N93-32141

Integration of Domain and Resource-Based Reasoning for Real-Time Control in Dynamic Environments*

Keith Morgan
Kenneth R. Whitebread
Michael Kendus
GE Advanced Technology Laboratories
Moorestown, NJ

Andrew S. Cromarty
Distributed Systems Technology
Palo Alto, CA and Tully, NY

Abstract

This paper describes a real-time software controller that successfully integrates domain-based and resource-based control reasoning to perform task execution in a dynamically changing environment. The design of the controller is based on the concept of partitioning the process to be controlled into a set of tasks, each of which achieves some process goal. It is assumed that, in general, there are multiple ways (tasks) to achieve a goal. The controller dynamically determines current goals and their current criticality, choosing and scheduling tasks to achieve those goals in the time available. It incorporates rule-based goal reasoning, a TMS-based criticality propagation mechanism, and a real-time scheduler. The controller has been used to build a knowledge-based situation assessment system that formed a major component of a real-time, distributed, cooperative problem solving system built under DARPA contract. It is also being employed in other applications now in progress.

1 Background

The results reported in this paper were derived in the course of developing the Situation Assessment (SA) component of the DARPA Submarine Operational Automation System (SOAS). The SOAS project explored the application of advanced automation techniques such as AI to the support of the commander of an attack submarine. It produced a large-scale distributed software system whose components provided support in such functions as tactical planning and situation assessment.

To control the course-depth-speed profile of the vessel (referred to as "ownship"), manage its staff, and allocate its weapons and other resources, the Commanding Officer (CO) must maintain a timely and accurate understanding of the external situation. SA's responsibility is to construct and maintain, in coordination with a human operator, a *scene assessment* in real time. The scene is generated by processing and interpreting sensor data (primarily sonar data) to create a representation of the submarine's physical environment and the various man-made objects (ships, other submarines, etc., which are known as *contacts*) within range of the sub's sensors.

Knowledge-based processing is used in SA due to the voluminous amount of highly uncertain and incomplete information it must intelligently analyze to achieve its goals. Among the key problems that had to be solved in the design and implementation of SA was the development of an effective method of *domain-based real-time control of problem solving*. This paper describes the SA control architecture which was developed to meet SA's control requirements. By treating problem solving control as a reasoning problem based on both domain knowledge and knowledge of computing resources, the SA control (called the "Metacontroller" succeeded in managing the complex task of situation assessment in real time.

2 Statement of the Problem

The technical challenges to effective problem-solving control for SA stem from four major factors:

- The system must operate in a real-time, dynamic environment.
- The system must select the most useful assessment tasks under current circumstances.
- The system must maintain consistency of its goals and actions (both current and planned) as new data alters perception of the situation.
- The system must support cooperative problem-solving (with its human operator and with other components).

This section explains each of these requirements and their implications for the design of the Metacontroller.

The submarine situation assessment problem is inherently real-time. Ownship must respond to external events such as possible collisions with other vessels. The CO must also consider time-constraints imposed by external circumstances in planning and executing actions which he initiates such as carrying out an attack. Since assessment is a prerequisite to sensible and effective vessel management and target prosecution, the time constraints of the larger command problem devolve on to assessment as well.

Because of the volume of incoming sensor data, the number of assessment tasks which could sensibly be pursued at any time is typically too large to allow execution

*Sponsored by Defense Advanced Research Projects Agency, DARPA ASTO/STP, Submarine Operational Automation System ARPA Order No.6661/50, Issued by DARPA/CMO under Contract MDA972-90-C-0005 of all such tasks. This information overload is one of the reasons for developing an assessment aid for the submarine command staff in the first place. The Metacontroller must therefore guide its own assessment activities on a moment-to-moment timescale based on the expected impact of pursuing one *vs.* another assessment task. For example, an explicit decision must be made concerning whether the next few milliseconds of computing time should be spent on pursuing possible contact correlations or whether a popup contact is a collision threat. Such decisions depend both on time constraints (if there there is a collision threat, is it imminent?) and on domain knowledge (is further analysis of a given contact likely to be of importance to ownship's mission or safety?).

The data which drive situation assessment are often noisy, unreliable, incomplete, unavailable, evolving, or even conflicting, due to sensor limitations, operator limitations, inherent sensor error, physical vessel limitations, and data processing limitations. The Metacontroller must therefore be capable of adapting its evaluation of the priority of assessment tasks as new sensor input is received.

SA is conceived as an operator-controlled component of a distributed problem solving system. It therefore cannot behave autonomously even though it is expected to intelligently control its own functions. Thus, SA control must accept and integrate with its self-derived problem-solving goals both directives from the operator and requests from other intelligent components.

The major challenge in designing the Metacontroller was the *simultaneous* satisfaction of these requirements by the design. Each of these topics has been researched, and treated in isolation. For example, methods for reasoning about task time constraints have been studied, but our application required that such reasoning be integrated with domain-based reasoning (essentially expert reasoning) since the choice of which assessment tasks to pursue depends both on time constraints and on knowledge about which elements of the current situation are operationally most important.

