Active Software

Robert Laddaga

MIT AI Lab

Introduction

The chief problems that software engineering will face over at least the next two decades are increasing application complexity, and the need for autonomy and serious application robustness. In other words, how do we actually get to declare success when trying to build applications one or two orders of magnitude more complex than today's applications? And, having declared success when building, how do we keep them from absorbing all our productivity to keep them patched together and running with more than just a vague semblance of doing what we want? The sources of increasing application complexity are:

1. simple growth in problem size, as a result of success at previous problem sizes and increased hardware capacity;
2. the fact that many applications are now more closely tethered to the real world, actively utilizing sensors and actuators, or being required to respond in real-time;
3. the enmeshed nature of today's applications – such as enterprise requirements to link all the elements of business process where it is useful to do so.

There has always been a significant difficulty with software robustness, for which the chief causes were high part counts, insufficient natural constraints, and requirements of interpretation of results; all of which make software innately difficult to make robust. In addition to these problems, further causes of insufficient robustness in the future will be:

1. The increase in software complexity that we discussed above;
2. The difficulty in getting modules built by one individual or team to interface with others, not quite so organically connected during the development cycle; and
3. Requirements change - the fact that as we use software, we discover that it can "almost" do more than we originally thought, or that we didn't quite understand precisely what we wanted the software for.

These dramatic problems, new at least in scale for the upcoming decades, require changes in our approach to software engineering. The old approaches to these problems: object oriented programming, client-server architectures, middleware architectures, distributed computation, verification, and exhaustive or statistical testing, etc. are all clearly insufficient to the task. While significant improvements to all these technologies are possible and also highly desirable, those improvements themselves won't be sufficient to the task.

We are proposing a collection of novel approaches that are informed by a single theme: that software must take active responsibility for its own robustness, and the management of its own complexity.

Naturally software can only do this if the software designer is able to program the software to take on these meta-tasks, and to supply the program with appropriate meta information: designs, alternative computations, goals and world state. This can only be done when we supply the designer with new tools, architectures, and patterns. Among these technologies, I want to mention the following four: tolerant software; self-adaptive software; physically grounded software and negotiated coordination.

Problems with Software

There are three principal interrelated problems facing the software industry. These are escalating complexity of application functionality, insufficient robustness of applications, and the need for autonomy. These are the problems that must be addressed if we are to continue to meet the software challenges of the future. These problems are interrelated because in general, greater robustness is necessary but not sufficient to deal with more complex operations, and autonomy generally increases complexity, and requires great robustness. Before considering how best to deal with these problems, we first consider how they arise and interrelate.

Lack of inherent limits on software functionality, combined with current successful applications, creates a demand for ever increasing software functionality, and without significant additional effort, this increased functionality will come with increased complexity. The decreasing cost of computational hardware, as compared to mechanical hardware and personnel costs, and the non-recurring nature of software development expense, provide economic incentives for software to assume more functional burden, and hence more complexity. Networking and communication between processing nodes also increases software complexity.

A second, and significantly more important source of complexity is an increasing emphasis on embedded software. It has been true for some time that 98% of the processors produced each year are utilized in embedded applications. The vast majority of those applications use less sophisticated and powerful processors, but use of more powerful processors and memory in embedded applications is growing even more quickly than that of less powerful processors. The physical reality of sensors and effectors guarantees that there will be cross cutting constraints that complicate tasks, violate modularity boundaries, and in general, make software more difficult to write, read, and update. Most microcontrollers have been used in point solutions, controlling a single device or sensor. Networking of controllers, and sharing information and coordinating control of complex assemblies of sensors and effectors, is happening rapidly today. For example, today cars mostly use microcontrollers to deploy a single airbag, or modulate a single brake. Tomorrow's cars will provide generalized traction control, using engine, transmission, and breaks and generalized accident response, coordinating multiple airbags, engine response, and notification to emergency channels. This type of networked, coordinated control will bring significant additional complexity to embedded applications.

It is also the number, breadth and pervasiveness of embedded applications that drive the requirement of autonomy. As embedded applications deal with both more data, and more data sources, it becomes more difficult to count on human beings to directly manage systems. Even in situations where a human being will exercise significant control, it is essential for the embedded system to be capable of autonomous operation, in order for it to have sufficient capability of informing, supporting and guiding the human controller.

