Dynamic Domain Architectures for Model Based Autonomy for (DDA for MBA)

Proposal for DARPA BAA 00-19
Technical topic areas: Model-Based Generation Technology
Contractor's type of business: Other educational
Authors: Brian Williams, Robert Laddaga, Howard Shrobe

Technical point of contact:
Dr. Brian Williams
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Mass. 02139
(617) 253-2739
Fax: 617-253-5060
E-Mail: `williams@mit.edu`

Administrative contact:
Robert Van De Pitt
Office of Sponsored Programs
Room E19-750
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, Mass. 02139
(617) 253-3884
Fax: (617) 253-4734
E-mail: `vandepit@mit.edu`

Brian Williams, Principal Investigator, MIT

Director, MIT Artificial Intelligence Laboratory

Office of Sponsored Programs Representative

Contents

A Innovative Claims

The nature of embedded system software has changed fundamentally. In avionics, for example, yesterday's embedded software was a federation of relatively simple control loops; but today's embedded software systems are the glue which integrate the entire platform, presenting high level situation awareness to the pilot. Soon we will have UCAV's in which software will play the role of the pilot, blurring the traditional boundary between pilot and platform and requiring wholly new forms of integration.

Our current techniques for producing these systems are inadequate. The traditional approaches of entangling concerns in order to meet space and time constraints has led to software systems inconsistent with the goal of evolution. Model-based methods offer a better approach. We argue that:

Programmers have a variety of *models* in their heads; these models structure the problem into functional domains, into levels of abstraction and into techniques for managing cross-cutting concerns. They use these models to translate high level goals into code. And then *they throw away the models leaving behind just the tangled, brittle code* that characterizes today's embedded system software.

The power is in the models. These should be represented explicitly and used to synthesize efficient reactive code. *The models should not be thrown away*, they should be available in the runtime system and to faciliate integration and robustness.

The software must present a model-based view of its execution in terms of goals, strategies, plans, state-variables, and invariants. These should be monitored at runtime and the failure to achieve expected results should trigger diagnosis, reconfiguration and recovery.

At the lowest level, reactive controllers iterate through the following three steps: Mode Identification which attempts to characterize the current state of the system; Mode Selection, which is invoked if the current state is faulty, and which identifies a working mode of the system that is reachable from the current state; Reactive Planning which identifies a set of steps capable of achieving the goals posted by Mode Selection and by external components.

Thus the software must make deductions at reactive speeds. But, the computational power available to embedded systems is becoming abundant, allowing model-based deduction to be embedded even in the lowest level, reactive components of the control systems.

Embedded software should be developed against the backdrop of model-based frameworks, each tailored to a single (or a small set of) issues. Some frameworks (those of concern to the PCES BAA) deal with cross-cutting issues such as fault tolerance; others deal with particular components and functional layers of a complex system (e.g. control of an observational asset in a UCAV). 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 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. Each framework exposes its own goals, plans, state-variables and invariants in terms of an embedded language which captures the terms and concepts used by application domain experts. Each framework also expresses its dependencies on other frameworks in the same language. These are then used to synthesize the code which integrates the ensemble of frameworks into a unified, reactive system.

B Proposal Roadmap

Main goal of the work: Our goal is to provide a toolkit to support the designers of Model-based Domain Specific frameworks for autonomous embedded systems (page 16). These frameworks rely on a new generation of intelligent reactive control systems that reason from models within real-time control loops (page 12). The toolkit provides software generators that synthesize the necessary code to coordinate the interactions between the several frameworks that make up a system as well as the code that links these frameworks to the intelligent reactive executives.

Tangible benefits to end users: Embedded software *Framework developers* will have access to a toolkit that allows them to capture and structure the knowledge of an application domain; the packing into a framework will allow integration with other frameworks covering other domains of application. (page 5). *Application programmers* will be able to implement and evolve application functionality at a high level, in concise, domain-appropriate, terms. *Application Users* will obtain more functionality in a more timely fashion and they will be able to rely upon the correctness and performance of the application as it continues to evolve. (page 3).

Critical technical barriers: Systems with extreme dynamism, introspection and reflection facilities have traditionally been too big, slow and unpredictable. Traditional reactive systems have been too brittle to achieve the dynamism needed for autonomy. We need to demonstrate that we can provide reactivity and predictable, high performance while supporting the expressiveness needed for model-based approaches. (page 10)

Main elements of the proposed approach: Our overall approach couples a type of model-based software framework with model-based reactive executives (page 3). We will design and implement a new generation of reactive model-based executives that reason within the control loop. (page 12). We will design Dynamic Domain Architectures, a domain-specific, model-based software framework that operates in terms of goals and plans. DDA frameworks generate the code necessary to couple to other frameworks by passing goals and sharing state observations; they also generate the code that couples them to the reactive executives that run the physical systems. (page 16).

Specific basis for confidence that the proposed approach will overcome the technical barriers: We have already developed a first generation model-based reactive executive which has been deployed in the NASA Deep Space 1 spacecraft (page 11).

Nature of expected results: We expect to drastically shorten the software development cycle for embedded systems by facilitating the development and integration of domain-specific frameworks (page 16), frameworks dealing with cross-cutting aspects and model-based reactive executives (page 12).

The risk if the work is not done: Without this proposed work we suspect that embedded systems will always be too brittle to sustain autonomy. (page 10)

Criteria for evaluating progress and capabilities: We will measure progress on two scales: The first is the ease of developing useful frameworks. We will attempt to conduct parallel developments of frameworks being built by other contractors using more traditional tools and measure the improved productivity under our approach. The second scale is the performance of systems built using our implementation techniques; we will measure this by observing the reactivity of our experimental systems (page 20).

Cost of the proposed effort for each contract year
Y1 $849,001; Y2 $881,307; Y3 $916,158;Total $2,646,466.

C Technical Description

C.1 The Problem

In the future, unmanned air and space vehicles will need to act autonomously to robustly achieve elaborate missions within uncertain and hostile environments. Achieving this robustness will require fluid coordination of the varied functions and subsystems of the vehicle. To survive exceptional situations these vehicles must embody the kind of resourcefulness exemplified by humans during the Apollo 13 crisis – reasoning from multiple models of the system to quickly diagnose the source of anomalous events, and quickly responding by exploiting all available assets in a novel manner if required. Approaching this level of autonomy has proven extremely costly; for example, some unmanned air vehicles require up to two dozen personnel to support operations.

Why is this? A dominant cost for highly autonomous missions is the number of operators and programmers who think through all possible system interactions in order to robustly manage the internals of the vehicle. This problem becomes exacerbated as the capabilities of these vehicles rapidly increase. The added complexity and fluidity of these vehicles precludes the traditional operations and software development approach. Traditionally, embedded system software has been crafted by progammers with the mindsets of "cobblers" and "weavers". Cobblers shoe-horn large volumes of code into a small package; weavers knot together cross-cutting concerns, producing entangled, unevolvable and brittle code. [1] In an era of long-lived and highly evolutionary systems this approach is untenable. The embedded software systems of concern to us and DARPA are, in effect, perpetual, they are deployed for a very long time, accumulating new capabilities and exploiting new technologies throughout their lifetimes.

Consider the following example: We are in the future, in an era of autonomous aircraft. A new sensor package has been developed for the JUCAV (Joint Unmanned Combat Air Vehicle). This high-powered optical imager is to be mounted on the underside of the aircraft; it pivots in a ball-joint housing allowing 35-degrees of rotation in each direction. The intent is for the plane to fly a path that maximizes the use of this optical sensor while avoiding enemy defenses and sensors. Now that the hardware is built, the manufacturers are prepared to ship the system; there's "just a small matter of software" remaining to be done. This small task involves integrating the control of the sensor package with the navigation, guidance and control of the airframe.

This software task is hardly trivial; it's likely to sink the project, as have so many other software integration tasks. On the other hand, there is something logical and appealing about the idea that the task *should* be simple. Why can't the software that controls the sensor package simply ask the aircraft's navigation and control software to take a course that keeps its observation target within its range of motion. Today, such joint control would be hand programmed by a team of programmers who think through all the potential system interactions. This programming is usually single purpose, often done wrong and it works only if everything is well behaved. If a processor should fail, or an actuator should get sluggish, then the mission is likely to be aborted.

C.2 The Solution

We propose to develop technology that makes such software integration possible, that enables a wide range of autonomous operations, and that responds to unforeseen breakdowns, achieving as good a result as is possible under the circumstances.

[1] We have also responded to BAA 00-23 PCES with a proposal that has much the same philosophy as this one. However, in that proposal we concentrate mainly on programming language technology and on aspectual decomposition. In this proposal, we concentrate on model-based autonomy and the integration across functional units.

Let us consider what today's programmers know and think about when they program a system like our imagined JUCAV. First, they understand the goals of the sensor, they understand its degree of maneuverability as well as that of the aircraft; they also understand the computational demands of each task and the degree to which the physical tasks are affected by choices in the information domain (e.g. the choice of sampling rate and number of bits of precision affects the accuracy and stability of control).

In short, today's programmers have a variety of *models* in their heads while they write the code; these models structure the problem into functional domains, into levels of abstraction and into techniques for managing cross-cutting concerns. They use these models to translate high level goals into the tangled, shoehorned and brittle code that characterizes today's embedded system software. And then they throw away the models; what is left is just the code.

We believe that the key to progress is to acknowledge that *the power is in the models.* These models need to be explicitly represented and then used to synthesize efficient reactive code capable of the responsiveness needed for a high-performance aircraft. However, *the models should not be thrown away*, they should be available in the runtime system. Even though the executing software must be highly reactive, it still must be able to present a view of its execution in terms of goals, plans, invariants and the like. These should be monitored at runtime and the failure to achieve expected results should trigger diagnosis, reconfiguration and recovery. In order to maintain flexibility, the system should always possess a variety of strategies and plans for achieving its goals. It attempts to achieve its goals by selecting that strategy and plan that it believes to be best suited to the task. The system must monitor its progress and change its plan and strategy if appropriate progress is not being made.

We address these concerns with two coordinated efforts:

1. We will design and develop a new generation of model-based reactive executives to execute the control loops. These will be capable of characterizing the state of the system and planning actions to make the system's state be consistent with externally imposed goals and constraints. To do this the executive must be capable of model-based reasoning at reactive time-frames.

2. We will develop a new form of model-based software framework based on our work on dynamic domain architectures (DDA). These frameworks will capture and structure both the code and the models for a functional domain of embedded systems and will present a goal-directed model of the software components to other frameworks. Programming in DDA frameworks will be facilitated by extending a powerful dynamic programming language with compositional, model-based capabilities. Such an extended language will allow the behaviors and interactions between subsystems of intelligent vehicles to be easily specified at a commonsense level. A Framework Developers Toolkit will generate the code necessary to couple multiple DDA frameworks to one another and to the reactive model-based executives.

