Domain Specific Software Design for Decision Aiding

Kirby Keller and Kevin Stanley
McDonnell Aircraft Company
McDonnell Douglas Corporation

McDonnell Aircraft Company (MCAIR) is involved in many large multi-discipline design and development efforts in the production of tactical aircraft. These involve a number of design disciplines that must be coordinated to produce a integrated design and successful product. Our interpretation of a domain specific software design (DSSD) is that of a representation or framework that is specialized to support a limited problem domain. Figure 1 contrasts domain specific vs. domain independent approaches. A DSSD is an abstract software design that is shaped by the problem characteristics. This parallels the theme of objected-oriented analysis and design[1] of letting the problem model directly drive the design. The DSSD concept extends the notion of software reusability to include representations or frameworks. It supports the entire software life cycle and specifically leads to improved prototyping capability, supports system integration, and promotes reuse of software designs and supporting frameworks.

Initial prototyping is improved if one can start development with a framework that is suited to the characteristics of the problem. This framework can be specialized as the development evolves to provide a more efficient means for the domain expert to prototype The effect is to shorten the distance between the domain expert and the working prototype by providing a domain language to state requirements and supporting automated code generation. This concept of a supporting framework can be extended to the systems level. Multi-discipline design efforts may require the integration of individual DSSDs which are critical to concurrent engineering efforts. Domain specific designs that capture problem solving representations can be leveraged in future work. These designs offer flexibility by addressing a problem domain and hence are a better starting point for reuse than particular application modules. It may also be possible to create libraries of such designs that can be matched to problem characteristics.

The example presented in this paper is the task network architecture or design which was developed for the MCAIR Pilot's Associate program. The task network concept supported both module development and system integration within the domain of operator decision aiding. It is presented as an instance where a software design exhibited many of the attributes associated with DSSD concept. The Pilot's Associate program (contract #F33615-86-C-3802) was sponsored by the Defense Advanced Research Projects Agency and administered by the United States Air Force. More recent work in this area has been performed in conjunction with McDonnell Douglas Research Laboratories and Michigan State University.

Pilot Decision Aiding Example:
As part of the Pilot's Associate (PA) program, McDonnell Aircraft (MCAIR) Company developed and demonstrated an "associate" system for tactical aircraft performing an air-to-ground battlefield interdiction mission. The demonstrated mission functionality included threat assessment, system capabilities assessment, threat reaction planning,

[1] Rumbaugh, et al, Object-Oriented Modeling and Design, Prentice Hall, 1991.

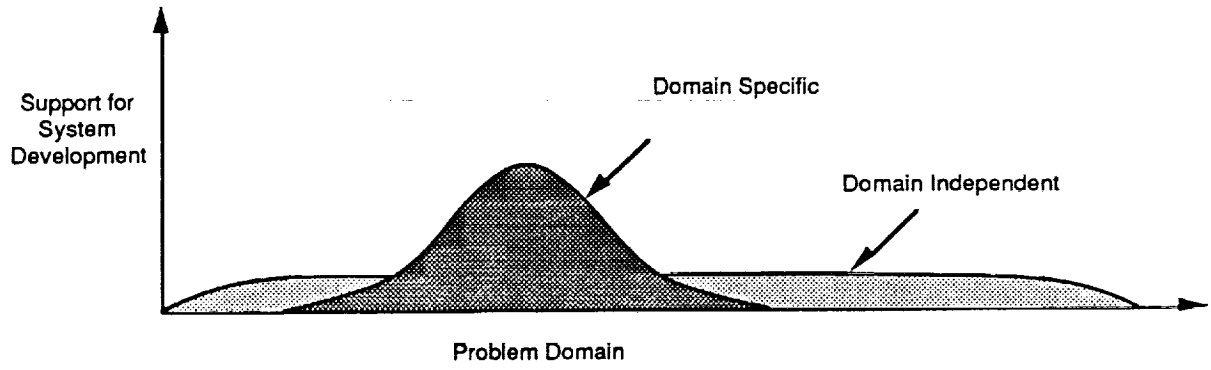

Figure 1. Domain Specific Software Design Provides Improved Support for Specific Problem Domains.

target attack planning, pilot monitoring, and information management. Appropriate controls and displays were developed to support the demonstration in a manned aircraft dome simulator. The system development approach and software architecture is based upon a task network system model. The activities of the pilot, PA and external agents such as a wingman are modelled by objects called tasks. Tasks may be decomposed into a complex sequence, or network, of more detailed subcomponents. This model of the task sequences and their functionality define a hierarchical network of tasks which allow the representation of complex system and pilot activity for both steady state behavior and reaction to changes in the environment. It captures dependencies and interactions between activities and provides a means for overall control of the PA problem solving process. The structure derived from the task network system model provides: 1) a domain specific requirements language or representation that is shared by the domain expert and software developer, 2) data structures and frameworks for the software design, and 3) visibility into the system behavior that helps create a more intuitive interface and system operation.

