is to use much larger time constants in the outer loop than in the inner loop, i.e. the adaptation process is much slower than the plant. Interestingly, it took a relatively long time to establish the mathematical framework for analyzing and proving the stability property of adaptive controllers. Adaptive control makes (at least) two important contributions to self-adaptive software: (1) the adaptation mechanism should be explicit and independent from the "main" processing taking place in the system, and (2) the overall system dynamics should be different for the adaptation mechanism and the main processing mechanism.

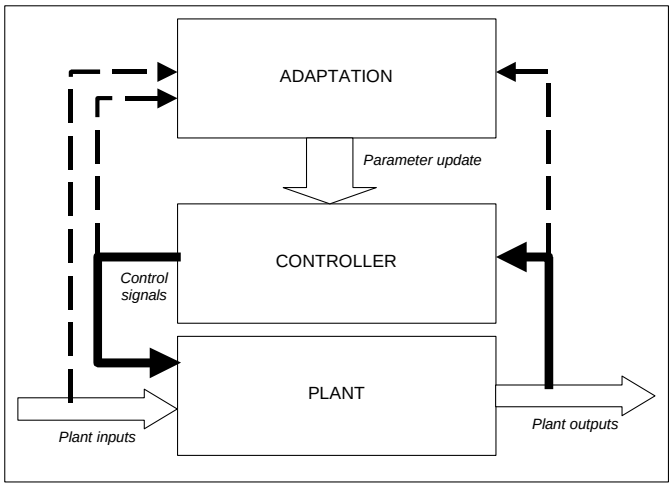

Figure 1: Adaptive control

While adaptive control introduces a second layer in control, supervisory and hierarchical control techniques [5] bring this concept to full implementation. For the sake of brevity, we will consider only supervisory control here (which we view as a special case of hierarchical control). In systems that use supervisory control, the control function is implemented in two, interdependent layers. The lower layer implements the regulatory function, and it typically contains simple regulators that keep process variables under control. The higher level, supervisory control layer is responsible for maintaining overall operational control, and it implements higher-level, goal-oriented, often discrete control behavior. Figure 2 illustrates the architecture of systems using supervisory control.

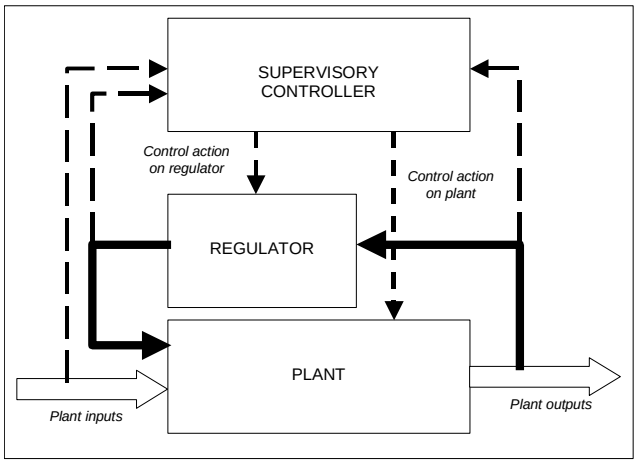

Figure 2: Supervisory control

The supervisory controller can interact both with the regulatory controller layer and the plant. When interacting with the regulator, it may change setpoints and gains, as it may reconfigure the structure of the controllers. When interacting with the plant, it may perform reconfiguration and/or execute other discrete control actions. Note that the regulators must have their own thread of execution (as they are operating directly on the sampled data), while the supervisory logic must also have its own thread (as complex decision making can —and should— be rarely performed between two samples). The supervisory control approach contributes the following concept to self-adaptive software: higher-level, goal-oriented decision-making that leads to reconfiguration done in a separate layer and thread.

Note that both the adaptive and supervisory control approaches are clear examples of a powerful technique used in (software) engineering: the separation of concerns [6]. The technique simply states that one shall place different concerns into different components, address the concerns independently, and rely on some "generic interface" to combine the pieces. In the controllers above, that primary regulatory function happens on the lower level, while the higher level addresses the issues of optimization, discrete decision-making, and fault accommodation. This "componentization" of concerns will serve well in self-adaptive software as well.