These efforts will be based on the synthesis of two research lines: The first of these is the line of research that has led to the successful deployment of the Remote Agent software on the Deep Space 1 space-vehicle this past year (this work was led by Brian Williams, the PI of this proposal) [MNPW98]. The second line of work is captured in the DARPA sponsored Dynamic Domain Architecture project at the MIT AI Lab. This project has focussed on the programming environment, language technology and runtime infrastructure needed to implement the model-based, self-adaptive systems of type we imagine running our imagined JUCAV.

The elements of our solution involve the following insights:

C.2.1 Reactive Executives for Model-Based Autonomy

- Computational power for embedded systems is becoming abundant. Commodity chips, providing up to 1 GHz performance and tailored to embedded software applications, are being produced by mainstream vendors (e.g. Motorola, IBM, Intel). Platform wide communication networks will soon provide gigahertz bandwidth. We can and must use this power to ease the design problem and to allow structured solutions.
- This new abundance of computational power, together with new algorithms developed in our previous work [WN97] makes it possible for model-based deduction to be performed within the real-time control loops of an executive. The system can afford to think about how it is controlling the physical plant.
- The reactive control loops of an embedded system should be seen as the continous execution of three model-based reasoning steps: (1) Mode Identification, characterizes the possible current configuration of the system; (2) Mode Selection, is invoked if the current configuration is inappropriate; it figures out a realizable configuration satisfying all imposed constraints; (3) Reactive Planning, form a plan to reach the selected mode and then executes the first steps to achieve the satisfactory configuration.
- The modularity and simplicity of the models which guide such executives will support extensive reuse. A set of increasingly capable execution kernels can be developed that coordinate internal interactions of intelligent vehicles by performing extensive planning and deduction from models at reactive time scales.
- Such reactivity at the lowest levels allows stategic and tactical guidance to be operationalized immediately. The executive is never stuck in a mode inconsistent with the goals of higher levels of the system.
- Such reactivity at the lowest levels allows stategic and tactical decision making to be informed of the instantaneous state of the system, preventing it from wasting valuable computational resources on planning an action which is no longer possible, timely or relevant.
- These ideas inspired our Livingstone system which was deployed in the Deep Space 1 spacecraft. However, these ideas can and should be generalized and reified into a software framework of broader applicability so that they can be used in a wide variety of embedded applications.

C.2.2 Software Frameworks for Dynamic Domain Architecture

- Embedded software should be developed against the backdrop of *frameworks*, each tailored to a single (or a small set of) issues. Some frameworks (those of concern to the PCES BAA) deal with cross-cutting issues such as fault tolerance; others deal with particular components and functional layers of a complex system (e.g. control of an observational asset in our imagined JUCAV).

  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 run-time infrastructure providing a rich set of layered services, models describing the goal-directed struture of these software services, a protocol specifying the rules by which other software interacts with the infrastucture provided by the framework, 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. goals, alternative strategies and methods for

achieving goals, subgoal structure, state-variables, declarations, assertions, and requirements). Such annotations inform program analysis. They also facilitate the writing of high level generators that produce wrapper code integrating the multiple functional frameworks. We believe that we can provide a generator toolkit for framework developers.

- System development involves a new player, the framework developer, who plays the role of a bridge between application developers and systems programmers. Framework developers provide tools that are too domain (or issue) specific for general system programmers to attend to, but too much in the style of core system code for application programmers to attend to. Framework developers extend and raise the level of the language and infrastructure that the application programmer uses to solve problems. Framework developers provide tools that synthesize the necessary low-level reactive code from the high level embedded language ofthe framework.
- The core functionality of the system is decomposed along physical lines. Families of functionally similar components in a common domain (e.g. Optical Sensors) are managed by parameterized frameworks that cover that domain. Such a framework embodies the domain architecture for this area of functionality. A domain architecture structures the procedural knowledge of the domain into layers of services, each capable of achieving specific goals; a service at one layer invokes services from the lower layers to achieve its subgoals. Each service has many implementations corresponding to the variability and parameterization of the domain. Each alternative implementation represents an alternative strategy, method, or plan for achieving the goal. The choice of which implementation is to be invoked is made at runtime, in light of runtime conditions, with the goal of maximizing expected utility. Such frameworks are therefore *Dynamic* Domain Architectures. Each such framework exposes to other frameworks its models, goal structure, state-variables, its API, its protocol of use and constraints on those subsystems that interact with it.
- Conceptually, these frameworks interact at runtime by observing and reasoning about one another's state and by posting goals and constraints to guide each other's behaviors. The posting of goals and constraints and the observation of state is facilitated by wrapper code inserted into the code of each framework at generation time by model-based generators of the interacting frameworks. The use of generated observation and control points as well as the use of novel, fast propositional reasoning techniques allow this to happen within reactive time frames. The composite system behaves as if it is goal directed while avoiding the overhead normally associated with generalized reasoning.
- The full application now consists of core functionality built by composing a variety of model-based frameworks as well as a variety of *cross-cutting aspects as sought by the PCES BAA*, each written in the embedded language of a specific framework and each describing how that aspect is woven into the core functionality.
- The frameworks capturing Domain Architectures as well as those capturing various cross-cutting aspects evolve and are maintained separately; they are bound into the final system as late as necessary. Software maintenance and evolution are decomposed along the lines of frameworks.
- A rich object-oriented language system, derived from the best ideas of Lisp and similar languages, can provide the base level platform for this new approach to embedded systems development. This core language system consists of a Virtual Machine, a Meta Object Protocol, a class system featuring multiple inheritance, multiple argument dispatch and method combination, a dynamic condition handling and recovery facility, and a powerful procedural macro system. These tools provide great power for expressing and synthesizing the code implementing model-based frameworks.
- This rich Dynamic Object Oriented Programming environment provides these rich service not just in the development cycle but also within the runtime environment. The availability of runtime dynamic redefinition, late binding, condition and exception handling, generalized diagnosis and recovery support make this the idea platform for supporting not just DDA

frameworks but also for supporting the model-based reactive executives.

C.2.3 Off Line Analysis

Although our focus is on the two areas above, it is important to understand that this approach makes the static analysis of the system easier. At both the Framework and the Executive level, such a system is inherently adaptive, attempting to guide itself away from anomalous states and back towards the intended behavior. This adaptivity makes the static analysis easier by reducing the possibility of the system taking excursions far away from the intended behavior.

In addition, the composition methodology itself lends itself to analysis. We have proposed to attack the analysis problem in our response to the PCES BAA, but we summarize the approach here because it carries over:

- Part of each framework is a protocol specifying the rules under which other components are supposed to interact with the infrastructure provided by the framework. Corresponding to this protocol is a proof that certain properties will hold as long as the protocol is adhered to. This proof is conducted off line once by the framework developers and delivered as part of the framework. The fact that the protocol is adhered to by other parts of the system is often guaranteed by simple inspection or checking methods. Thus, part of the analysis of the system is done once offline and then repeatedly reused.
- Corresponding to each framework are analytic tools that can be used to examine the code that 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. Analysis of the system can proceed iteratively with each framework first showing that it satisfies the constraints placed on it by others and then determining what constraints it places on its sibling frameworks in order to guarantee its properties of interest. The analysis and understanding of the overall behavior of the system is, therefore, decomposed in just the same way as are the development, maintenance and evolutionary tasks.
- Each framework establishes its own natural proof techniques; therefore heterogeneous reasoning capabilities are needed to support the analysis of software decomposed into frameworks. We have long experience in the development of Joshua, which is just such a reasoning system.

C.3 Examples

We will present a few examples of how this model-based approach can deal with the integration of diverse sub-systems and how it can deal with breakdowns during the mission.

We start with a simple example of normal operation, illustrating model-based integration between sub-systems.

C.3.1 Model-Based Integration of Multiple Frameworks

Our JUCAV is sent on a observational mission to fetch optical imagery of a target.

- The sensor management framework registers a goal with the navigation, guidance and control framework to be notified when the aircraft reaches the position designated for the start of the observation pass.
- When this point is reached the sensor management framework is notified and puts the sensor into operation.
- The sensor management framework also posts a constraint, requesting the navigation guidance and control framework to keep the aircraft within a corridor allowing its sensor to lock onto

the target during the entire observation pass. Goal directed reasoning within the navigation and control framework deduces and issues a set of low level flight control commands that will keep the trajectory consistent with the posted constraint.
- In addition, the sensor management framework passes in utility-based guidance expressing its preferences for positions that lead to better observational angles. Other frameworks, for example the threat avoidance component, also express their preferences and utilities. The navigation guidance and control framework then attempts to find a course that produces maximum expected global utility, trading off the utilities and probabilities of its various clients.
- During the observation pass, the sensor management framework is continuously given positional information from the navigation, guidance and control framework and uses this to aim the sensor. Positional information flows down the abstraction layers within the sensor management framework leading to the reactive executive issuing the appropriate motor commands to the sensor's actuator.

Suppose, however, that during the approach to its observational target our JUCAV experiences a problem with the rotational actuator of its optical sensor and that this limits the range of motion of the sensor. This problem would be dealt with as follows:

C.3.2 Dealing With a Physical Breakdown

- Internal monitoring within the sensor management framework would notice that issuing actuator commands does not result in appropriate positioning of the sensor. This causes an abnormal condition to be signalled.
- Model-based diagnostic routines are invoked to determine the most likely cause(s) of the failure. In this case it is determined that the actuators aren't working appropriately.
- The sensor management framework contains many recovery plans for this type of failure. A promising recovery plan is selected. The first step of this plan is invoked; it attempts to characterize the failure by issuing a series of motion commands to the sensor, exploring the intended range of operation. The result of this test is to determine that the range of motion is limited, but that positioning is accurate within the intended range.
- The sensor management framework completes its recovery plan by posting a new constraint, requesting the navigation guidance and control framework to keep the aircraft within a narrower corridor, one that will allow its sensor to lock onto the target during the entire observation pass. It is now the task of the navigation and control framework to pass this information to the appropriate reactive executives; these issue sets of low level flight control commands that will keep the trajectory consistent with the posted constraint.
- During the observation pass, the sensor management framework continuously fetches positional information from the navigation, guidance and control framework and uses this to aim the sensor appropriately.

C.3.3 Computational Breakdown

Now assume that one of the aircraft's onboard processors fails, reducing the supply of computational power available for the delicate task of guiding the plane through the observational pass. This might be handled as follows:

- Monitoring routines generated by a cross-cutting fault-tolerance framework notice that the failed processor has stopped functioning. This causes an alert to be generated and top level fault tolerance routines to be invoked.
- The fault tolerance framework migrates the tasks assigned to the failed processor to other processors. New resource allocation budget and schedules are calculated and distributed.