In assessing the degree of functional complexity, physically embedding, and autonomy we are likely to see in the near future, let's consider the emerging computational infrastructure in general. We will see variations of several orders of magnitude in communication bandwidth, further chaotic, emergent and unreliable behavior from networks, huge numbers, great heterogeneity and unreliability of host nodes, no possibility of centralized control, and serious concerns about deliberate attempts to compromise systems. Data sets and work loads can be expected to be extremely dynamic, and control functions will also be dynamic and situation dependent. We can also expect weak theories of system behavior, for perception and sensing in particular, and weak knowledge about the degree of trust we can have in operability and intentions of system components – hardware and software.

In the area of robustness, there is a significant irony. We mentioned above that the malleability and unlimited quality of software contribute to software taking on more of the functional burden in systems. Despite the enormous malleability of software *itself*, software *applications* are incredibly brittle. They tend to work only in a narrow domain of utilization, and exclusively in a predetermined subset of that domain. Outside the intended use of its creators, software applications fail miserably. They also evidence lack of robustness over time. As functional and performance changes are required, software applications are difficult to maintain, difficult to pass from one programming team to another, and difficult to update, preserving correctness, performance, and maintainability.

The inherent complexity of software systems makes testing and proving correctness each suffer from combinatorial overload. Imagine the complexity of an automobile engine, if we built it atom by atom. Our "character set" would consist of a subset of the elements, and the number of possible combinations would be staggeringly huge. Despite larger numbers of such character-atoms, there is considerably less freedom in combining the mechanical atoms, than in combining characters into programs. The complexity induced conceptual overload means that testing and verification are even more limited than the domain of applicability, and increasing that domain exacerbates the testing and verification problems. In a sense, we are deliberately reducing robustness, in the form of ability to accommodate a wide range of domain conditions, in order to make the program more understandable. However, what we chiefly understand about such programs is that they are brittle and fragile.

Are current methods enough?

One might, however, stipulate that everything discussed above is true, but that current methods and technology of software development will continue to be adequate to the tasks ahead. We make the strong claim that this is not so. Instead we believe that two features create, exacerbate and interrelate the problems mentioned above: meeting the needs of embedded systems, and software development technology that provides insufficient support for modifiability of software applications. Further, the lack in current development technology to support both highly modifiable and physically grounded software systems, will ensure that the challenges indicated above will not be met by current technology. Physical interactions of embedded systems violate functional modularity, and hence make systems more complex, less robust, and more difficult to operate autonomously. Systems that are built using tools that are oriented to building things once, and right, make it difficult to add complex functionality incrementally, which in turn makes it difficult to build in sufficient robustness and autonomy.

To see why this is so, first consider how we currently manage complexity. The chief tool we have for dealing with computational complexity is an abstraction mechanism: functional decomposition. Functional decomposition includes breaking problems down into functional components, recursively. These components can be individually programmed, and then combined in relatively simple ways. The principle that makes functional decomposition generally work is that we can limit the complexity of interactions among functional units.

For non-embedded applications, it is relatively easy to limit interaction complexity to clean interfaces between functional modules. Even these non-embedded applications have real world context that adds interaction complexity, such as deadlines, and resource scarcity, such as memory, processors, communication bandwidth, OS process time slots, and other similar resources. We typically handle these issues via Operating System policies of sharing, priority and fairness, and by depending on the programmer to build in sufficient capacity.

With embedded applications, it is impossible to prevent pervasive intrusion of real world concerns. This intrusion increases complexity and results in tighter constraints on the computational resources, which we deal with by using explicit resource control, or using special real-time capable Operating Systems. In highly connected and decentralized embedded systems, higher level general resource managers may not always be a workable option. Even when there is some approach to dealing with resource allocation issues, the remaining interaction constraints are still the individual responsibility of the programmer, adding to his conceptual burden. Therefore, the cleanliness and value of functional decomposition is dramatically reduced. Also, resource conflicts are a chief source of interaction complexity themselves, which necessarily violates modularity based principally on functional decomposition. Since we generally handle resource management implicitly, we don't replace the violated functional modularity with any other form of modularity.

Even worse than the modularity violations, the implicit management of resources is handled statically, rather than dynamically. It is clearly advantageous to late bind the decisions about resource management and conflict resolution, since the exact nature of these conflicts is highly context dependent. In order to late bind resource conflict resolution policy implementations, we must explicitly represent and reason about such resource management issues.