Obviously, controllers using adaptation or supervisory layers are naturally suitable for implementing systems that exhibit adaptivity. However, in typical situations both the adaptive and supervisory mechanisms are designed to follow some principles and observe constraints. For adaptive control, the designer performs careful analysis and puts limits on the adaptation (e.g. limits on controller gains) to avoid undesired evolution in the adaptive system. For supervisory control, the designer "maps out" the entire discrete state-space of the supervisory controller. Furthermore, if the supervisory component is used to mitigate the effects of anticipated faults, all behaviors are (or must be) very carefully analyzed. This makes the design and implementation of these sophisticated controllers rather difficult. However, the potential advantages of a well-separated, higher-level layer in implementing controllers arguably outweigh the costs in most applications.

Supervisory Control for Self-adaptive Software

The principles and techniques invented in supervisory control offer a natural architecture for implementing self-adaptive software systems. Figure 3 below illustrates this canonical, generic architecture.

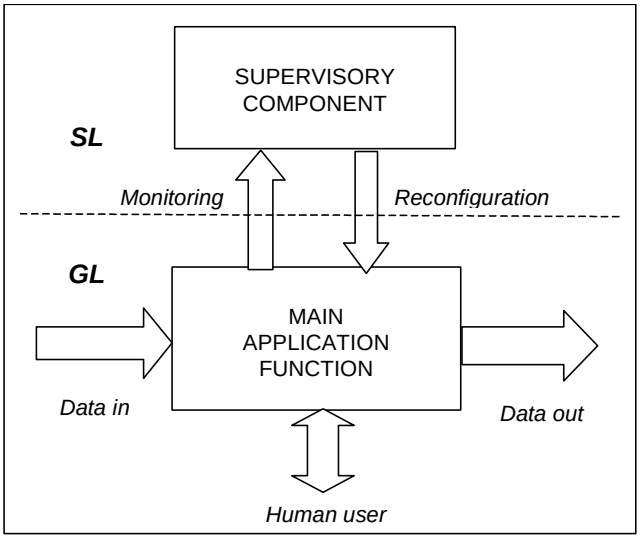

Figure 3: Self-adaptive architecture

The obvious conclusion is that one can build a self-adaptive software system using "ground-level" (GL) layer that includes baseline processing and using a "supervisory-level" (SL) layer that is responsible for the adaptation and reconfiguration. The two layers address separate concerns: one is the tuned for baseline functionality and performance, while the other one is tuned for optimization, robustness, and flexibility. On the ground-level one can create components that are highly optimized for specific situations, while the supervisory-level will have to recognize what situation the system is in, and select the most optimal component.

The introduction of SL solves the simultaneous requirements of robustness and flexibility as follows. The designer can naturally prepare and encode a large degree of flexibility in the system by using architectural templates. By “architectural template”, we mean a pattern, where alternative implementations are allowed for a particular function in the system. The designer may place all potential implementations into the GL, and select one of them for the initial state of the system. As the system’s lifetime progresses, the SL may decide that another implementation is needed, and may chose to reconfigure to another implementation. Note that the alternative component is already present in the system: it is merely dormant. The reconfiguration is a form of adaptation, where the running system, the GL, is adapted to a new situation. When it is performed, the system is switched to a new component, and it continues operating. Note that a capability is needed for the very flexible configuration of components and systems. In fact, during run-time we need to replace entire component sub-trees, and silently switch over the functionality to the new implementation. Note also that by separating the concern of configuration from the concern of functionality we can actually build a simpler and more compact system than if these two concerns were addressed in one layer only.

To address the issue of robustness the designer can follow well-established engineering techniques by introducing explicit fault accommodation logic in the SL. In current software, exception handling is often an afterthought, if it is done at all. By making exception handling explicit in the SL, and forcing the designer to explicitly address exceptions in the logic, one can prepare the system for various fault scenarios. Fault accommodation logic means that the SL is made sensitive to exceptions generated by the GL, and the designer has the means of taking supervisory actions to mitigate the effects of those faults. The SL may also incorporate diagnostics functions if the exceptions do not easily map to failure modes of components in the system. The main goal of diagnosis is fault isolation based on detected discrepancies, down to specific components and their failure modes. Run-time diagnosis of software faults is a somewhat novel area, but if self-adaptation is to be used for achieving robustness in systems, it has to be addressed. The supervisory actions for the fault-tolerance are similar to the ones used in addressing flexibility: components (and sub-trees of components) may have to be replaced.