- The navigation, guidance and control framework deduces that it no longer has the computational budget to execute the highly accurate control routines needed to maintain its position within the tight corridor required by the sensor management framework. It instead chooses to use less computationally expensive, but less accurate versions of the control routines.
- During the observation pass the sensor management framework continuously fetches positional information from the navigation, guidance and control framework. Some of the time, the position is inconsistent with the maneuvering range of its optical sensor. It does not engage the imager in such cases. But most of the time, there is a realizable orientation for the imager that will sense the target and in those cases the imager is engaged.
- Even though there has been a dual failure (imager actuator and processor) useful observations have been obtained.

C.3.4 Change of Strategy

Now imagine that while doing the above the emitter-locator framework reports that enemy sensors have possibly locked on.

- The emitter-locator framework calculates the likely location of the enemy sensor and using models of enemy capabilities it calculates an area of observability for those sensors.
- The emitter-locator framework creates a constraint for the navigation guidance and control framework to avoid the area covered by the enemy sensor.
- The navigation guidance and control reactive executive reaches a point in which it can no longer satisfy all the constraints. (There is no longer any place useful for observation that it can guide the plane to that isn't also likely to be covered by the enemy's sensors).
- The executive signals a condition to all the frameworks involved in causing it be overconstrained. Each of these attempts to handle the condition.
- The sensor management framework handles the condition by reporting to the mission planning framework, which had originally invoked it, that it cannot make any useful observations in the current circumstances.
- Mission planning abandons this particular observation pass and looks for another goal that is still achievable.

We do not claim that these are high fidelity examples, capturing all the subtleties of modern avionic systems. But these examples do illustrate the ease with which integration and recovery occur when the language of integration is goal-oriented, it is based in the models describing each framework, and the reactive executives that run the physical plant speak this same language.

We now briefly present more details of our technical approach. We begin by reviewing our past work on the Remote Agent system which was deployed on the Deep Space 1 spacecraft. We follow this with an outline of the work needed to create the next major improvement in Model-Based Autonomous systems. We then give a sense of how Framework developers will interact with our system through a Framework Developer's Toolkit. This toolkit is based on our work on dynamic domain architectures and its enriched programming language substrate. In particular, we will show how the DDA development environment allows model-based synthesis of composite frameworks to work.

C.4 Model-Based Autonomy

A new generation of intelligent embedded systems, such as unmanned air vehicles, is emerging that are sensor rich and massively distributed. This includes a diverse set of space and aeronautic systems of interest to DARPA. Metaphorically speaking, such systems will achieve high performance and longevity by exploiting their vast nervous system of sensors to model internal constituents, and by exploiting sophisticated immune and regulatory systems that use these models to maintain and control the internal functions.

Robust autonomous vehicles contain an internal web of sensors, actuators and microprocessors. Our work to date has focussed on techniques for automating the "nervous", "regulatory" and "immune" systems of these sensor/actuator webs, within the space craft domain. Nervous system functions include the communication of commands and data along distributed, multi-processor networks; regulatory functions include the distribution of power, oxidizer and fuel to engines and instruments, and immune functions involve the management of redundant components and functions. A dominant portion of flight software and mission operations is currently devoted to these functions. This includes interpretation tasks like state estimation, command and goal confirmation, monitoring, fault detection, isolation and diagnosis. It also includes command tasks like device commanding, hardware reconfiguration, repair and control policy coordination.

The problem of developing these long-lived systems is an instance of one of computer science's most pressing challenges – managing software complexity resulting from system interactions. A real-time system's immune function is typically comprised of the following ten tasks: goal and command tracking, software/hardware reconfiguration, anomaly detection, fault isolation, diagnosis, fault avoidance, recovery and safe shutdown. Hand coding each task in a low-level procedural language like C requires the programmer to reason through system wide interactions, along lengthy paths between the sensors, control processor and control actuators. The resulting code typically lacks modularity, is fraught with error and makes severe simplifying assumptions.

Note, however, that to write this code programmers use their knowledge of how software and hardware modules behave and interact. This implicit knowledge is simple, modular and highly reusable. The idea of model-based programming is to exploit this modularity, by having engineers program real time systems by articulating and composing together these commonsense models. The scientific challenge of this endeavor is to develop computationally tractable methods that generate the immunological functions necessary for survival by automating all reasoning through systemwide interactions from commonsense models. Management of system interactions include automated capabilities for command sequencing, resource allocation, monitoring, diagnosis and selection of contingencies.

C.4.1 Reactive Model-directed Executives

A major driver in the complexity of long-lived systems is the fact that surviving an anomalous situation requires fast reaction. This has led to a tradition of having programmers envision all likely failure situations and all possible contingencies at coding time. A somewhat analogous tradition has dominated artificial intelligence (AI) for the last decade, by focusing on robotic execution systems that avoid deduction within the reactive loop at any cost. However, since unmanned vehicles often operate in harsh and hostile environments, the task of explicitly enumerating responses to all possible situations quickly becomes intractable. This problem was highlighted in the Apollo 13 crisis where a quintuple fault occurred, and the solution included an innovative repair that exploited a sneak path to a secondary battery.

The first thrust of our proposed research tackles this complexity problem by breaking away from the compile-time tradition. Instead we will develop model-directed executives that wait until the moment that an unexpected situation occurs, and only then synthesize a timely response, based on onboard models. The decision to wait reduces complexity dramatically, by replacing a breadth of possible futures to be planned for with a single reality. However, the need to respond correctly in novel, time-critical situations introduces the challenge that these model-directed executives must perform extensive deductive reasoning within the reactive control loop, sometimes on the order of tens of milliseconds. This extensive use of fast deduction goes directly counter to the conventional wisdom of AI, particularly during the early 90's, where it was believed that even mild forms of deduction would become intractable.

Three research trends offer us insight into meeting the challenge of providing fast deduction. First, experimental research on phase transitions between solvable and insoluble constraint problems demonstrated that hard satisfiability problems are strikingly elusive to find. This hints that real world problems, while NP hard, could often be quite manageable. Second, great strides have been made in the development of efficient search methods, highlighted for example by Deep Blue beating Kasparov. Finally, the model-based diagnosis community has found that adequate diagnosis can be achieved using strikingly simple models, and correspondingly simple deductive algorithms. At the same time these diagnosis algorithms have scaled to thousands of components [dW89, dW87], and have response times within a factor of roughly 100 of what our long-lived agents might require.

C.4.2 Accomplishments

During the last four years we have developed a model-based configuration management and fault protection capability, called Livingstone, that automatically performs these regulatory and immune tasks from a set of component level models. These tasks are achieved robustly by reasoning extensively about system interactions at reactive time scales. Livingstone's capabilities were recently flight validated as part of the remote agent autonomous control experiment aboard the Deep Space One probe, which received NASA's 1999 software of the year award.

Livingstone is currently being incorporated within a variety of NASA's mission test beds, including X-34 and X-37 reusable launch vehicle demonstrators, the ST-3 separated spacecraft interferometer, the Marsokhod rover, and an in situ propellant production plant. Hence real-time, model-based, deduction has recently emerged as a vital component in AI's toolbox for developing highly autonomous reactive systems.

Livingstone is a first generation executive that is highly responsive due to three key properties. First, Livingstone identifies and reconfigures modes by implementing a new kind of deductive feedback controller that generates optimal responses. Second, Livingstone uses a single onboard model specified in a representation that combines the transition system models underlying reactive languages with qualitative representations from model-based reasoning. This formalism achieves broad coverage of hybrid software/hardware systems, yet can be reasoned about efficiently. Finally, Livingstone meets the stringent requirements of responsiveness by using an efficient search kernel based on an event-driven, deductive database [NW97].

From the research standpoint, Livingstone provides an interesting existence proof that a useful level of highly responsive, deductive system can be built. It also offers a starting point for a variety of far more ambitious methods that are at the opposite end of the spectrum from traditional reactive approaches. As one example, Livingstone's ability to plan sequences of action is quite limited. Relevant research questions include, how much more expressive can Livingstone's sequencing ability be made, while preserving the desired level of responsiveness? Can a hierarchy of model-directed algorithms be developed that trade increased expressiveness for decreased response time? In order to improve responsiveness, what level of contingency planning can be added in before the complexity of the number of options becomes problematic? To support large-scale model-based programming, what encapsulation mechanisms should be drawn from classical programming languages, and what testing methods should be taken, for example, from declarative debugging? How can reduced ground teams of the future most effectively interact with model-directed health management and execution systems?

We explore two complementary development paths to see how far we can go in answering these questions. In the first of these we explore the development of a new generation of goal-directed and model-based reactive executives. In the second, we examine capturing and structuring high-level domain knowledge by introducing an explicit goal-orientation in a high level dynamic object oriented programming system.

C.4.3 Developing Autonomous Systems with Survival Instincts

To be long lived an autonomous system must have basic survival instincts. For example, a ground team accidentally commanded the Clementine spacecraft to point away from Earth, thus losing communication forever. This is the focus of our second research thrust. To avoid this problem, Clementine would have needed the ability to recognize and avoid damaging actions. We are developing planning algorithms that take a conservative approach, by disallowing any action sequence whose effects cannot be reversed. A second type of survival instinct is wariness of the unknown. For example, when a new vehicle is first deployed, little is known about how it will behave in its new environment. Hence, ground operators act very tentatively, gaining familiarity with the spacecraft and its quirks over a period of time, before trying anything ambitious. Model-directed systems need to be similarly wary. They should initially be distrustful of the models they have been programmed with, developing confidence only as relevant parts of the model are exercised. This raises a variety of interesting research questions, for example, how do model-directed systems estimate the accuracy of their concurrent transition system models from performance data? How do model-directed systems generate control sequences that are optimal in terms of determining the accuracy of their models? How should model-based planning algorithms embody caution, by exploiting these estimates of model accuracy?

C.4.4 Developing Distributed Collections of Model-based systems

The research outlined thus far assumes that the control of model-directed autonomous systems is highly centralized. In reality most complex real-time systems have computation distributed throughout, along pathways to the sensors and actuators. Imagine instead a design approach in which each major component of an autonomous system contains its own local model and deductive capability; we'll call this a model-based agent. An autonomous system is then programmed and reprogrammed by simply plugging together its components. The challenge of our third research thrust is, how do we make the collective behavior of these model-based agents mimic that of the centralized approach, discussed thus far?

For example, Livingstone, our first generation model-based executive, performed a set of roughly ten immune tasks based on a single deductive search algorithm. How can this algorithm be distributed? The search algorithm is a form of combinatorial optimization problem with propositional logic constraints. An intriguing approach to solving classical combinatorial optimization problems using a distributed auction mechanism has been developed by Bertsekas at MIT. An interesting research question is, can auction algorithms be generalized to solve the deductive search problem at the core of model-directed autonomy?