There are some excellent evolutionary programming methodologies, that make it relatively easy to modify software over time. We might speculate that if we simply shifted to more use of those technologies, we could converge to sufficiently robust applications. However we modify our software, returning the software/system to the shop to redesign, reimplement and retest involves a very large response time delay. That delay is often critical, and in general it means that we run software "open loop". Of course, it is much more desirable to run software "closed loop", in which evaluation and adjustment happens in real time.

Software is the most malleable part of most systems that we build, but the software applications themselves, once built, are remarkably rigid and fragile. They only work for the set of originally envisioned cases, which is often only a proper subset of the cases we need to have the software handle. Traditional approaches to enhancing the robustness of software have depended on making all the abstractions (requirements, specifications, designs, and interfaces) more mathematically precise. Given such precision we could attempt to prove assertions about the attributes of the resulting artifacts. Then, whether precisely specified or not, the software systems would be extensively tested. For complex software artifacts, operating in complex environments, extensive testing is never exhaustive, leaving room for failure, and even catastrophic failure. Also, often, the precision of specification works against producing systems that reliably provide service which good enough, delivered soon enough. However, there is significant distinction between engineered tolerance and simple sloppiness. Much software is today is built on the assumption that software will have bugs and that systems will simply crash for that reason. The further assumption is that regardless of how much testing is done "in the factory", the bulk of the "testing" will happen while the software is in service, and all repairs will happen "in the shop". We mean something very different by software tolerance.

A New Approach to Software

We have claimed that problems with future software development efforts center around handling increased functional complexity, providing substantially increased robustness, and providing autonomy. These problems are all exacerbated by increasing emphasis on embedded systems, and by a "do it once, right" engineering approach. What we are proposing is a radically new form of abstraction, that we call *Active Software.* The overarching themes of the Active Software approach are that ***software must take active responsibility for its own robustness, and the management of its own complexity***; and that to do so***, software must incorporate representations of its goals, methods, alternatives, and environment.***

Research can provide us with software systems are active in several senses. They will be composed of modules that are nomadic, going where network connectivity permits. They will be able to register their capabilities, announce their needs for additional, perhaps temporary capability, evaluate their own progress, and find and install improvements and corrections as needed and as available anywhere in the network. They will manage their interfaces with other modules, by negotiation and by implementing flexible interfaces. Thus active software is nomadic, self-regulating, self-correcting, and self aware. In order to accomplish these purposes, the underlying environments and system software must support these capabilities.

We do not as yet have a clear and unambiguous understanding of this new approach. What we do have is a collection of technologies that can be viewed as component technologies under Active Software. By discussing and comparing the component technologies, we will attempt to get a clearer picture of the Active Software concept. In fact, the concept of Active Software derives from some common aspects of the component technologies.

The component technologies are self adaptive software, negotiated coordination, tolerant software and physically grounded software. Self adaptive software evaluates its behavior and environment against its goals, and revises behavior in response to the evaluation. Negotiated coordination is the coordination of independent software entities via mutual rational agreement on exchange conditions. Tolerant software is software that can accommodate any alternatives within a given region. Physically grounded software is software that takes explicit account of spatio-temporal coordinates and other physical factors in the context of embedded systems.

The four listed technologies are not a partition of the concept of Active Software. It would probably be useful to have a partition of the concept, but the current four components are certainly not exclusive, and they are probably not exhaustive either. For example, negotiation of interfaces can provide tolerant interfaces, but there are certainly other ways to get tolerance. Also, physically grounded software is necessary for self adaptive embedded systems, but physically grounded software need not be self adaptive. Neither is active software the only new approach to building software able to address the challenges we have discussed. Aspect oriented programming is an extremely interesting alternative, and complementary approach [7].

Self Adaptive Software

Software design consists in large part in analyzing the cases the software will be presented with, and ensuring that requirements are met for those cases. It is always difficult to get good coverage of cases, and impossible to assure that coverage is complete. If program behaviors are determined in advance, the exact runtime inputs and conditions are not used in deciding what the software will do. The state of the art in software development is to adapt to new conditions via off-line maintenance. The required human intervention delays change. The premise of **self adaptive software** is that the need for change should be detected, and the required change effected, *while the program is running* (at run-time).

The goal of self adaptive software is the creation of technology to enable programs to understand, monitor and modify themselves. Self adaptive software understands: *what it does; how it does it; how to evaluate its own performance; and thus how to respond to changing conditions*. We believe that self adaptive software will identify, promote and evaluate new models of code design and run-time support. These new models will allow software to modify its own behavior in order to adapt, at runtime, when exact conditions and inputs are known, to discovered changes in requirements, inputs, and internal and external conditions.