A common problem in reconfigurable systems is the size of the configuration space of components. It is easy to see that if one is using a hierarchical structure, where on each level of the component tree multiple implementations are allowed, the size of the space of configurations grows very rapidly. Figure 4 illustrates the problem: if the components in the middle layer allow multiple implementations (3, 2 and 3, respectively) the simple diagram has 18 different configurations. In real-life systems, it is not unusual to have 4-5000 components [7]. The configuration space spanned by these components is obviously astronomical. It is not obvious how we can manage, let alone build systems with configurations of this size.

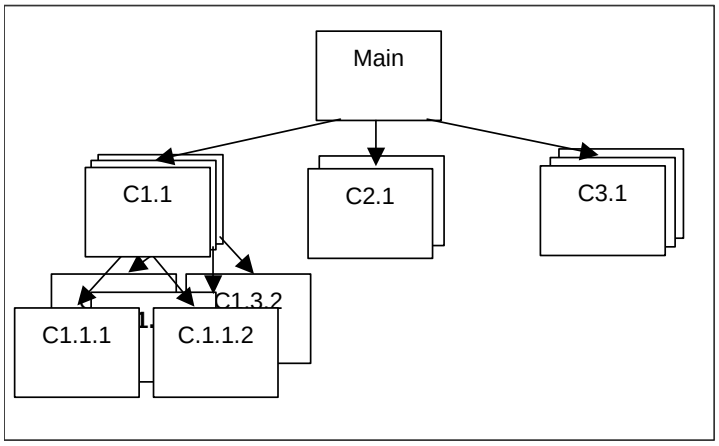

Figure 4: Configuration space

Note that we don’t have to store all the configurations in the system. We must store the components classes, such that we can quickly manufacture instances when needed, but only the active configuration is needed for functioning. The configuration space can be represented symbolically using a technique we describe later, such that when the SL decides on a new configuration it can quickly be instantiated.

Designing with the Supervisory Control Layer

Systems with a separate supervisory layer can be designed using well-established techniques, but the designer always have to consider the two-level nature of the architecture in these systems. For each class and component of the system, one has to answer the question: does this belong to the GL or to the SL? The purpose of GL is computation and functionality for the final application, so its capabilities and performance is the one that ultimately determines the success of the system. The purpose of SL is to provide flexibility and robustness, so its capabilities help ensuring the services provided by GL, but they alone will not directly implement application functionality. The designer also has to ask: is this a management function or is this an application function? The answer to this question will decide to what layer the function belongs.

When designing the SL, the designer has to anticipate "failures" in the GL. By "failure", we mean here both failures in the performance space and failures in the function space. Examples for failures in the performance space are: required accuracy in numerical computations was not achieved, or speed of computing results was not sufficient. Examples for failures in the function space are: the component has crashed, or the component has executed an illegal access. The SL has to have an ingredient: the "monitor" or "evaluator" that detects these failures and informs the supervisory logic about them. The monitor should have access to components in the GL, possibly all the data-streams connecting these components, and should be able to "tap into" the interaction patterns among components. Monitors may also be quite complex in order to detect performance degradation.

The GL/SL separation also imposes some design requirements on the components of the GL and run-time infrastructure. In order to monitor all interactions among components in the GL, all these interactions must happen through "channels" that can be accessed from the SL. Figure 5 illustrates the concept.