The top-level architecture of the task network framework is shown in Figure 2. The main components of the framework are: input packet post-processing, the context model, the task network mission model, exception handling and task execution. Data flow in this architecture consists of communication from external processes through packet post processing which appropriately manipulates the data to update objects in the context model. Events are signalled to the task network mission model, resulting in changes in task status or the execution of an exception handler. When tasks are activated they are placed on a task agenda and executed in order of priority. The execution of tasks may result in the modification of internal models (internal actions) or the communication of data to other processes (external actions).

The task network is modelled after the procedural network structure, first proposed in the NOAH system[2]. The partially ordered sequence of tasks in the network identifies control flow and context information for the state of the mission. Through an explicit representation of system and pilot tasks, the system may reason about it's own

[2] Sacerdoti, Earl D., A Structure for Plans and Behavior (Elsevier: Computer Science Library, 1977).

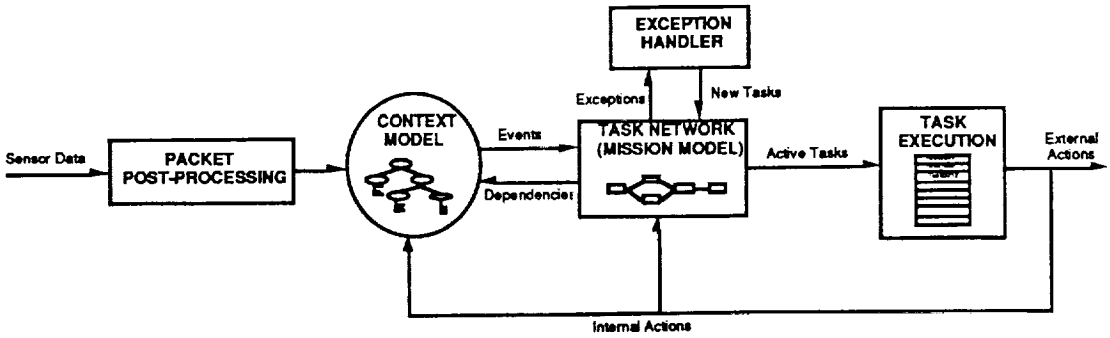

Figure 2. Task Network Top-Level Architecture

activities. Typically, this reasoning involves predicting system timeliness, interactions between tasks, errors of omission by an external agent (e.g. a pilot), the information requirements of the pilot, or responses to failure conditions of the task. Each task is represented as a specialist responsible for performing a function when activated.

Tasks are defined in a hierarchical manner such that they may be decomposed into subtasks which refine the activities that they represent. This is useful for reasoning about tasks at different levels of abstraction in monitoring, planning, and execution. The task network provides mechanisms for:

1) system coordination,
2) maintaining assumptions about the environment,
3) handling exceptions, and
4) representation of interaction with the pilot.

The task network allows dependencies to be placed on states of the environment, the pilot, and the PA system through a subset of the task network framework referred to as the Context Model. The Context Model is developed as an hierarchical, distributed, object-oriented database. It is used to represent information about the external environment, the pilot, the aircraft, and the PA system itself. This representation was designed to allow the detection of events from state data. Since PA has relatively little control over actions in the external environment (e.g. hostile threats, weather, etc.), it must make many assumptions during plan generation and execution. PA must be able to adapt quickly and correctly when the external environment changes in a way that invalidates the planned system behavior. Dependencies allow the tasks to represent complex relationships between the tasks and the state of the environment (i.e. the context of the current situation). These dependencies are checked when changes are made to the Context Model parameters. When dependencies are violated, this is signalled to the task. This signal is referred to as an exception.

Exceptions represent violated dependencies which require a response by the system. This response is referred to as an exception handler. These exception handlers are defined for tasks to aid in the recovery of violations in the assumptions of the plan. Exception handlers may be either local or system exception handlers. Local handlers are implemented using methods on the task which result in minor, local changes to the plan or states of various systems. System handlers involve the creation of System Response Plans which use the task network framework as a control mechanism for replanning portions of the currently executing task network.

The task network architecture is a domain specific design in that it is a framework

that provides support for requirements specification, design, and development at the module and system level. The benefits of the task network architecture are realized from a set of features which aid in the development of an application which lies in the real-time decision support domain. The components of the architecture support system integration by providing a uniform representation of the elements of the domain. These components and their inter-relationships were developed to address the requirements of the PA domain but it has been implemented as a explicit framework that is readily applicable, in part or in whole, to problems with similar characteristics. These features are described in the following sections.