A definition of self adaptive software was provided in a DARPA Broad Agency Announcement on Self-adaptive Software [2]:

> Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.
>
> …This implies that the software has multiple ways of accomplishing its purpose, and has enough knowledge of its construction to make effective changes at runtime. Such software should include functionality for evaluating its behavior and performance, as well as the ability to replan and reconfigure its operations in order to improve its operation. Self adaptive software should also include a set of components for each major function, along with descriptions of the components, so that components of systems can be selected and scheduled at runtime, in response to the evaluators. It also requires the ability to impedance match input/output of sequenced components, and the ability to generate some of this code from specifications. In addition, DARPA seek this new basis of adaptation to be applied at runtime, as opposed to development/design time, or as a maintenance activity.

Self adaptive software constantly evaluates its own performance, and when that performance is below criteria, changes its behavior. To accomplish this, the runtime code includes the following things not currently included in shipped software:

1. descriptions of software intentions (i.e. goals and designs) and of program structure;
2. a collection of alternative implementations and algorithms (sometimes called a reuse asset base).

Three metaphors have been useful to early researchers on self adaptive software: coding an application as a dynamic planning system, or coding an application as a control system, or coding a self aware system, [9]. The first two are operational metaphors, and the third deals with the information content and operational data of the program.

In programming as planning, the application doesn't simply execute specific algorithms, but instead plans its actions. That plan is available for inspection, evaluation, and modification. Replanning occurs at runtime in response to a negative evaluation of the effectiveness of the plan, or its execution. The plan treats computational resources such as hardware, communication capacity, and code objects (components) as resources that the plan can schedule and configure. See [1], [6], [11], and [16].

In program as control system, the runtime software behaves like a factory, with inputs and outputs, and a monitoring and control unit that manages the factory. Evaluation, measurement and control systems are layered on top of the application, and manage reconfiguration of the system. Explicit models of the operation, purpose and structure of the application regulate the system's behavior. This approach is more complex than most control systems, because the effects of small changes are highly variable, and because complex filtering and diagnosis of results is required, before they can serve as feedback or feed-forward mechanisms. Despite the difficulties of applying control theory to such highly non-linear systems, there are valuable insights to be drawn from control theory, and also hybrid control theory, including for example the concept of stability. See [6], [8], and [13].

The key factor in a self aware program is having a self-modeling approach. Evaluation, revision and reconfiguration are driven by models of the operation of the software that are themselves contained in the running software. Essentially, the applications are built to contain knowledge of their operation, and they use that knowledge to evaluate performance, to reconfigure and to adapt to changing circumstances (see [9], [17], and [14]). The representation and meta-operation issues make this approach to software engineering also intriguing as an approach to creation of artificial intelligence.

Problems, Future Issues:

The hardest and most important problem for self adaptive software is evaluation. There is great variability in the ease with which evaluation can be accomplished. The hardest case, where we have an analytic problem without ground truth available, may require us to use probabilistic reasoning to evaluate performance at runtime. There is still much work to be done in determining what classes of application require what forms of evaluation, which tools will provide better evaluation capability, and to what extent such tools will need to be specific to particular application domains.

Because self-adaptive software is software that adapts and reconfigures at runtime., there are crucial issues of dynamism and software architecture representation to be addressed in building development tools. Any piece of software that is revised on the basis of problems found in the field can be thought of as adaptive, but with human intervention and very high latency. Thus, we can think of these problems as generally when and how we implement the functions of evaluating program performance, diagnosing program problems and opportunities, revising the program in response to problems/opportunities, and assuring that desirable program properties are preserved, and who implements these functions. The single large step of changing the binding time for these functions can provide a tremendous amount of help with these problems. If we evaluate and diagnose performance immediately, at run time, and if further we can revise the program in response to diagnostics, then we can build a significantly adaptive functionality. In such a system, immediate evaluation, diagnosis and repair can play an enormous role in testing behavior and assuring the preservation of desirable features. It is much easier to introduce this kind of late binding in dynamic languages and environments that already support significant late binding. Similar concerns relate to explicit rendering of software structure or architecture: reconfiguration causes one to pay close attention to modularity, and to the unit of code that is considered configurable or replaceable. However, current dynamic languages aren't sufficient to the task, without the following improvements:

1. More introspective languages, and better understanding of how to use introspection;
2. Improving the debug-ability of embedded, very high level, domain specific languages;
3. Better process description – including better event handling and description, and explicit consideration of real time;
4. Better structural descriptions – inclusion of architecture description languages in programming languages (see [3], [4], [10], [11], and [12].