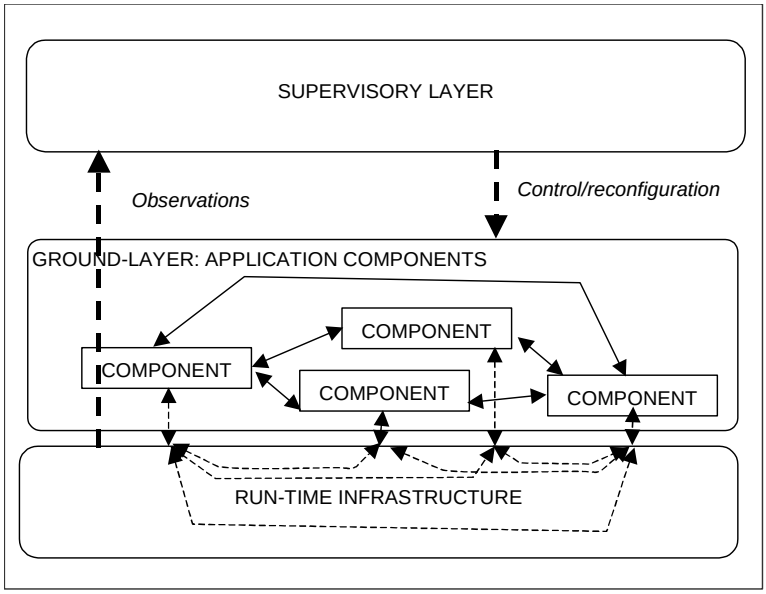

Figure 5: Integration of SL and GL via the run-time infrastructure

The GL component interactions should be implemented via the services provided by the run-time infrastructure (RTI), and the infrastructure must provide access to these interactions from the SL. On the diagram, the solid lines among the components denote the logical interactions among the components. In reality, the components interact with each other via the run-time infrastructure, as shown with the dashed lines. The SL should be able to access these data streams in order to support the monitoring function. Furthermore, the run-time infrastructure must be able to catch exceptions generated by components and provide them for the SL.

In order to facilitate GL component interactions via the run-time infrastructure, the components have to be implemented in a way such that it is not physically dependent on other components it interacts with. Direct component interactions, like object method calls are to be replaced with communication via the infrastructure. Direct or indirect references, like pointers or CORBA [8] IOR-s to other components are to be replaced by logical links that are mapped via the RTI. This requirement also implies that all interactions will have to be precisely documented and modeled, in order to verify the correctness of component configurations.

The supervisory logic implements adaptation as follows. When the monitoring subsystem detects a need for change, the SL will change the parameters or the structure of the GL. When the structure is modified, a set of components of the running system is replaced with other components. This brings up some interesting questions about the verification of structurally adaptive systems. If the component configurations are encoded as hierarchies, with multiple alternatives on each level (like Figure 4), how can we ensure overall consistency of the system under all potential configurations? If we reconfigure in one part of the system, how will this be made consistent with configuration decisions in other regions? As it was discussed above, the configuration space is potentially huge, and the designer needs to have the capability to verify configurations, or, at least, specify undesirable configurations that the system must never reach. For high-confidence systems, it is desirable to have tool support for exploring the configuration space at design time to verify assurance.

Modeling and analysis on the SVC layer

The approach described above easily maps into the use of high-level models in the design process [9], and the use of model-based generation wherever feasible. The modeling must happen on two levels: on the GL, where components and component configurations are represented, and on the SL, where the supervisory control logic is captured.

The modeling on the GL has to offer capabilities similar to those usually available in modern CASE environments, like Rational Rose [10]. However, UML in itself is not sufficient. On the GL, one has to create architecture models that capture component instances, their properties and their interactions. UML supports modeling of software artifacts in the form of class diagrams, interaction diagrams, and others, but seems to lack sophisticated facilities for modeling architectures. Another issue is that modeling on the GL has to happen in conjunction with a component integration infrastructure. If, for instance, CORBA [8] is used as the run-time infrastructure, all component integration has to happen via the object broker, and the designer must explicitly be made aware of this fact. If components have internal structure, which they do not expose to the system level, intra-component communication does not need the services of the RTI. Obviously, this type of communication can be made very efficient. The two interaction types should be clearly distinguished in the design.

The modeling on the SL has to capture the supervisory logic of the self-adaptive application. One natural, well-known, and powerful way of capturing supervisory logic is to use the Statecharts notation [11]. Supervisory logic involves mostly discrete decision making, and the hierarchical parallel finite-state machine (HFSM) approach of Statecharts offers a natural way to capture this logic. Better still, a number of modeling tools are available. A supervisory layer can be implemented as shown on Figure 6.