C.4.5 A New Generation of Reactive Executives for Model-Based Autonomy

The long-term goal of this project is to provide the nervous, regulatory and immune functions for coordinating the internal subsystems of vehicles and to coordinate large collections of vehicles (e.g. a complete UCAV network). At the highest level this involves maintaining UCAV networks in which the sensors and actuators are themselves highly robust, autonomous vehicles. At the lower level this involves maintaining internal vehicle webs whose sensors and actuators may be complex, computer controlled instruments. The focus of this effort is the later situation.

To date, our approach for raising a vehicle's level of autonomy has involved writing the lower, reactive levels of control software in a traditional, low-level language like C. Next a set of software components are placed on top of this substrate; these reason about system interactions to manage hidden variables (e.g. Livingstone), or they automatically choose between alternatives procedures based on uncertain outcomes (e.g. task decomposition executives). The use of these separate components with their distinct representations has proven clumsy at best.

Our approach in this project is to incorporate these services into the run time execution kernel of the reactive language itself. This seamlessly places the power of these deductive capabilities at the disposal of the programmer. The net effect is to offer a programming abstraction that allows the programmer to avoid reasoning about many aspects of hidden variables and uncertain outcomes.

Since our goal is to support the distributed coordination of robotic webs, we start with a concurrent reactive programming language that has a clean semantics, such as Esterel. We then minimally extend the language and execution kernel to model and address the hidden state and uncertain outcome issues.

The central problem of managing system interactions, mentioned earlier, is the complexity of the reasoning that programmers must perform in order to observe or change the values of hidden variables. Unlike traditional reactive languages, we allow the programmer to refer to hidden variables as if they could be directly sensed or actuated. It is then the responsibility of the language's execution kernel to reason through the system interactions necessary to observe or control these hidden variables.

For example, suppose a program residing at a central command has been invoked to execute a coordinated mission scenario among distant UCAVS. This program might specify that each UCAV execute a maneuver by accelerating along a trajectory, as if these accelerations are directly controllable. It is then the responsibility of the underlying execution kernel to autonomously determine a setting of control variables for the UCAV that achieve this acceleration, and then to estimate the acceleration to confirm that the maneuver has been achieved. It is also the responsibility of the execution kernel to determine a way to send messages from the central command to the desired UCAV through the UCAV's communication network.

The problem of managing uncertain outcomes is the need to determine which of a distribution of possible outcomes has occurred and to decide which procedure is optimal based on these outcomes. In our approach we allow programs to specify an *a priori* distribution of possible out comes (e.g., an actuator might move or fail stuck closed), and to specify a series of alternative actions with corresponding rewards. It is then the responsibility of the execution kernel to determine the relative likelihood of these outcomes as observations are made (i.e., belief state computation), and to choose between alternative actions based on these outcomes (i.e., solving a decision process). This is similar to the function of a task decomposition executive, with the important exception that the uncertain outcomes and actions may once again involve estimating or controlling hidden variables.

Providing the above capability requires developing three components. First, we need to develop a programming language that achieves the above desiderata and that has a clean semantics. We refer to the language as the Reactive, Model-based Programming Language (RMPL). Its semantics is in terms of partially observable Markov decision processes (POMDPS). Second we must develop an efficiently executable object code and a compiler to this object code. We specify this object code in a representation called a hierarchical, constraint-based POMDP (HCPOMDP). Finally we must provide an execution kernel that efficiently solves the POMDP on line. This involves automated deduction techniques that perform efficient state estimation, monitoring and control sequence generation. These are discussed further in the next few subsections.

C.4.6 The Reactive Model-based Programming Language RMPL

Our preferred approach to developing RMPL is to first introduce a minimum set of language primitives that are used to construct programs. Each primitive that we add to the language is driven by a key design principle of the language. Using a minimum set of primitives is particularly important in order to facilitate the problem of developing techniques for reasoning about these programs. We then define on top of these primitives a variety of program combinators that make the language easy to use.

To support reasoning about interactions our design must support the specification of co- temporal constraints and variable assignment. To achieve the expressivity of synchronous reactive programming languages our design must support conditional branching, iteration and preemption. To handle distributed processes our design must support logical concurrency. To handle uncertain outcomes our design should support probabilistic choice, and to support optimal selection between alternate actions our design should support decision theoretic choice. A candidate set of primitive combinators for RMPL that meet these design criteria are listed below, where c denotes a constraint and A and B denote legal RMPL programs:

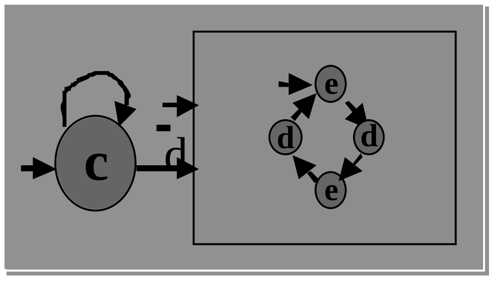
```
c                                interaction or assignment
If c next A                      conditional branching
Unless c next A                  preemption
Always A                         iteration
A,B                              concurrency
Choose with Probability          probabilistic choice
        {P1 A1}
Choose with Reward               decision theoretic choice
        {P2 A2}
```

Semantically an RMPL program is a specification of a partially observable Markov decision process (POMDP). An execution kernel for RMPL is a decision procedure that approximately or exactly solves the specified POMDP. This is in contrast to most synchronous, reactive programming languages, like Esterel, Signal, Lustre and State Charts [Hal93, MP92, BA93], whose semantics are specified as traces through determinate transition systems.

C.4.7 Hierarchical Constraint-based MDPs

While POMDPs offer a natural way of thinking about reactive systems, as a direct encoding they are notoriously intractable. To develop an expressive yet compact encoding we introduce four key attributes. First, the POMDP is factored into a set of concurrently operating automata. Second, each state is labeled with a constraint that holds whenever the automaton marks that state. This allows an efficient, intentional encoding of co-temporal processes, such as fluid flows. Third, automata are arranged in a hierarchy – the state of an automaton may itself be an automaton, which is invoked when marked by its parent. This enables the initiation and termination of more complex concurrent and sequential behaviors. Finally, each transition may have multiple targets, allowing an automaton to be in several states simultaneously. This enables a compact representation for recursive behaviors like "always" and "do until". The result is an encoding that is roughly linear in the size of the original RMPL program.

An example of an HCMDP is shown below in figure 1. Circles denote primitive states, while squares denote automata, which are composite states. Each primitive state has an associated constraint. Arrows with no source state indicate start states. Some automata have multiple start states and transitions to multiple next states. Transitions may go to composite states, in which case their contained start states are immediately marked. Finally, an arc is enabled if its condition is logically entailed by the constraints of the currently marked states. If the condition has an over bar, then the transition is enabled if the condition is not entailed. This allows preemption mechanisms to be encoded.



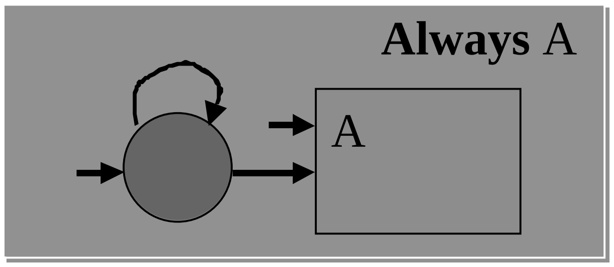


Figure 1: HCMDP Encodings

The RMPL compiler will be responsible for mapping each of the above seven RMPL combinators into an HCMDP encoding. For example, an encoding for the “Always A” combinator is shown in the lower half of figure 1

C.4.8 A Distributed Model-based Execution Kernel

The execution kernel is supplied an RMPL program specifying the desired behavior to be achieved and a second RMPL program specifying the behavior of the constituents of the robotic web that are to be commanded and sensed. Each RMPL program is encoded as a hierarchical, constraint-based POMDP. The execution kernel is given a trace of observables, and then sequentially decides values of control variables online that solve the POMDP. To accomplish this we build upon the constraint-based optimization, belief update, reconfiguration and planning algorithms incorporated within our past Sherlock and Livingstone systems.

Recall that our goal is to ultimately manage robotic webs whose sensors and actuators are themselves intelligent vehicles. To model the behaviors of these vehicles we must exploit the full expressive power of RMPL as a modeling language. To sense and actuate state variables on these vehicles the execution kernel must be able to reason about RMPL programs. We will develop efficient belief state update algorithms for RMPL through a careful generalization of the state enumeration algorithms that are successfully employed by the Sherlock [dW89, dW87] and Livingstone [WN97, WN96] systems on simple modeling languages. Likewise to support indirect control we will develop reactive planning algorithms that can generate control sequences for commanding devices described by RMPL programs. This is a substantial research problem and will be addressed in years two and beyond. Together these capabilities will help elevate the executive from the tasks of commanding and configuring simple vehicle components to commanding and configuring collections of intelligent vehicles.

C.5 Dynamic Domain Architectures

Reactive executives form the lowest tier of model-based control. Our goal is to use models to integrate components at all levels of the system, both reactive and deliverative. To that end we will also work on a toolkit to facilitate the construction of frameworks that capture Domain Architectures. The domain architecture frameworks that we are interested in are repositories of existing software components structured along goal-oriented lines. Models of the goals and plans embedded in the software components are also included in the framework. These models as well as models of the physical phenomenon controlled by the framework are then used to synthesize the glue necessary to allow this framework to interoperate with others.

A Dynamic Domain Architecture structures a domain into service layers; each service is annotated with specifications and descriptions of how it is implemented in terms of services from lower levels. Like other domain architectures, a Dynamic Domain Architecture provides multiple instantiations of each service, with each instantiation optimized for different purposes. Thus, it serves as a well-structured software repository. Typically, the application is a relatively small body of code utilizing the much larger volume of code provided by the framework. Typical domains of concern for military embedded software systems include sensor management, navigation guidance and control, electronic warfare, etc.

A Dynamic Domain Architecture is, however, different from the domain architectures developed in earlier DARPA programs (e.g. STARS and DSSA). In earlier systems, the Domain Architecture was a static repository from which specific instantiations of the services were selected and built into the run-time image of the application. Neither the models nor the deductions used to select specific instantiations of the services are carried into the runtime environment. In a Dynamic Domain Architecture, however, all the alternative instantiations, plus the models and annotations describing them are present in the run-time environment, and multiple applications may simultaneously and dynamically invoke the services.

Dynamic Domain Architectures allow late binding of the decision of which alternative instantiation of a service to employ. Like Dynamic Object Oriented Programming (DOOP) systems, the decision may be made as late as method-invocation time. However, Dynamic Domain Architectures go further than DOOP, allowing the decision to be made using much more information than simple type signatures. The models which describe software components are used to support runtime deductions leading to the selection of an appropriate method for achieving a service.