We will also have problems with runtime performance. It takes time to evaluate outcomes of computations and determine if expectations are being met. Most often, expectations will be met, and checking seems like pure overhead in those cases. On the other hand, comprehensively evaluating what algorithms and implementations will be used is an advantage if it means that we can select the optimal or near optimal algorithm for the input and state context we have at runtime, rather than a preselected design time compromise. Additionally, hardware performance keeps increasing, but perceived software robustness does not. Even so, effort will need to go into finding ways to optimize evaluation cycles, and develop hierarchical systems with escalating amounts of evaluation as needed for a particular problem.

Performance is also a problem at software creation time. The kind of programming that delivers code able to evaluate and reconfigure itself is difficult to build by hand. We would therefore prefer automated code generation technology. Specifically, we like approaches that generate evaluators from specifications of program requirements and detailed designs.

Advances in computer hardware provide challenging opportunities. Self adaptive software runtime support also exploits opportunities identified by evaluators and replanning operations to restore/improve functionality or performance. It is clear that reconfigurable software could provide a useful top level application programming level for reconfigurable hardware. In such a system, the hardware configuration would simply be another resource to be scheduled and planned. Another opportunity lies in helping to address the increasing mismatch between processor speed and memory access speeds. One of the approaches being taken is to move portions of processing out to memory modules, to reduce the requirement to move all data to main microprocessor. Self adaptive and reconfigurable software can play a role in dynamically determining the distribution of components in such highly parallel architectures.

Negotiation

One of the ancillary problems to be addressed in any software system, (and the primary problem of some systems), is resource allocation and resolution of resource contention. The principal tool humans use for these problems is negotiation, and active software aims at the same goal. The advantages of negotiation are that it is inherently distributed, inherently multi-dimensional, and robust against changes in circumstances. Although centralized, one dimensional (e.g. price) negotiation is possible, it is neither necessary nor natural. Negotiated settlements need not themselves be robust against changed circumstances, but then the answer is to simply renegotiate, so no special exception handling is required.

Commercial business processes, logistics systems, and DoD weapon and C3 systems, all have hierarchical control, gatekeeping barriers, and incur delays based on unnecessary human interaction. The hierarchical control induces a command fan-out and data fan-in problem that seriously limits the ability of such systems to scale with the problems they are designed to solve. In many cases we improve our ability to manage such systems by distributing computing resources, but without making the crucial application changes needed to make these applications genuinely distributed in nature. Organization with compartmented groups inhibits the flow of information and commands needed to rapidly respond to crises and an adversary's moves. In addition, passing data and commands across these organizational barriers includes further delay as items are added to input queues of human action processors.

Logistics provides paradigm examples of systems that cannot afford centralized control and decision making, nor afford artificial limits on scalability. Weapon systems that must respond to attacks automatically, such as Aegis or Electronic Countermeasures (ECM), are extreme examples that add a requirement of hard real-time response. What is needed are locally competent and efficient mechanisms for assessing situations and taking actions, which percolate information and intelligence up to higher levels, I.e. a *bottom-up* approach to system operation. The bottom-up approach emphasizes peer-to-peer communication, and *negotiation* as a technique for resolving ambiguities and conflicts. This entails a fundamentally new approach to how we build the complex systems of the future, (see [21]).

The new approach promises multiple benefits:

- Near linear scalable systems
- Highly modular, recombinant systems with distributed problem solving
- Removal of middle-man organizations in logistics and planning

The goal of Negotiation Software Agents (NSA) is to autonomously negotiate the assignment and customization of dynamic resources to dynamic tasks. To do this we must enable designers to build systems that operate effectively in highly decentralized environments, making maximum use of local information, providing solutions that are both good enough, and soon enough. These systems will have components that communicate effectively with local peers, and also with information and command concentrators at higher levels of situation abstraction. They will explicitly represent goals, values, and assessments of likelihood and assurance, and reason about those quantities and qualities in order to accomplish their task.