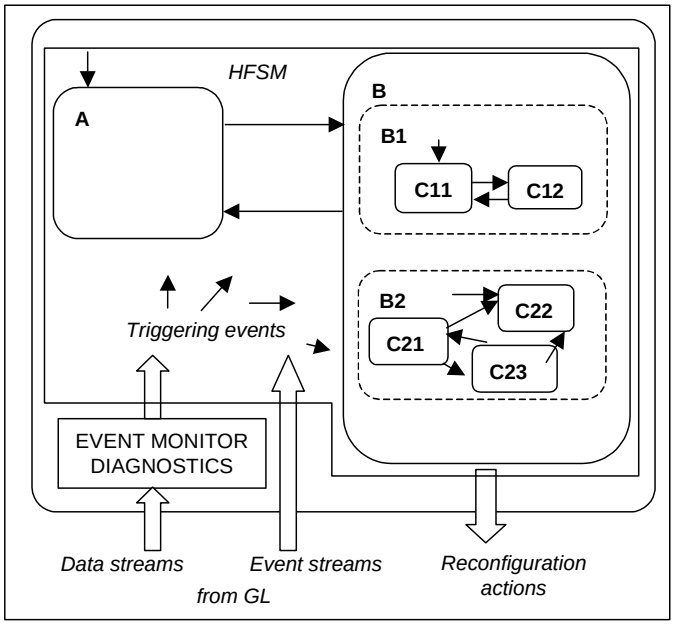

Figure 6: Supervisory controller

The figure also shows an example HFSM. It has two, OR-states: A and B, A being the initial state. B is decomposed into two AND-states: B1 and B2. B1 contains two OR-states: C11 and C12, C11 being the initial state. B2 contains three states: C21, C22, and C23, with C22 as the initial state.
In the modeling language of Statecharts, the basic building block is a state, which may contain other states, called sub-states. The sub-states of a state can be of type AND-state or OR-state. The sub-states of a parent state must be homogeneous: all of the sub-states must be of the same type. From among the OR-states precisely one can be active at any time (i.e. the FSM is "in" exactly *one* of those states). On the other hand, the FSM is "in" *all* of the AND type sub-states of the parent state, when the parent state is active. This latter situation is achieved by having multiple, concurrent FSM-s being active simultaneously.
A HFSM consist of states and transitions. Events trigger transitions among states, and the transitions can be enabled by conditions called *guards*. If a guard enables a transition, and a corresponding event is active, then the transition is taken. During the transition, the HFSM can execute an *action.* A triggering event is either generated by the event monitor, or it can be generated by a time-related function. Two examples for time-related functions are as follows:

- `after(Event,TimeValue)`
- `every(TimeValue)`

The first variant represents a triggering event, which goes active `TimeValue` units after the `Event` went active. The second variant represents a "clock": a triggering event with a fixed period: `TimeValue`. This time-triggered behavior allows initiating adaptation synchronized to time.
The supervisory layer can trigger reconfiguration actions, but it can also make parametric changes in the components of the GL. Both of these changes are facilitated through action expressions. In the HFSM, each state transition has three associated expressions: the guard, event, and action expression. The event expression is mandatory, while both the guard and the action expressions are optional.
A guard expression is a Boolean valued formula, which enables the transition to happen. A guard expression can refer to variable values that are calculated in the GL, or to states, etc., as specified in the Statechart documentation. Event expressions refer to specific events and follow the Statechart conventions with the addition mentioned above: event expressions can contain the `after(E,T)` and `every(T)` clauses as well.
There are two types of action expressions: (1) actions that influence the HFSM, and (2) actions that interact with the GL. The actions of the first type are similar to the ones available in Statecharts. Actions that modify the GL values can be, for instance:

- `set(Component, Attribute, Value)`
  Set the value of an attribute of a component in the GL.
- `send(Component, Message)`
  Send a message to a component in the GL.

The following actions are related to reconfiguration performed on the GL.