Dynamic Domain Architectures recognize that in many open environments (e.g., image processing for ATR) it isn't possible to select the correct operator with precision, *a priori*. Therefore, Dynamic Domain Architectures support an even later binding of operator selection, allowing this initial selection to be revised in light of the actual effect of the invocation. If the method chosen doesn't do the job well enough, alternatives selections are explored until a satisfactory solution is found or until there is no longer any value to be gained in finding a solution.

Dynamic Domain Architectures remove exception handling from the purview of the programmer, instead treating the management of exceptional conditions as a special service provided by the run-time environment. The annotations carried forward to run time include formal statements of conditions which should be true at various points of the program if it is to achieve its goals. The DDA framework generates runtime monitoring software that invokes error-handling services if these conditions fail to be true. The exception-management service is informed by the Dynamic Domain Architecture's models of the executing software system and by a catalog of breakdown conditions and their repairs; using these it diagnoses the breakdown, determines an appropriate scope of repair and possibly selects an alternative to that invoked already; it then restarts the computation.

Finally, a DDA framework provides an embedded language in which the developers of other frameworks can access its state-variables and influence its goal and plan structure.

C.5.1 Domain modeling

The idea of domain architecture dates back to the Arpa Megaprogramming initiative where it was observed that software reuse could best take place within the context of a Domain Specific Software Architecture. Such an architecture would identify important pieces of functionality employed by all applications within the domain, and would then recursively identify the important functionality supporting these computations. In a visual-interpretation domain, for example, typical common functionality might include region identification, which in turn depends on edge detection, which in turn depends on filtering operations (eg, convolutions).

This process of identifying and structuring the common functionality is the first component of a process termed Domain Analysis. Domain Analysis structures common functionality into a series of "service layers," each relying on the ones below for parts of its functionality. The second component of Domain Analysis is the identification of variability within the commonality. Returning to our visual-interpretation example, there are several different approaches to region identification, dozens of distinct edge-detection algorithms, and many different ways to perform filtering operations. When looked at in even finer detail, there may be an even greater number of variant instantiations of any of these operations. Variations arise due to different needs for precision, time and space bounds, error management, and the like.

The power of Domain Analysis is that its identification of common functionality lets one view the code in a new terms: the bulk of the code is in the service-layer substrate and implements functionality common to many applications. Each application consists of a thin veneer of application-specific code, riding on top of this substrate of service layers. However, the substrate contains many variant instantiations of each service. Although each instantiation is relevant to only some of the applications, Domain Analysis lets us see these as variants of a common conceptual service. Before the Domain Analysis was performed, each application stood alone using its particular instantiations of the common services and was ignorant of the fact that other applications used the same conceptual services but with different instantiations.

C.5.2 Dynamic Object Oriented Programming

The Lisp community, with its close connection to artificial intelligence research, has independently discovered some of the same ideas, but has packaged them in a more dynamic but less formal framework. This approach was first identified and termed "super-routines" in a paper by Eric Sandewall [San79, SSS81]. Sandewall noted that it is often the case that a whole class of computations are instances of some very general pattern of computation, where the members of the class differ only in the details. He termed this higher-level structure a "super-routine" and noted that data-driven programming techniques could dynamically determine which subroutine was relevant at run time. As object-oriented programming ideas developed in the Lisp and Smalltalk communities, several researchers began to understand that the Dynamic OOP facilities common in these languages were exactly what was need to build a super-routine.

The high-level common services of a Domain Architecture are precisely the same idea as Sandewall's notion of a super-routine (rediscovered in another context by another community a decade later). Unlike the Static Domain Architectures, the runtime environment of the systems Sandewall characterized all included many variant instantiations of the common services, and dynamically invoked a particular instantiation based on run-time conditions.

The mapping between the super-routine (or Domain Architecture) idea and the features of DOOP is straightforward: each high-level abstract operation (or Domain Architecture service) is identified with a generic function; the different instantiations are provided by different methods, each with a unique type signature. Method invocation performs the dynamic run-time selection of the appropriate instantiation of the service. This style of building extensible, domain specific architectures

Figure 2: The DDA uses Domain Models to Integrate Frameworks

has become known as “open implementation” [Kic96].

Dynamic OOP also provides significant facilities for managing exceptional conditions. In the case of Lisp, these facilities were motivated by the needs of adaptive planning systems. In particular, the facilities provided to signal exceptional conditions allow the error-handling code access to the environment of the exception and this, in turn, allows the handler to characterize the nature of the breakdown. Facilities similar to the signalling of the exception are used to transfer control from the error handler to an appropriate “restart” handler. Once the error handler has characterized the nature of the breakdown, it invokes a repair mechanism, not by name, but by description. Finally, the language provides facilities to specify what cleanup work must be done to perform the appropriate recovery work as rollback to the restart position takes place.

What is needed is to enrich this infrastructure with extensive models of the software’s structure, function and purpose and to build in facilities for noticing if an operator has failed to achieve its purpose. We must carry into the run-time environment all the descriptive information as well as all the variant instantiations of operators present in the development environment, and we must use this information reactively to control the physical system in which the software is embedded. We call such a framework a Dynamic Domain Architecture because it incorporates and extends ideas from the two traditions of Domain Architectures and Dynamic Object Oriented Programming.

C.5.3 Services Provided

A Dynamic Domain Architecture is far more introspective and reflective than conventional software systems. This allows many tasks which currently burden the programmer to instead be synthesized from the models within a framework or to be provided as system services. As shown in figure 2 this includes:

1. The synthesis of code that selects which variant of an abstract operator is appropriate in light of run-time conditions.

2. The synthesis of monitors which check that conditions expected to be true at various points in the execution of a computation are in fact true.

3. Diagnosis and isolation services which locate the cause of an exceptional condition, and characterize the form of the breakdown which has transpired.

4. Alternative selection services which determine how to achieve the goal of the failed computation using variant means (either by trying repairs or by trying alternative implementations, or both).

5. Rollback and recovery services which establish a consistent state of the computation from which to attempt the alternative strategy.

6. Allocation and reoptimization services which reallocate resources in light of the resources remaining after the breakdown and the priorities obtaining at that point. These services may optimize the system in a new way in light of the new allocations and priorities.

7. The synthesis of connections to reactive executives that manage physical components of concern to the DDA framework.

8. The synthesis of connections to other DDA frameworks with whose state-variables, goals and plans the current framework interacts.

C.6 Summary

The primary *technical innovation* of our proposed research is to reframe the entire endeavor of building embedded system software. We break with the tradition of working within a framework of impoverishment. The resources are now available to allow extremely high level, modular views of the system to be expressed and executed directly. Computational abundance also allows us to stop trying to deduce all possible contingencies at design time; rather we believe the resources and techniques are already present to allow us to rely on runtime deduction to characterize the state of the system and to plan appropriate reactions.

We divide the problem into several distinct tasks. The application developer expresses the core functionality of the composite system. Some framework designers provide solutions to specific cross cutting issues, while others encapsulate the knowledge of an entire domain architecture in a framework.

Each framework designer exposes to the other developers an embedded language which allows programmers to couple their functionality to the infrastructure provided by the framework. Each framework designer also provides a guarantee that certain properties will be maintained as long as the protocol of the frameworks is obeyed and as long as a small set of other constraints are obeyed by the rest of the program. The framework designers provide specialized analysis and reasoning tools to help the application developer conduct an analysis of the total system.

Our task is to provide the expressive power needed to state solutions in a clear, modular and evolvable fashion and to develop and employ implementation techniques that allow high performance, reactive execution of the resulting system. We provide model-based techniques for the generation of a total system from an ensemble of frameworks, some representing domain specific architectures and others representing cross cutting aspects. Model-based techniques also support reactive diagnosis and reconfiguration of the composite system. Finally, we provide the designers of applications and frameworks with an extremely rich collection of models, analysis tools, reasoning environments and integration techniques that allow the analysis of a software system to be performed in a way that mirrors the composition of its implementation.

C.7 Research Plan

Our project will design and implement two major packages: A Reactive Executive for Model-Based Autonomy based on our work on Livingstone and Remote Agent [MNPW98, B^+98, WN97, NW97, WN96] and a Developers Toolkit for DDA frameworks based on our work on dynamic domain architectures. We have three areas of potential application: vehicle autonomy, perceptually enabled user interfaces (as in our Intelligent Room and MIT's Projet Oxygen) and Autonomous Image Understanding in which we are anxious to apply these results. In these areas we have ready made testbeds and experimental setups. However, we wish to drive all of these efforts through collaborations with other MOBIES contractors in which we attempt to use our tools to implement their frameworks and apply the results to both their and our own applications. Interleaved in all these efforts will be experimentation and integration tasks (and quite conceivably modifications of the workplan based on experience).

Our research plan is driven by an experiment: we will attempt to build a few core frameworks and related Model-based Reactive Frameworks. We will then test whether an application can be implemented using these frameworks and analyzed using the frameworks' analysis tools. In the initial experiments, our group will be implementing all the framework and application code. If the initial efforts are promising, we will then attempt a second series of experiments in which other researchers use our toolkits.

D Deliverables

The results of this work will be in the form of software, representation languages, and knowhow. The software will include an implementation of a new reactive executive for model-based autonomous systems. It will also include a prototype Developers' Toolkit illustrating how the underlying language can be used to facilitate DDA's for autonomous embedded systems. Finally, we will deliver one or more prototype implementations of frameworks built using these tools. The representation language will be a prototype system useful for describing and reasoning about program properties relevant to embedded systems and for generating integrated applications from an ensemble of frameworks.

These languages, and the knowhow that goes with them, will be described in technical papers and manuals, illustrated with examples from our work, and applied to tasks of interest to DARPA customers.

There will be no proprietary claims to results, prototypes, or systems.

Anticipated deliverables include:

1. A meta-language appropriate for describing properties of the of embedded systems that are manipulated by each framework. This language center around the description of goals, plans, strategies, and state-variables.

2. A next-generation reactive executive for model-based autonomous systems based on our experience with the Livingstone system.

3. A Developers' Toolkit useful for implementing and reasoning about and composable DDA frameworks.

4. A generator of composite applications that composes an ensemble of DDA frameworks into an integrated application.

5. Examples of use and written descriptions at each level.

E Statement of Work

Our work will proceed in three tracks. The first, Reactive Executives for Model Based Autonomy (MBA) is concerned with the runtime environment. The second, Development Toolkit for MBA is concerned with the tools available at software development time. The third, Integration, Experimentation and Evaluation, is concerned with the experimental evaluation of all of our technology through compelling prototype applications. The runtime support is the Remote Agent Executive, and the design time support is the DDA Development Toolset. Applications include the Intelligent Room, Adaptive Autonomous Image Understanding, and two testbeds available through the Aero-Astro department. The first is a testbed consisting of autonomous land vehicles and mini-helicopters; the second is a separated spacecraft interferometer testbed.