Negotiation systems will scale to much larger problem sizes by making maximum use of localized, rather than global information, and by explicitly making decision theoretic trade-offs with explicit time-bounds on calculation of actions. This new technology will enable us to build systems that are designed to utilize, *at the application level*, all the distributed, networked computational resources (hardware, operating systems, and communication) that have been developed over the past two decades. These advantages will be especially effective in cooperative action, since the process of matching tasks and resources is naturally decentralized in the NSA approach.

The first NSA task is development of the basic framework for bottom-up organization. This includes three subtasks, the first of which is discovery of peer NSAs, and their capabilities, tasks and roles. Second, NSAs must be able to determine how to get access to data, information and authorizations, and they must have secure authorization procedures. Finally, there must also be processes for coordination and information sharing at levels above the peer NSAs.

The second NSA task is reasoning based negotiation. Its subtasks are: handling alternatives, tradeoffs, and uncertainty, including noting when new circumstances conflict with current plans; enabling speedy, optimized and time-bounded reasoning processes to allow for real-time response; and procedures for assuring that goals are met.

A promising approach to reasoning under uncertainty is to use explicit decision theoretic methods, and develop heuristics and approximate decision theoretic approaches. Use of market based trading ([18], [19] and [20]), and qualitative probabilities are alternative approximate approaches. Satisficing and anytime algorithms will also be useful.

Tolerant Software and Interfaces

As an alternative to the approach of increasing precision to make applications "bullet-proof", we envision that reliability will result from an ability to tolerate non-critical variations from nominal specification. The concept of software tolerance derives from a consideration of the concept of mechanically interchangeable parts, and the role they played in the industrial revolution. In order for parts to be interchangeable (and hence significantly speedup the assembly of mechanical devices) there had to be a match between the ability to produce similar parts, and the ability of the overall design to accommodate whatever differences remained among the similar parts. Software systems today have highly intolerant designs, requiring components that exactly match rigid requirements. While this mechanical view of tolerance is useful, and has a direct analog in the area of component technology and software reuse, there are other significant aspects to tolerance. For example, we often don't need mathematically exact answers to questions for which such answers are possible. We often don't need exhaustive searches when those are conceptually possible but impractical. Tolerance involves a critical examination of acceptable variation, and engineering methods and technology in support of building systems with appropriate tolerance.

Tolerant software covers three related concepts:

1. Applications that don't have to be 100% correct – these include algorithms that are inherently probabilistic, applications where the answer is always checked by humans, queries that are inherently inexact (e.g. "Show me a selection of restaurants in Oxford.")
2. Lower level software that is checked by higher level applications and system software – examples are disk controllers, memory subsystems, UDP network protocol (checked by TCP layer or application software).

3. Analogy of mechanical tolerance, and interchangeable parts – this in particular includes software interfaces, word processor input formats, and middleware (RPC,RMI,CORBA [15]) information exchange, for example.

Other examples include:

- a web search engine, in which 24 by 7 availability is more important than being able to access the full database on every query;
- mailer software advertising a conference, or new product - not every potential recipient needs to receive it;
- object oriented programming systems, which often have highly specific code dispatched for arguments of specific types, and a more general (and less optimal) code for dealing with the remainder of cases.

Tolerant interfaces

We said the concept of interface tolerance derives from a consideration of the concept of mechanically interchangeable parts, and the role they played in the industrial revolution. The place where this analogy comes closest in software systems is in the interfaces between software modules or components, which play the role of mechanical parts. Unlike the case of mechanical tolerances, based on standards and static design principles, tolerant interfaces are active interfaces. Tolerant interfaces know how they interface with other modules, know or can find out how other modules handle their own side of interfacing, and can modify their own interface behavior to meet the requirements of interfacing with other modules. Thus tolerant interfaces can actively modify their interface behavior to meet interface requirements. They may also need to be able to negotiate interface responsibilities with other modules, in much the same way that modems today negotiate over the protocol they will use during data communication.

Clearly, to have these capabilities, the code will need to contain and operate on explicit descriptions of intent, specifications, design, program structure and mapping of tasks to functions. Also, the code will need to contain a large number of alternative interface implementations, or have access to such alternatives over network connections.

Another important concept for tolerant interfaces is that of wrapped data (objects, agents or smart data). Consider, for example, self-extracting compressed archives of programs. The file contains the software needed to decompress, and install the program at your site. Even now such installation programs accept some parameters from the user, but more could be done in this regard. This type of approach is useful in situations where you have data exchange where provider and consumer share responsibility or capability to operate on the data.