Explicit Representation of System Plans

The requirements of the PA system are often described in terms of the aircraft mission. This mission description includes the objectives of the pilot and his weapon system in a hostile and uncertain environment. Mission decomposition is usually performed using a number of representative scenarios. This mission decomposition is a key characteristic of the domain. Mission decomposition is a top-down approach for dissecting a combat mission into its functional segments. These functional segments are then divided into the tasks which are required to complete each segment. As functions and tasks become more specific, they can be analyzed in terms of information flow and functional partitioning. The task network supports this specification through it's explicit representation of the sequence of tasks in the mission.

The explicit representation of functions as tasks in the system provides advantages in software design by supporting graceful adaptation through reasoning about task timeliness, the explicit representation of parallelism in task execution, by promoting modular coding techniques, explicit control synchronization between tasks, and visibility into system operation through the use of mnemonic names for tasks.

Enables Control Reasoning

Completing tasks by their assigned deadlines is the very definition of a hard real-time system. However, the character of the Pilot's Associate prompted us to expand the definition to include the concepts of both hard and soft deadlines. While meeting hard deadlines is a requirement for correctness, meeting soft deadlines is not strictly required, but is certainly desirable. Control reasoning is useful for a decision support system which is attempting to optimize its performance outside of hard scheduling constraints. The system may predict missed deadlines, delete unnecessary steps to meet imminent deadlines, and perform reasoning about solution quality/timeliness trade-offs. Control reasoning is also supported by the management of system priorities on tasks.

Supports Coordination and Cooperation

Knowledge partitioning is a natural and inevitable approach to the design and development of large systems. The PA system was partitioned into modules, each of which is a knowledge based system with the possibility of concurrent execution. While concurrency may not be utilized physically, the components of the PA are intended to operate in a functionally distributed fashion. Functional distribution, in this context, merely means that the components are designed to allow the possibility of concurrent operation. Each component is a real-time system. That is, each component receives events and data asynchronously and carries out steps of assessment, planning and execution, all constrained by timing requirements. For such a collection of real-time knowledge based systems to form an integrated system, they need to behave in a coordinated manner that is also timely, responsive, and adaptive to a changing environment. Coordination refers to a system-wide coherence among tasks and

plans, to a resource management scheme based on a global perspective, and to dynamic adjustment of tasks and plans to accommodate changes in overall system performance goals.

Opportunistic Execution of Tasks

Quite often in system design, the correct sequence of execution of system tasks is unknown. The task network representation allows the specification of incomplete temporal constraints on control flow. The non-linear plan representation allows ambiguity of task ordering. The execution of parallel tasks may be performed opportunistically and behavior is situationally dependent. This allows the system to improve and tune it's performance based on the context of the current situation.

Exception Handlers Modify Behavior Through Changes in Explicit Plans

The PA problem domain is dynamic and hostile. Subsequently, plans may be expected to be invalidated quite often. This adds complexity to the system requirements and design. The Task Network Architecture handles this through an explicit link between environmental data and tasks referred to as dependencies. Exception handlers are procedures which are implemented to respond appropriately to events. Each task is responsible for handling these events by one of several classes of reactions such as: abandoning the task, retrying to achieve the results of a task, choosing an alternate method for accomplishing the task results, or repairing the cause of the error. The complexity of exception handlers may be quite simple, or may require extensive replanning of the mission.

Replanning and Execution Are Interleaved

It is not possible to predict the time that events impacting the mission will be encountered. Deliberation on new plans often involves extensive processing resources devoted to solving problems encountered in the execution of plans. However, a real-time system cannot afford to halt execution while replanning is underway. Due to this, the system must be capable of replanning portions of the mission, while completing unaffected portions. The current design of the task network allows the system to inhibit the execution of tasks which are in an exception state, while continuing to execute other tasks which are unaffected by the exception.

Control Flow Manipulated Graphically

One of the tools available to software developers for managing complexity is that of graphical interfaces. The partially ordered sequence of tasks lends itself very well to a graphical depiction of the sequence of tasks performed during the mission. The implementation of tools for the graphical manipulation of tasks provides an efficient and intuitive interface for system control specification. At the same time, these tools will also provide aid in debugging the performance and functionality of the system since the current state of the system is represented pictorially through the state of the tasks in the network.

One of the key features of the task network approach is the ability to describe tasks from the perspective of a mission, and then use that same description as a foundation for code development. This philosophy is supported by the encapsulation of functionality as provided by object-oriented programming. The most efficient means of designing and modifying a task network data structure is through the use of a graphical interface which allowed for direct manipulation[3] of the task network. The task network

[3] Hutchins, E.L, Hollan, J.D., and Norman, D.A (1986). Direct Manipulation Interfaces in D.A. Norman, W.S. Draper (Eds.): User Centered System Design: New Perspectives in Human-Machine Interaction, Hillsdale London: Lawrence Erlbaum, 1986).

implementation offers a mechanism by which application code could be seamlessly integrated with code generated via these graphic descriptions.