- `configure(Component, Configuration)`
  Select a configuration for a component. This action can be attached only to a state, and never to a transition. Strictly speaking, it is not an action, rather an assertion, which declares that when the state is active, a particular configuration of components is active.
- `select(Component, SelectorProcedure)`
  Select a configuration for a compound via a selector procedure. This action can be attached only to a state, and never to a transition. Strictly speaking, it is not an action, rather an assertion. It declares that when the state is entered, the selector procedure is invoked which will choose a particular configuration for the component.
- `construct(Component, ConstructorProcedure)`
  Generate a configuration for a GL component via a constructor procedure. This action can be attached only to a state, and never to a transition. Strictly speaking, it is not an action, rather an assertion. It declares that when the state is entered, the constructor procedure is invoked which will dynamically generate a particular configuration for the component.
- `strategy(Strategy)`
  Select a reconfiguration strategy. This action can be attached only to a transition, and never to a state. Strictly speaking, it is not an action, rather an assertion, which declares that when the transition is executed, a particular reconfiguration strategy is to be used.

The supervisory layer supports the reconfiguration on the component layer as follows. The reconfiguration is broken down into three phases: (1) determining the new component architecture, (2) calculating the parameters of

the new architecture (if needed), and (3) switching from the currently active component architecture to the new architecture. There are three cases for determining the new component architecture.

1. The designer supplies component architecture alternatives for each situation and the supervisory logic simply selects from them based on input data. The input data may be data from the running components, measured performance data, fault information, etc.
2. The designer supplies controller alternatives and a *selector procedure*, which, possibly via complex calculations, determines which alternative to choose given input data.
3. The designer supplies a *construction procedure,* which, given input data, will dynamically calculate the topology of the new controller.

In the first two cases, the designer must supply component configurations, via architectural templates. An architectural template enumerates a set of structural alternatives.

The models for the supervisory layer capture these reconfiguration activities and options as follows. In each application that needs reconfiguration, the designer should build separate, parallel HFSM-s describing the reconfiguration logic. In the states of these HFSM-s, the designer can introduce `configure()` actions to declare what configuration is active in that state. Alternatively, the designer can use `select()` or `construct()` actions to facilitate the selection or dynamic construction of a configuration. The `strategy()` actions can be attached to state transitions, and they indicate that when switching from the current configuration into a new one, what kind of reconfiguration strategy is to be invoked. Figure 7 illustrates the technique for modeling reconfiguration with a (H)FSM. The example shows two states that use different alternatives (`alt1` and `alt2`) for a component (`c1`), and use two different strategies depending on the switching direction (`X` and `Y`).

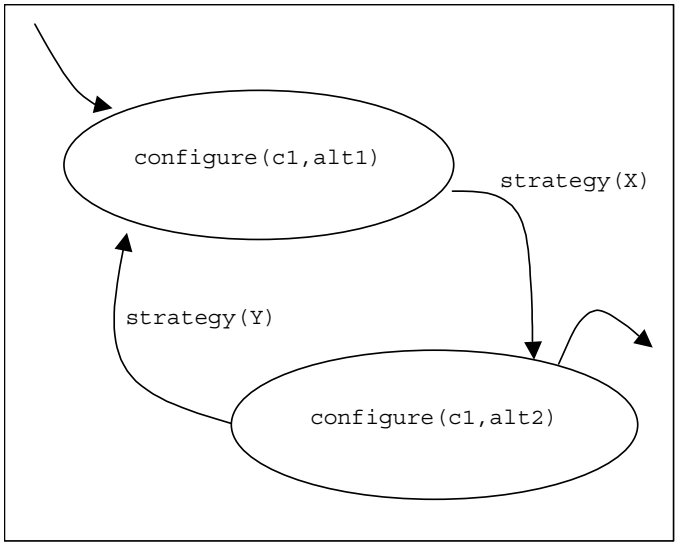

Figure 7:Modeling reconfiguration

Naturally, using the approach of `configure()` is the simplest: the designer specifically selects the new architecture. The `select()` approach is somewhat more sophisticated: the designer can supply a complex decision making procedure that may perform some sophisticated reasoning to come up with the "best" new architectural alternative. The input to this reasoning process can be arbitrarily complex (e.g. performance data about the running system or fault data), but the selection is still made from a finite, pre-specified set of alternatives. The most sophisticated (and difficult) option is when the `construct()` action is applied: the designer has to supply a generative model [12] that, when executed, will synthesize the new architecture on the fly.