Imagine a sensor-actuator application, in which image data is gathered by a sensor, and delivered to a consumer that controls actuators. The sensor module sends data wrapped/prefixed with code for interpretation of the data. The consumer needs to change some operators or filters, in the context of the sensor's interpretation code.

The sensor and consumer code cooperatively arrange interfaces and a configuration suitable for the appropriate sharing of processing tasks and algorithms.

Tolerant software requires research on:

- Automated analysis and implementation of interface requirements
- data wrapping/prefixing
- calculation and mitigation of risks and consequences of software breakdown
- assurance/validity checking for tolerant combinations
- performance optimization

We mentioned earlier that the relationships of these components of active software are not simple. Negotiation technology can clearly be used to help build active tolerant interfaces, and tolerant interfaces are necessary for self adaptive software.

Physically grounded software

The most challenging software to write, debug, and maintain is software for embedded systems. Traditionally, we produce such software by writing relatively low level code, and counting cycles and memory usage in order to ensure real time requirements. Behavioral assurance is very hard to obtain, requiring combinations of difficult verification technology and exhausting (but not exhaustive) testing. This approach to embedded software will not continue to work as systems get more complex, more connected and require more functionality. In place of traditional approaches, we are advocating Physically Grounded Software (PGS). PGS is software that is thoroughly model based. It incorporates models of its environment, of its operational theory, of its behavior and of crucial physical quantities, including time, space, information and computational requirements. Those models will drive the design of systems, generation of code, refinement and maintenance of systems, and evaluation of behavior and behavioral guarantees.

We argue that computational power for embedded systems is becoming abundant. We should use this power to ease the design problem of embedded systems. This will require development of model driven simulations, for both design and runtime, development of language support for computations about time, space and other physical quantities, and special support for composition of such modules with behavioral guarantees, which we are calling a framework.

The age of shoehorning complex embedded systems into feeble processors is coming to an end. In avionics, for example, today's embedded software systems are the glue which integrate the entire platform, presenting high level situation awareness to the pilot. But since these systems must evolve constantly; our traditional approaches of entangling concerns in order to meet space and time constraints has led to unmaintainable software systems which are inconsistent with the goal of evolution.

Embedded systems that attempt to use software to integrate collections of systems, such as integrated avionics, require a model-based approach to be comprehensible, maintainable and evolvable. Model based approaches must further tie together standard operational control and failure recovery via models that relate layered systems dealing with behavior at different levels of generality and abstraction. In particular, an overall mission oriented model, with goals and plans is required to generate recovery actions. Subgoals and subtasks must recurse through the layers of the embedded systems to assist in tying the relevant behavior models together.

PGS is a prerequisite for successful autonomous systems. The goals of autonomous systems are reliable, safe, and cooperative operation of free-ranging systems in the world. There are numerous examples of autonomous systems: agent-based software that ranges over cyberspace performing information services including retrieval and delivery of information [5]; autonomous air, land, or sea vehicles; autonomous mobile robots in physical proximity to humans; automated plants and factories; and automated maintenance and repair systems. Such systems need to cooperate with each other and with humans.

Autonomous systems must also be reliable and robust enough to serve in real world, changing environments. A gross metric for the safety and reliability of such systems is that they be safe and reliable enough to actually be deployed and used where unsafe, unreliable operation would cause physical or economic harm. Reliability implies an element of goal driven operation, which requires the ability to dynamically structure and utilize resources to achieve a desired external state change, and the ability to handle uncertainty, and unpredicted situations. Robustness means that a desired behavior or functionality is maintained under changing external and internal conditions. Reliability and robustness assumes dynamic internal organization, integrated self-monitoring, diagnostics and control, and fully integrated design of the physical processes and information processing that forms the autonomous system. An increasingly important requirement for autonomous systems is cooperation. Cooperation is an ability to participate in the dynamic distribution of goals (and tasks) and in the coordinated execution of required behaviors.

Embedded software should be developed against the backdrop of model based frameworks, each tailored to a single (or a small set of) issues. A framework, as we use the term, includes:

- a set of properties with which the framework is concerned,
- a formal ontology of the domain,
- an axiomatization of the core domain theory,
- analytic (and/or proof) techniques tailored to these properties and their domain theory,
- a runtime infrastructure providing a rich set of services, and
- an embedded language for describing how a specific application couples into the framework.

A framework thus reifies a model in code, API, and in the constraints and guarantees the model provides. Frameworks should be developed with a principled relation to the underlying model, and preferably generated from a specification of the model. The specification of the model should be expressed in terms of an embedded language that captures the terms and concepts used by application domain experts.