Requirements Specification Language

The task network framework is a programming paradigm for the development of intelligent systems. The task network architecture provides support through the entire software development process, from requirements generation (specification) through maintenance as shown in Figure 3. Each module function is developed using the task network framework for planning, assessment, and human interface functions. The goal of the framework is to provide a common language between the requirements specifier, system designer, and system user. This will lead to systems which have traceable requirements in the program design and whose operation may be easily understood by the user. The program structure serves as a model of the user in performance of the mission. The network of tasks describe the sequence of tasks to be performed by the system and user. Unplanned events must also be accounted for in the system design. The design for the detection of unplanned events, the dependency mechanism, makes the conditions for plan failure explicit.

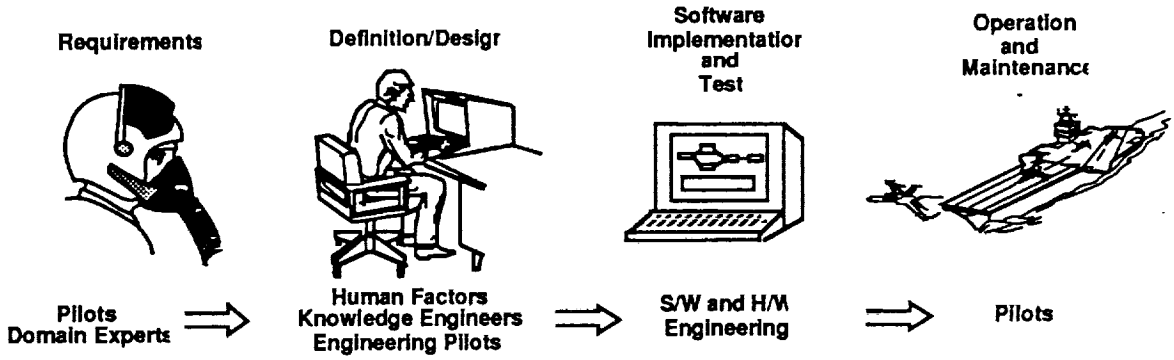

Figure 3 - Software Model Supports the System Life Cycle

The analysis of the mission results in identification of pilot and system activities as they relate to various phases of the mission. This analysis includes the identification of mission objectives, tasks that need to be performed, information required to perform the mission successfully, candidate approaches for automation and decision aiding support, and the identification of constraints imposed by combinations of the above.

Sequencing tasks in the mission identifies the context within which tasks are to be performed and the temporal constraints for efficient and effective mission performance. Through the process of mission decomposition, functional requirements are identified along with the context in which they are to be executed. This is a result of the representation of the mission sequence--mixing pilot and system activities together in a coordinated fashion.

User Activity Model

Interactive decision support systems must provide more aid than they require in user attention to the system. The primary goal for modeling the user in the task network architecture is to minimize interactions between the system and the user and thereby develop a non-intrusive, cooperative decision support framework. Through modeling the user, the system is supplied with necessary context

sensitivity to work efficiently with the user.

The pilot monitoring approach which was adopted, focussed on the state of the world represented in the Context Model, rather than on explicit pilot interaction. The approach isolates the monitor from the need to identify all methods of performing a task, all actions that may undo a given task, and explicit legal time intervals for tasks. The result is concise, robust, task-monitoring rules that can be incrementally enhanced as the Context Model grows richer. The task network represents the activities of the user by activating tasks when evidence indicates that they are being performed or have been completed. Active tasks identify activities which are being performed by the user which may be used to identify the information which is required by the user of the system. This provides a mechanism for providing both timely, and relevant information to the user.

Issues:

The major benefit of DSSD promises to be the creation of a library of reusable designs which can be classified by problem characteristics or domain to which they are applicable. An application developer could then quickly piece together a development framework from these designs. What is needed is an enumeration of the fundamental designs and a description of the range of domains that they cover.

The DSSD concept supports the notion that the initial prototyping effort should be directed at establishing or assembling a design for the particular application. This will allow leveraging the representations/frameworks associated with component DSSDs. The development of a application design based on existing DSSDs should be a goal in order to achieve system modularity, reuse, and development efficiency (eg. automated code generation).

Traditionally the press for real-time performance tends to drive designs toward system representations that are flat and efficient at the expense of rich representations which support the management of design complexity and effective interface design. It becomes a matter of development costs vs. the need for a real-time design.

The integration of DSSDs to support and integrate different design disciplines is a key to the application of the DSSD idea to large systems. In the PA example, the task network is used as a means to analyze the human factors elements of information management and automation, threat assessment, mission and tactical replanning, and as a means to determine the effect of system failures on mission activities. A focus on the concepts of DSSD should result in frameworks for integrating lower level module designs into a more coherent system design.