Each track structures its tasks across 4 fiscal years, beginning in FY'00 and continuing through FY'03. Work is assumed to begin 5/1/00.

E.1 Reactive Executives for Model Based Autonomy

The major subtasks for REMBA are extending the DDA runtime infrastructure to provide support for Reactive Executives; extending the Reactive Executive to support distributed, cooperative model based autonomy; and the metamodeling of the relationship between reactive executives across levels of operation and across domains.

FY'00

1.0.1: We will redesign the Livingstone kernel along the lines of the the DDA framework.
1.0.2: We will study how to create generalized versions of the Livingstone executive capable of supporting the operations of an Intelligent Room testbed, and of a Separated Spacecraft testbed.

FY'01

1.1.1: We will recode the Livingstone kernel in the DDA framework.
1.1.2: We will develop semantic models for the primitives of distributed cooperating autonomous systems.
1.1.3: We will create generalized versions of the Livingstone executive capable of supporting the operations of an Intelligent Room testbed, and of a Separated Spacecraft testbed.

FY'02

1.2.1: We will implement a new framework integrating the reactive planner and the reconfiguring adapter.
1.2.2: We will develop a reactive execution kernel (RE) by extending Livingstone to support the RMPL language.
1.2.3: We will generalize RE operations to support the prototype applications of our Intelligent Room testbed as well as those of our Separated Spacecraft testbed.

Year FY'03

1.3.1: We will reformulate RE as a collection of distributed, cooperative execution kernels.
1.3.2: We will apply models of distributed cooperative autonomy to the domain of Adaptive Autonomous Image Understanding. We will Improve these models for use in other applications.
1.3.3: We will Revise and improve generalized meta models of RE.

E.2 Development Toolkit for MBA

The major subtasks for DTMBA are to extend the tools and frameworks of DDA to cover planning, scheduling and replanning, in order to better support the Reactive Executive; adding static analysis tools to DDA; defining specific frameworks that generalize models such as Livingstone;

FY'00

2.0.1: We will develop or select embedded languages, including RMPL for expressing plans, schedules and goals, for use in DDA frameworks.
2.0.2: We will identify the crucial models used in DS1, and study how to express them as parameterized DDA frameworks.

FY'01

2.1.1: We will implement the languages selected in task 2.0.1 for use in DDA frameworks.
2.1.2: We express the crucial models used in DS1 as parameterized DDA frameworks.
2.1.3: We will define and reuse/customize analysis tools that support constraint checking, constraint propagation, and consistency checking.

FY'02

2.2.1: We will test and refine our embedded plan languages. We will develop the support connections between the components of our adaptive infrastructure (e.g. diagnosis and recovery) and our plan languages.
2.2.2: We will develop support for generating code in embedded languages from model based reactive planing frameworks.
2.2.3: We will develop tools for applying static analysis incrementally to prototyped code.

FY'03

2.3.1: We will extend our embedded plan languages to utilize explicit real-time models, as developed by ourselves and other MOBIEs contractors
2.3.2: We will package code generators in domain specific frameworks. We will improve our optimistic optimization and our recovery mechanisms.
2.3.3: We will improve our tools for static analysis of real-time properties.

E.3 Integration, Experimentation and Evaluation

The integration, experimentation and evaluation task includes integrating the diverse components of our research and incorporating the work of other MOBIEs contractors, experimenting with these tools in challenging prototype application tasks, and evaluating our progress

FY'01

3.0.1: We will study how to apply the reimplementation of Livingstone to our Intelligent Room and Separated Spacecraft testbeds.
3.0.2: We will identify complementary MOBIEs efforts and design mechanisms for integrating the results of those efforts.

FY'01

3.1.1: We will apply the reimplementation of Livingstone to our Intelligent Room and Separated Spacecraft testbeds.
3.1.2: We will begin participating in MOBIEs program-wide evaluation efforts along with partners identified in task 3.0.2

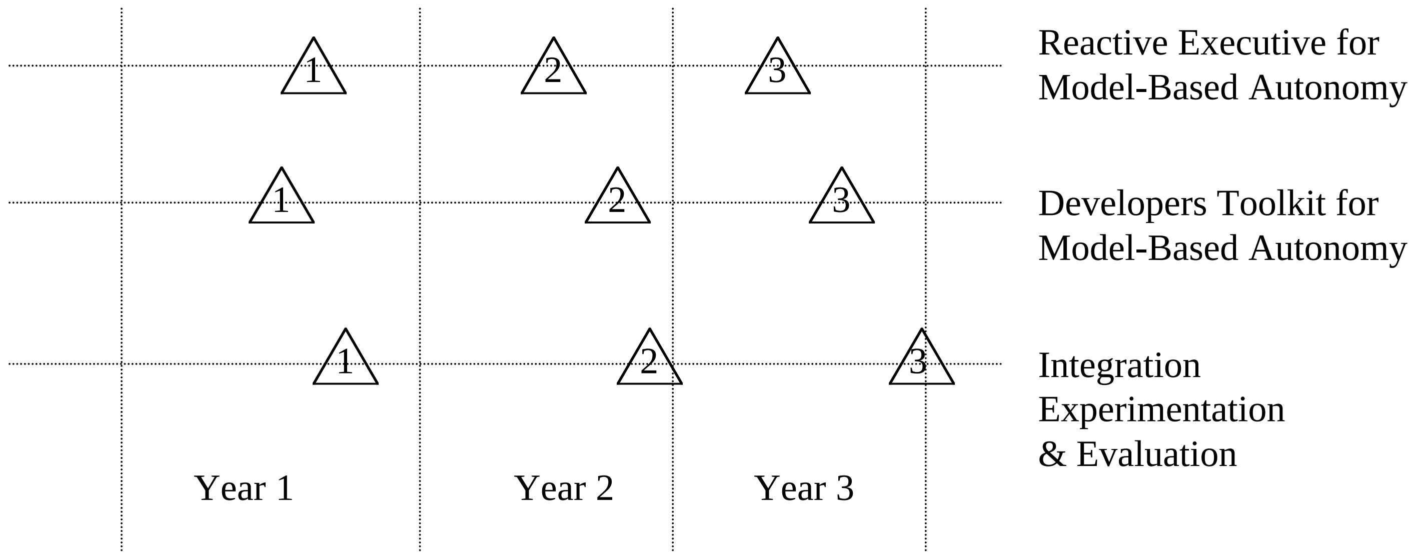
FY'02

3.2.1: We will apply the recoded and extended Livingstone Reactive Executive to the Adaptive Autonomous Image Understanding and Separated Spacecraft testbeds.
3.2.2: We will develop tools and toolkits for implementing application specific code for our three testbed application areas.
3.2.3: We will begin to integrate the results of complementary MOBIEs efforts. We will continue participating in MOBIEs program-wide evaluation efforts.

FY'03

3.3.1: We will produce a DDA framework for one of the three candidate application areas.
3.3.2: We will participate in the MOBIEs program-wide challenge problem.

F Schedule of Milestones



Reactive Executive for Model-Based Autonomy

1: Complete design of next-generation Livingstone.
2: Complete implementatin of next-generation Livingstone.
3: Complete testing and evaluation of next-generation reactive executive.

Developers' Toolkit for Model-Based Autonomy:

1: Complete the design of the meta-language for describing DDA programs.
2: Complete prototype of DDA-based Application Generator
3: Complete testing and evaluation of DDA toolkit

Integration, Experimentation and Evalution
1: Complete a prototype testbed application using Livingstone
2: Complete Application specific toolkits for testbed applications
3: Complete DDA framework for testbed applications

G Technology Transfer

The results of this work will take the form of software, representation languages, and knowhow. The software will be a collection of prototype programs including a Framework Developer's Toolkit, a Dynamic Virtual Machine and a heterogeneous reasoning system suitable for program analysis. The representation languages will provide a way of describing, and perhaps more important, thinking about the structure and behavior of large scale embedded systems. These languages, and the knowhow that goes with them, will be described in technical papers and manuals, illustrated with examples from our work, and applied to tasks of interest to DARPA customers.

MIT has a very long tradition of technology transition supported by an active Industrial Liaison Program and a Technology Licensing Office. Information Technology in general and Artificial Intelligence in particular have been actively involved in this activity. For example, the original concept of timesharing was invented by John McCarthy (then at MIT) and was then developed into CTSS and Multics. IBM, Bell Labs, GE and later Honeywell were then engaged as partners in the effort to transition Timesharing and virtual memory from the lab to common practice. We have had many similar successes over the years with direct impact on DARPA programs and the DoD. These include the original bit-mapped display, the MIT Lisp Machine, and the Connection Machine. More recently our research was carried by graduate to NASA and formed the core of the recent Mars Rover team. Several recent spinoffs (Sensible Technologies, IS Robotics, Imagen) has helped to transition other research. Finally we mention that a document distribution, indexing and retrieval system developed in our Lab has found its way into the daily operational life of the Executive Office of the President.

Our technology will be transferred outside of MIT via several paths: traditional publication of research papers; network publication of software artifacts, manuals, and papers; through our students and through collaborations with other PCES contractors.

Network publication Our research will be concretely realized in software artifacts: programming environments, run-time systems, analysis and monitoring systems, representation languages and programmer interfaces. Usable software systems are a strong way to make a point, and we intend to use the Internet to widely disseminate our technology in source-code form for general use by the global Internet community.

Students We consider our students to be important vectors for transferring technology out into the real world. Besides the graduate students whose dissertations will comprise much of the research, MIT also has a long-standing commitment to involving undergraduates in our ongoing research through the Institute's UROP program.

Other Contractors

We expect that some of the other MOBIES contractors will focus on what we have been calling "frameworks". We would enthusiastically welcome the opportunity to work with those contractors and to implement their frameworks in our infrastructure. We could then compare and contrast implementations in our system versus others and transfer an understanding of our approach to other contractors.

H Related Research

H.1 Reactive Systems

Our work in this area draws heavily from our experiences with the strength and weaknesses of Livingstone and the entire Remote Agent experiment of which it was a component [MNPW98, B$^+$98]. From that experience we developed powerful new techniques for reactive planning [WN97] and for fast propositional reasoning [NW97]. Much of our work was inspired by earlier work on Model-Based diagnosis [DS82, dW87, dW89, Dav84]. Most of the key work in this area was produced by members of this team or by our closest collaborators.