The supervisory layer can also be interfaced with a fault diagnostics system. A fault diagnostic system can be considered as a sophisticated event monitor that not only detects fault events in the GL, but also maps those events to specific components and their failure modes[1]. This interface is necessary for being able to model and implement *fault accommodation logic* with the supervisory layer. Imagine a scenario where the GL develops a fault, for example a component is generating bad data. The designer of the supervisory layer anticipates this

[1] We borrow here some terminology from the language of fault diagnostics of physical systems: by failure mode we mean a particular way a component fails and exhibits faulty behavior [13].

situation, and wants to prepare the supervisory layer to manage the fault scenario by introducing changes in the GL. This portion of the supervisory control logic is called fault accommodation logic.
The interface between the fault diagnostics and supervisory layer is unidirectional: the fault diagnostics system supplies data and triggering events to the supervisory controller. The output of the diagnostic system is a *diagnostic event*. There are two types of diagnostic events:

- Fault detection events (that signal the presence of faults)
- Fault source identification events (that indicate the location of the fault)

Fault detection events (FDE) indicate a problem with a specific variable: for instance, a data stream in the GL. Usually, an FDE is triggered before the detailed diagnostics, and can be used as an early indicator of an incipient (or already existing) fault.
Fault source identification events (FSIE) indicate problems with specific components in the GL. They can be associated with requirement violations by components, or actual failure modes. An FSIE is activated after the diagnostics has reached the conclusion and identified the source of a fault. The source can be scoped to a component's failure mode or just to a component, or both. Thus, the FSIE activation is an indication of an existing fault.
The resulting events can be used in the HFSM to trigger transitions (thus initiate reconfiguration), but event information may also be passed into the decision-making procedures used in the `select()` and `construct()` actions. This gives rise to the opportunity to directly base the details of the reconfiguration decision on the circumstances that triggered the reconfiguration (i.e. the fault).
The `strategy()` action mentioned above has a very significant task: it provides a procedural description for the reconfiguration. It is very rare when software components can be simply removed and inserted into a running system. For instance, the procedure for reconfiguration may require the instantiation of the new architectural component, its initialization with data from the old component, gradually switching over to the new one, and garbage collecting the old one. The `strategy()` specification allows the designer to select the most appropriate script for reconfiguration.
Just like in adaptive control systems, the question of stability is of great importance for the designer of self-adaptive software. While there is no formal definition for "stability" for self-adaptive systems, as a working model one can use the following: "a system is stable if the frequency of reconfiguration actions is significantly lower than the frequency of GL component interactions". This admittedly imprecise and vague definition tells us to avoid situations when the system spends most of its time in frequent switching between configurations. Further research is needed to precisely define what stability means for a structurally adaptive software system, and how to verify from the formal model of the supervisory controller.
Modeling both offers and necessitates the verification of supervisory controller for a self-adaptive system. Note that reconfiguration actions refer to specific components, and components may be nested hierarchically. Thus, if a reconfiguration action selects one configuration for a higher-level component, another, parallel FSM may select a completely inconsistent configuration for an embedded component. To avoid this situation, sophisticated model checking has to be performed on the HFSM to avoid contradictory actions on the GL.
In general, the verification and assurance of self-adaptive software is a complex problem. However, with the use of explicit models for the reconfiguration logic we can probably cast the problem in a model-checking framework, for which techniques are available.

Implementation

The approach presented above has relevant implications for the implementor who wants to build self-adaptive systems using it. Arguably, the following four issues need to be addressed by the implementation:
Expectations for components: Components should be developed with some degree of independence, i.e. they must not depend on the precise implementation features (only the interfaces) of other components. If the self-adaptive software is for embedded, resource-constrained applications, we need a way (e.g. a language) for characterizing the components. There is also a need for expressing alternative component architectures in the form of architectural templates.
Expectations for the run-time infrastructure: The RTI must provide services not only for component communication and coordination, but also for dynamic reconfiguration and probing of component interactions from another layer. Current middleware packages (CORBA and COM [14]) have some support for the first two, but do not have adequate support for the rest. We need an industry standard (perhaps a CORBA service) for supporting these activities.