The ontology of each framework constitutes a component of a semantically rich meta-language in which to state annotations of the program (e.g. declarations, assertions, requirements). Such annotations inform program analysis. They also facilitate the writing of high level generators which produce the actual wrapper methods constituting the code of a particular aspect.

Corresponding to each framework are analytic (or proof) tools that can be used to examine the code which couples to this framework. Each such analytic framework can show that a set of properties in its area of concern is guaranteed to hold as long as the remaining code satisfies a set of constraints. The analysis of the overall behavior of the system is, therefore, decomposed.

References

[1] I. Ben-Shaul, A. Cohen, O. Holder, and B. Lavva, ``HADAS: A network-centric system for interoperability programming," International Journal of Cooperative Information Systems (IJCIS),vol. 6, no. 3&4, pp. 293--314, 1997.
[2] ``Self adaptive software," December, 1997. DARPA, BAA 98-12, Proposer Information Pamphlet, www.darpa.mil/ito/Solicitations/ PIP_9812.html.
[3]Garlan, David & Shaw, Mary. "An Introduction to Software Architecture," 1-39. Advances in Software Engineering and Knowledge Engineering Volume 2. New York, NY: World Scientific Press, 1993.
[4]Garlan, D.; Allen, R.; & Ockerbloom, J. "Exploiting Style in Architectural Design Environments." SIGSOFT Software Engineering Notes 19, 5 (December 1994): 175-188.
[5]Jennings, K. Sycara, and M. Wooldridge A Roadmap of Agent Research and Development. In Autonomous Agents and Multi-Agent Systems, Vol. 1, No. 1, July, 1998, pp. 7 - 38.
[6] G. Karsai and J. Sztipanovits. A model-based approch to self-adaptive software. IEEE Intelligent Systems,May/June 1999:46{53,1999.
[7]Kiczales G., Lamping J., Mendhekar A. et al. *Aspect-Oriented Programming.* In proc. European Conference on Object-Oriented Programming, Finland, 1997.
[8] M. M. Kokar, K. Baclawski, and Y. Eracar. Control theory-based foundations of self-controlling software . IEEE Intelligent Systems,May/June 1999:37{45,1999.
[9] R. Laddaga. Creating robust software through self-adaptation. IEEE Intelligent Systems, May/June 1999:26{29,1999.

[10]Luckham, David C., et al. "Specification and Analysis of System Architecture Using Rapide." IEEE Transactions on Software Engineering 21, 6 (April 1995): 336-355.
[11] Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems and Their Applications, vol. 14, no. 3, pp. 54-62 (May/June 1999).
[12]Nenad Medvidovic and Richard N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. IEE Proceedings Software Engineering, vol. 144, no. 5-6, pp. 237-248 (October-December 1997).
[13] D. J. Musliner, R. P.Goldman,M.J.Pelican, and K. D. Krebsbach, ``Self-Adaptive Software for Hard Real-Time Environments," IEEE Intelligent Systems,vol. 14, no. 4, pp. 23--29, July/August 1999.
[14] G. Nordstrom, J. Sztipanovits, G. Karsai, A. Ledeczi: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems, April, 1999.
[15] Object Management Group, The Common Object Request Broker: Architecture and Specification. Revision 2.2,February 1998. Available at: http://www.omg.org/corba/corbaiiop.html.
[16] P. Robertson and J. M. Brady. Adaptive image analysis for aerial surveillance. IEEE Intelligent Systems,May/June 1999:30{36,1999.
[17] J. Sztipanovits, G. Karsai: "Model-Integrated Computing", IEEE Computer, April, 1997
[18] CA Waldspurger, T Hogg, B Huberman, JO Kephart, and WS Stornetta: Spawn: A Distributed Computational Economy. IEEE Transactions on Software Engineering 18:103-117.
[19]Wellman, M: "Market-oriented programming: Some early lessons." In Market-Based Control: A Paradigm for Distributed Resource Allocation (S Clearwater, ed.). World Scientific, 1996.
[20] Wellman, M: A market-oriented programming environment and its application to distributed multicommodity flow problems. Journal of Artificial Intelligence Research, 1:1-23, 1993.
[21] ``Autonomous Negotiating Teams," October, 1998. DARPA, BAA 99-05, Proposer Information Pamphlet, www.darpa.mil/ito/Solicitations/ PIP_9905.html.