The New Millennium Remote Agent (RA) architecture is closely related to the 3T (three-tier) architecture [BKMS97]. The 3T architecture consists of a deliberative component and a real-time control component connected by a reactive conditional sequencer. RA and 3T both use RAPS [Fir78] as our sequencer, although a new sequencer which is more closely tailored to the demands of the spacecraft environment [Gat96] is under development. The deliberator is a traditional generative AI planner based on the HSTS planning framework [Mus94], and the control component is a traditional spacecraft attitude control system [HBR93]. An architectural component explicitly dedicated to world modeling (the mode identifier) was added, and the project distinguished between control and monitoring. In contrast to 3T, the prime mover in this system is the RAP sequencer, not the planner. The planner is viewed as a service invoked and controlled by the sequencer. This is necessary because computation is a limited resource (due to the hard time constraints) and so the relatively expensive operation of the planner must be carefully controlled. In this respect, the architecture follows the design of the ATLANTIS architecture [Gat92]. The current state of the art in spacecraft autonomy is represented by the attitude and articulation control subsystem (AACS) on the Cassini spacecraft [BBR95, HBR93] (which supplied the SOI scenario used in the RA prototype). The autonomy capabilities of Cassini include context-dependent command handling, resource management and fault protection. Planning is a ground (rather than onboard) function and onboard replanning is limited to a couple of predetermined contingencies. An extensive set of fault monitors is used to filter measurements and warn the system of both unacceptable and off-nominal behavior. Fault diagnosis and recovery are rule-based. That is, for every possible fault or set of faults, the monitor states leading to a particular diagnosis are explicitly encoded into rules. Likewise, the fault responses for each diagnosis are explicitly encoded by hand. Robustness is achieved in difficult-to-diagnose situations by setting the system to a simple, known state from which capabilities are added incrementally until full capability is achieved or the fault is unambiguously identified. The RA architecture uses a model-based fault diagnosis system, adds an on-board planner, and greatly enhances the capabilities of the on-board sequencer, resulting in a dramatic leap ahead in autonomy capability. Ahmed, Aljabri, and Eldred [AAE94] have also worked on architecture for autonomous spacecraft. Their architecture integrates planning and execution, using TCA [Sim90] as a sequencing mechanism. However, they focused only on a subset of the problem, that of autonomous maneuver planning. Their architecture did not address problems of limited ob servability or generative planning. Systems developed for applications other than spacecraft autonomy present some features comparable to RA. Bresina et al. [BESD96] describe APA, a temporal planner and executive for the autonomous, ground-based telescope domain. Their approach uses a single action representation whereas RA uses an abstract planning language, but their plan representation shares with ours exibility and uncertainty about start and finish times of activities. However, their approach is currently restricted to single resource domains with no concurrency. Moreover, APA lacks a component comparable to MIR for reasoning about devices. Phoenix [CGHH89] is an agent architecture that operates on a real-time simulated fire fighting domain. The capabilities provided by the agent are comparable to those provided by the RA executive although many aspects of the solution seem specific to the domain and do not appear to be easily generalizable. Unlike RA, Phoenix's agent does not reason explicitly about parallel action execution, since actions from instantiated plans are scheduled sequentially on a single execution timeline. A notable characteristic of Phoenix is reliance on envelopes [HAC90], i.e., pre-computed expected ranges of acceptability for parameters over continuous time, which are continuously monitored for robust execution. Among the many general-purpose autonomy architectures is Guardian [HR95], a two-layer architecture which has been used for medical monitoring of intensive care patients. Like the spacecraft domain, intensive care has hard real-time deadlines imposed by the environment and operational criticality. One notable feature of the Guardian architecture is its ability to dynamically change the amount of computational resources being devoted to its various components. The RA architecture also has this ability, but the approaches are quite different. Guardian manages computational resources by changing task scheduling priorities and the rates at which messages are sent to the various parts of the system. The RA architecture manages computational resources by giving the executive control over deliberative processes, which are managed according to the knowledge encoded in the RAPs. SOAR [LNR87] is an architecture based on a general purpose search mechanism and a learning mechanism that compiles the results of past searches for fast response in the future. SOAR has been used to control flight simulators, a domain which also has hard real-time constraints and operational criticality [TJJ$^+$95]. SOAR-based agents draw on tactical plan expansions, rather than using first principles planning as does RA. CIRCA [MDS93] is an architecture that uses a slow AI component to provide guidance to a real-time scheduler that guarantees hard real-time response when possible. CIRCA can make tighter performance guarantees than can RA, although CIRCA at present contains no mechanisms for long-term planning or state inference. Noreils and Chatila [NC95] describe a mobile robot control architecture that combines planning, execution, monitoring, and contingency recovery.

Their architecture lacks a sophisticated diagnosis component and the ability to reason about concurrent temporal activity and tight resources. The Cypress [WMLW95] architecture combines a planning and an execution system (SIPE-II [Wil88] and PRS [GL87]) using a common representation called ACT [WM95]. This serves as an example of a unified knowledge representation for use by heterogeneous architectural components. Cypress is similar to RA, and unlike most other architectures, in that it makes use of a component for sophisticated state inference, which corresponds to RA's MI component. A major difference between Cypress and RA is our use of an interval-based rather than an operator-based planner. Drabble [Dra93] describes the Excalibur system, which performs closed-loop planning and execution using qualitative domain models to monitor plan execution and to generate predicted initial states for planning after execution failures. The "kitchen" domain involved concurrent temporal plans, although it was simplied and did not require robust reactions during execution. Currie and Tate [CT91] describe the O-Plan planning system, which when combined with a temporal scheduler can produce rich concurrent temporal plans. Reese and Tate [RT94] developed an execution agent for this planner, and the combined system has been applied to a number of real-world problems including the military logistics domain. The plan repair mechanism [DTD96] is more sophisticated then that of RA, although the execution agent is weaker and does not perform execution-time task decomposition or robust execution.

H.2 Dynamic Domain Architectures

Our ideas on Dynamic Domain Architectures stem from several trends in AI and software engineering. The idea that programs can be viewed as instances of plans and that they exhibit a goal-directed structure dates back at least to the work on the Programmer's Apprentice [RS76, Shr79, Ric81]. DDA frameworks build in diagnostic services which draw on the work on model-based diagnosis [DS82, dW87, dW89, Dav84]. The idea of a formalizing a domain architecture and even the idea of dynamic invocation go back to a little referenced but nevertheless seminal paper by Sandewall [San79]. There is a host of more modern work similar in spirit to ours. These include ideas of decomposing systems in cooperating frameworks, using advanced object oriented techniques to build integration techniques, generating the integration code that links frameworks in to larger ensembles.

We briefly survey some of that work below:

H.2.1 Subject-Oriented Programming

Subject-oriented programming [HO93] allows the natural specification of an application in terms of the composition of domain and task specific decompositions. The decompositions can cut across normal object-oriented organization involving partial definitions of classes and methods. These decompositions address subject-oriented design paradigms such as product lines, evolutionary development, and multi-team collaborations. Subject-oriented programming permits a user to specify the composition of these separate components as a set of rules for managing their combination and resolving conflicts.

The limitations of pure object-oriented programming have been recognized for a longtime in the LISP community. The Common Lisp Object System (CLOS) provides the ability to decompose applications along object and procedural lines. For example, multimethods allow a developer to define a set of methods on a particular class or set of classes outside the usual class definition. This permits additions or modifications to object-oriented programs to be specified as separate files or libraries in language without resorting to outside compositional support. Mixin classes, method combination, and runtime namespaces increase the compositional power even further. Finally, procedural macros provide one more tool for specifying behavior that cross-cuts the usual object or procedural boundaries without tangling the source.

H.2.2 Product Lines, Scalable Libraries, and Software Generators

A product-line architecture (PLA) [BS99] is a design for a family of similar applications. A generator is a tool that takes a specification for a composition of scalable libraries and produces a high-performance application. In [SB98], the authors propose an object-oriented building block called a mixin-layer. The idea here is that often additions to software are not localized to a single class but instead span several classes. They show how to compose these layers using an implementation of the Gen Voca theory [BO92].

Our work demonstrates another set of techniques for implementing scalable libraries. In contrast to their work, our CLOS-style substrate is not as constrained as their object-centric foundation (e.g., C++). For example, multimethods can already be added outside of class definitions. On the other hand, one strength of their approach is the view that libraries are composed to build applications. We would like to research more mechanisms for specifying the composition of our frameworks.

H.2.3 Aspect-Oriented Programming (AOP)

Aspect-oriented programming [KLM$^+$97] is a style of programming that complements object-oriented and procedural programming by providing an alternative decomposition of a program into features of a particular domain, called aspects, that cross-cut multiple classes and/or procedures. These aspects can then be merged with the object-oriented and/or procedural parts of a program using a weaver to form an application.

AOP is a generalization of subject-oriented programming in that it goes beyond merely combinations of aspects, but addresses semantic and performance issues as aspects in their own right.

H.2.4 AspectJ

AspectJ [LK98] is an implementation of AOP for Java. It provides a mechanism for combining methods and fields defined in separate aspects. In particular, advise methods can be combined with existing methods and fields can be added. Aspects can be easily plugged in or out of applications by invoking a weaver at compile time. Unfortunately, this is a static operation. Pattern matching is used to specify the domain of the aspects, that is, the methods to which method combination is to be applied.

We maintain that AspectJ is limited in a number of key areas. First, aspects can be woven in only at compile time. Second, the forms of method combination are limited. Third, the system is not user extensible. Fourth, the semantics are opaque. In contrast, our meta-object approach, overcomes all of these limitations.

H.3 Reflection and Metaobject Protocols

Reflection and metaobject protocols [KdR91] are a powerful way to implement AOP. A reflective language is embodied in a base language and several meta languages which control the semantics and implementation of the given base language. The meta languages provide hooks that allow a user to implement cross cutting functionality, functionality to which no single base language has access.

I Personnel

Brian C. Williams is an Associate Professor of in the MIT department of Aeronautics and Astronautics; he is also a member of the MIT Artificial Intelligence Laboratory. He has done research in the Development and formalization of model-based reasoning methods for the control and synthesis of highly autonomous systems. From 1997 - 1999 he served as Lead, NASA Autonomy and Information Management Program and as the Technical lead for NASA's long term, crosscutting information technologies program, including NASA Ames, Goddard, Johnson, Marshall and JPL. He was one of the leaders of the team at NASA Ames which developed Remote Agent the first model- based autonomous control system for a spacecraft with significant onboard deduction. Remote Agent was deployed on the Deep Space 1 vehicle. This project included the development of real-time executives that use onboard models to perform goal-directed commanding and monitoring, fault detection, diagnosis, reconfiguration and fault recovery. Dr Williams received his PhD (1989), MS(1984) and BS(1984) from MIT.

Robert Laddaga, Ph.D., is a research scientist at the M.I.T Artificial Intelligence Laboratory. He has done research in Software Engineering, Artificial Intelligence, Programming Languages, Cognitive Psychology, and Intelligent Tutoring. He served as a Program Manager in the DARPA Information Technology Office (ITO) from December 1996 through February 1999. In this capacity he had a significant role in managing several programs including EDCS and Information Survivability, and in creating the ASC and ANTs programs. He has also served as Director of Software Development at Symbolics Inc, and President of Dynamic Object Language Labs, and was an Assistant Professor of Computer Science at the University of South Carolina.