Expectations for the HFSM: Experience shows that the HFSM model for the supervisory controller can be compiled into executable code in a straightforward manner. The procedures involved in the `select()`, `construct()` and `strategy()` actions may be formulated in a procedural language, but it would be interesting to investigate how to provide a modeling language for describing these procedures on a higher, more abstract level.

Expectations for design space management: As it was shown above (Figure 4), the configuration space for design hierarchies with alternatives can be potentially enormous. Efficient techniques are needed for representing this configuration space, and for the rapid selection of desired alternatives. One approach that we have used in the past [16] seems particularly feasible: the hierarchical design is encoded as a tree, where each node is encoded using a binary vector containing bits of 1,0,X (unknown) values. This allows the use of Ordered Binary Decision Diagrams to represent and manage the design space symbolically. Architectural selections (to be used in the `select()` actions) can be performed symbolically, like the evaluation and search of a very large number of alternatives.

Summary, conclusions, and future work

We have shown a systematic, technical recipe for building self-adaptive software systems. The approach is based on the well-established techniques of control engineering, and model-integrated computing. We have shown how the two-layer approach that separates main application functionality from supervisory, adaptation-related activities can be used to architect the system. We have illustrated how HFSM-s can be used to model and implement reconfiguration logic, and how sophisticated decision logic can be incorporated into the design.

We are currently working on using the described approach to implement fault-adaptive controllers (FAC) [15]. A FAC is a controller architecture that is able to autonomously detect faults in physical plants, and reconfigure the regulatory controllers to maintain control in the face of failures. All the regulators, fault diagnostics, and reconfiguration logic is implemented in software. This problem domain is simpler than general-purpose self-adaptive applications, but it arguably offers an excellent opportunity to try out the ideas described above.

There are number of highly relevant research topics introduced by the proposed approach. Extending the formal modeling into the selection and construction procedures, verification and validation of reconfigurable systems for all possible configurations, and establishing some degree of assurance for the final systems' properties like schedulability and stability, just to name a few. However, the benefits offered by self-adaptive systems —their inherent ability to address flexibility and robustness in the same design— may one day far outweigh their problems.

Acknowledgement

The DARPA/ITO SEC program (F33615-99-C-3611) has supported, in part, the activities described in this paper.

References

1. McLean, J and Heitmeyer, C. "High Assurance Computer Systems: A Research Agenda," America in the Age of Information, National Science and Technology Council Committee on Information and Communications Forum, Bethesda, 1995.
2. Kumpati S. Narendra, Anuradha M. Annaswamy: *Stable Adaptive Systems*, Prentice-Hall, 1988.
3. Karl Johan Åström and Björn Wittenmark: *Adaptive Control,* Addison-Wesley,1995.
4. Ken Dutton, William Barraclough, Steve Thompson, Bill Barraclough: *The Art of Control Engineering*, Addison-Wesley, 1997.
5. Karl J. Astrom (Editor): *Control of Complex Systems*, Springer, 2000.
6. http://trese.cs.utwente.nl/Workshops/OOPSLA2000/
7. Personal communication with engineers from a major aerospace manufacturer.
8. http://www.omg.org
9. Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," *IEEE Computer*, pp. 110-112, April, 1997.
10. http://www.rational.com
11. David Harel, Michal Politi: *Modeling Reactive Systems with Statecharts: The Statemate Approach*, McGraw-Hill, 1998.
12. Ledeczi A., Bakay A., Maroti M.: Model-Integrated Embedded Systems, in Robertson, Shrobe, Laddaga (eds) Self Adaptive Software, Springer-Verlag Lecture Notes in CS, #1936, February, 2001.

13. IEEE Std 1232-1995. Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture, New York, IEEE Press. 1995.
14. Don Box, Charlie Kindel, Grady Booch: *Essential COM,* Addison-Wesley, 1998.
15. http://www.isis.vanderbilt.edu/Projects/Fact/Fact.htm
16. Neema S.: Design Space Representation and Management for Model-Based Embedded System Synthesis, ISIS-01-203, February, 2003.