Howard Elliot Shrobe, Ph.D., is Associate Director of the M.I.T Artificial Intelligence Laboratory. He has done research in Software Engineering, VLSI design, Computer Architecture and Artificial Intelligence. He served as Chief Scientist of the DARPA Software and Intelligent Systems Technology Office (SISTO) and of the Information Technology Office (ITO) from September 1994 through August 1997. In this capacity he had a significant role in the creation of several programs including EDCS and Information Survivability. He has also served as Chief Technology Officer of Symbolics Inc. He served as Chair of the AAAI Conference Committee for 5 years and is a Fellow of the AAAI.

Gregory Timothy Sullivan, Ph.D., is a research scientist at the M.I.T. Artificial Intelligence Laboratory. He has done research in Software Engineering, the formal semantics of programming languages, and dynamic language implementation. Both before and after his graduate work, he worked in a variety of software companies, working first on CASE (Computer Aided Software Engineering) tools and later on the dynamic object-oriented language Dylan.

Jonathan Richard Bachrach, Ph.D., is a postdoctoral researcher at the M.I.T. Artificial Intelligence Laboratory. He has done research in dynamic language implementation, language design, compiler design, computer architecture, neural networks, and performance evaluation. He played an important role in the design of the dynamic object-oriented language Dylan and was the manager of Harlequin's Dylan team and one of its key compiler and runtime developers. He was also was involved in the design of the object-oriented language Sather while a postdoctoral researcher at U.C. Berkeley,

J Facilities

All work will be performed at the MIT Artificial Intelligence Laboratory and at the MIT Department of Aeronautics and Astronautics. The AI Laboratory and the Aero-Astro department are equipped with a variety of computers and computational equipment, ranging from PCs and Macs to workstations, Silicon Graphics systems, to a number of machines of our own design (e.g., Lisp machines), along with access to supercomputers within the building. Administratively backed up large file servers run on local workstations; Internet access is available through the MIT spine via NearNet and is being upgraded to a 45Mbit data rate.

The AI Laboratory also maintains extensive mechanical and electrical prototyping facilities, allowing rapid construction of systems. A complete machine shop and sheet metal prototype fabrication facility is available, along with stocks of small electrical components, connectors, wire, etc. All of this enables us to make quick mock-ups of proposed designs.

The Laboratory provides an environment of excellent students interested in advanced software technology, reasoning, planning, and decision-theoretic representation and inference.

The department of Aeronautics and Astronautics has several testbeds useful for experimentation with embedded software systems. These include an autonomous vehicle testbed involving mobile ground vehicles and mini-helicopters and a separated spacecraft interferometry testbed.

We anticipate no need for Government Furnished Equipment.

K Experimentation and Integration

Our research plan is driven by experimentation: We will develop several simple, prototype DDA frameworks as well as related Reactive model-based executives. We will then test whether a robust application can be implemented using these frameworks and executives. We will measure the amount of high level code needs to be written in the embedded languages of our frameworks in our to faciliate the generation of an integated application. We will also measure the degree of robustness achievable in the applications.

Such experiments ultimately require one or more applications to drive them. We note that we have access to several interesting application areas with ready made testbeds. The first of these is vehicle autonomy; in particular, we have access to two interesting testbeds in the MIT Aero and Astro department. The first, a vehicle autonomy testbed, consists of multiple ground vehicles and mini-helicopters. The second is a separated spacecraft interferometry testbed.

A second application area is in the area of perceptually enabled systems. The MIT Intelligent room is such a system, combining active machine vision (using several cameras) and speech understanding. More generally, MIT Project Oxygen is a joint effort of the MIT AI Lab and the MIT Lab for Computer Science; a significant component of Oxygen involves the use of perceptually enabled human computer interfaces to embed computation in the normal work and living environment of its users. Such systems are envisioned to be ubiquitous: they may be embedded in building environments or they may be mobile personal devices. We and our colleagues are key contributors to this project.

It is also a significant component of our plan that we attempt to implement the frameworks of other group and that we demonstrate how these independently developed frameworks can cooperate. We plan to begin our efforts by identifying at least two groups interested in developing frameworks; we will then work with these groups to investigate how their designs can be realized using our tools. In the initial experiments, our group will be implementing all the framework and application code. If the initial efforts are promising, we will then attempt a second series of experiments in which the framework designers use our toolkits.

There are two key hypotheses at the core of our experiments. First we believe that model-based deduction can be made fast enough to function within the reactive time frames of the executive and that such reasoning capabilities will drastically increase the robustness and autonomy of the system. The second hypothesis is that model-based DDA frameworks can be integrated with one another and with the reactive executives by using models of the software's goal and plan structure as well as models of the physical systems to guide the automatic generation of the composite system. We will devise both quantitative and qualitative metrics to assess progress and to help steer our developments. Also, by collaborating with potential users of such tools we believe that we can measure the effectiveness of such tools by comparing even such coarse measurements as number of lines of code, and the time and the effort that a framework developer must expend using our approach as opposed to others.

Additional material

Organizational Conflict of Interest

In compliance with Federal Acquisition Regulation (FAR) Subpart 9.5, Organizational Conflict of Interest, we state affirmatively that no member of the offering team is supporting any DARPA technical office through an active contract or subcontract.

References

[AAE94] A. Ahmed, A. S. Aljabri, and D. Eldred. Demonstration of on-board maneuver planning using autonomous s/w architecture. In *8th Annual AIAA/USU Conference on Small Satellites*, 1994.

[B+98] D. Bernard et al. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the IEEE Aerospace Conference*, 1998.

[BA93] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice Hall, 1993.

[BBR95] G.M. Brown, D.E. Bernard, and R.D. Rasmussen. Attitude and articulation control for the cassini spacecraft: A fault tolerance overview. In *14th AIAA/IEEE Digital Avionics Systems Conference*, November 1995.

[BESD96] John Bresina, Will Edgington, Keith Swanson, and Mark Drummond. Operational closed-loop obesrvation scheduling and execution. In Louise Pryor, editor, *Proceedings of the AAAI Fall Symposium on Plan Execution.* AAAI Press, 1996.

[BKMS97] R. P. Bonasso, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *JETAI*, 9(1), 1997.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, October 1992.

[BS99] D. Batory and Y. Smaragdakis. Object-oriented frameworks and product-lines. *Submitted for publication*, 1999.

[CGHH89] P.R. Cohen, M.L. Greenberg, D.M. Hart, and A.E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, 1989.

[CT91] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, December 1984.

[Dra93] B. Drabble. Excalibur: A program for planning and reasoning with processes. *Artificial Intelligence*, 62(1):1–40, July 1993.

[DS82] Randall Davis and Howard Shrobe. Diagnosis based on structure and function. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 137–142. AAAI, 1982.

[DTD96] Brian Drabble, Austin Tate, and Jeff Dalton. O-plan project evaluation experiments and results. Technical Report Oplan Technical Report ARPA-RL /O-Plan/ TR /23 Version 1, AIAI, July 1996.

[dW87] Johan deKleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[dW89] Johan deKleer and Brian Williams. Diagnosis with behavior modes. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989.

[Fir78] R. James Firby. Adaptive execution in complex dynamic worlds. Technical report, Yale University, 1978.

[Gat92] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of AAAI-92*. AAAI Press, 1992.

[Gat96] Erann Gat. Esl: A language for supporting robust plan execution in embedded autonomous agents. In Louise Pryor, editor, *Proceedings of the AAAI Fall Symposium on Plan Execution*. AAAI Press, 1996.

[GL87] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. Technical Report Technical Report 411, Artificial Intelligence Center, SRI International, January 1987.

[HAC90] D.M. Hart, S.D. Anderson, and P.R. Cohen. Envelopes as a vehicle for improving the efficiency of plan execution. Technical Report COINS Technical Report 90-21, Department of Computer Science, University of Mas- sachusetts at Amherst, 1990.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[HBR93] J. Hackney, D.E. Bernard, and R.D. Rasmussen. The cassini spacecraft: Object oriented flight control software. In *1993 Guidance and Control Conference*, 1993.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428, 1993.

[HR95] Barbara Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence*, (72), 1995.

[KdR91] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ(tm). In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 1998.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1987.

[MDS93] David Musliner, Ed Durfee, and Kang Shin. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[MNPW98] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote agent: to boldly go where no ai has gone before. *Artificial Intelligence*, (103), 1998.

[MP92] Z. Manna and A. Pneuli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[Mus94] N. Muscettola. Hsts: Integrating planning and scheduling. In Mark Fox and Monte Zweben, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[NC95] Fabric Noreils and Raja Chatila. Plan execution monitoring and control architecture for mobile robots. *IEEE Transactions on Robotics and Automation*, 1995.

[NW97] P.P. Nayak and B.C. Williams. Fast context-switching in real-time propositional reasoning. In *Proceedings of AAAI-97*, 1997.

[Ric81] Charles Rich. Inspection methods in programming. Technical Report AI Lab Technical Report 604, MIT Artificial Intelligence Laboratory, 1981.

[RS76] Charles Rich and Howard E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report Technical Report 354, MIT Artificial Intelligence Laboratory, December 1976.

[RT94] Glen Reece and Austin Tate. Synthesizing protection monitors from causal structure. In *Proceedings of AIPS-94.* AAAI Press, 1994.

[San79] Erik Sandewall. Why superroutines are better than subroutines. Technical Report LiTH-MAT-R-79-28, Linkoping University, November 1979.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming*, 1998.

[Shr79] Howard Shrobe. Dependency directed reasoning for complex program understanding. Technical Report AI Lab Technical Report 503, MIT Artificial Intelligence Laboratory, April 1979.

[Sim90] Reid Simmons. An architecture for coordinating planning, sensing, and action. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297. DARPA and Morgan Kaufmann, 1990.

[SSS81] Erik Sandewall, Claes Stromberg, and Henrik Sorensen. Software architecture based on communicating residential environments. In *Fifth International Conference on Sofware Engineering*, San Diego, 1981.

[TJJ$^+$95] M. Tambe, W. Lewis Johnson, R. M. Jones, F. Koss, J. E. Laird, Paul S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1):15–39, Spring 1995.

[Wil88] David E. Wilkins. *Practical Planning.* Morgan Kaufman, 1988.

[WM95] David E. Wilkins and Karen L. Myers. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation*, 1995.

[WMLW95] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *JETAI*, 7(1):197–227, 1995.

[WN96] B. C. Williams and P. P. Nayak. A model-based approach to reactive, self-configuring systems. In *Proceedings of AAAI-96*, 1996.

[WN97] B. C. Williams and P. P. Nayak. A reactive planner for a model-based executive. In *Proceedings of IJCAI-97*, 1997.