3 Approach

The approach treats control of the situation assessment reasoning process as itself a reasoning process carried on at a metalevel with respect to reasoning done on the domain. The assessment process is conceived as a set of domain-level reasoning tasks (e.g., the task of determining whether a given contact poses a collision threat). The control of that process is conceived of as a reasoning task whose purpose is to decide which domain- level task to perform next. The control reasoning process reasons about goals and tasks. Its function is to determine current system goals, their criticality, and which tasks should be performed given current goals. The reasoning process takes into account the effects of new input data, time constraints, and estimates of the time required for task completion.

We found that this control model could be implemented using conventional AI techniques for the various functions required. The following list indicates the methods used and their purpose:

1. Use domain expert as source of domain-based control knowledge. (much of what the expert said was about control)
2. Divide all functions into tasks supporting goals
3. Use goal-based tasking to perform all system and domain functions
4. Use rule-based reasoning to determine current system goals and tasks
5. Ensure consistency of goals (hence, of the current and planned course of action) through the use of a truth maintenance function.
6. Establish deadlines for all real-time tasks and use these deadlines to schedule the tasks for execution.
7. Trade-off competing real-time tasks according to deadline and precedence.
8. Trade-off competing non-real-time tasks according to criticality.

The remainder of this section presents a description of the system.

3.1 Overview of the Control Architecture

The SA component is composed of the Metacontroller and a set of procedures for performing the domain-level reasoning tasks which create an assessment from sensor input data.

The top-level architecture of the Metacontroller is depicted in Figure 1. The principal elements of this architecture are *Event generation*, the *Metaplanner*, the *Scheduler*, the *Executor*, and the *Truth Maintenance System* with their associated data and knowledge bases and queues.

Event Generation signals the controller, when an event has occurred. An event may occurs when new information is sent to SA via its communication system, or when SA's own domain-reasoning infers a tactically significant datum that could affect SA's control decisions..

The *Metaplanner* uses expert control knowledge to establish and modify goals in response to the events signalled by event generation. It also creates tasks from events. Additionally, the metaplanner is responsible for establishing the class of a task (real- or non-real-time) and initialization of relevant parameters of the task, e.g. its deadline.

The *Scheduler* produces a total ordering of all tasks in accordance with the constraints implied by the class and parameters of the tasks. The scheduler uses a real-time control policy to schedule a real-time task before a non-real- time task, regardless of the relative criticality of those tasks.

The *Executor* executes the task on the top of the schedule. It is also responsible for collecting run-time statistics on each of the task to assist in accurate scheduling of future tasks.

The Metacontroller's *Truth Maintenance System* maintains the consistency of the Metaplanner's goal and task reasoning as new input data is received.

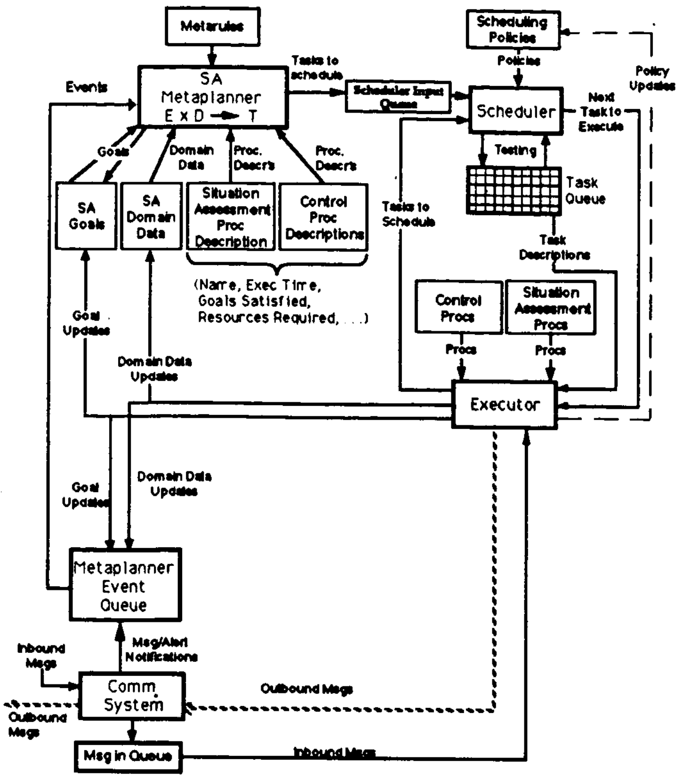

Figure 1: SA Architecture

The tasks whose execution the Metacontroller governs fall into two categories: *control procedure* and *domain procedures*. Domain procedures perform operations fulfilling SA's assessment function. For example, a task that attempts to correlate a newly received sonar tonal with a known contact is classed as a domain procedure. Control procedures perform the operations required to implement the control process itself. For example, a procedure that reads new messages from an input port is a control procedure. Both domain and control procedures are identified for execution by the Metaplanner and scheduled by the Scheduler for execution. Thus, the Metacontroller controls both the execution of SA's problem-solving process and its own behavior as an algorithm.

We now discuss in detail the role of each of the Metacontroller components.

3.2 Distributed Event Generation

Inbound messages to SA are received by the Communications System Interface. The receipt of a message results in the generation of an SA *event*, with the type (class) of event signalled depending on the class of received message.

When the Communications System Interface signals a message receipt event, it also places the body of the received message in the Message InQueue, where it is held until an appropriate control procedure dequeues and processes the message.

Outbound messages are created by individual domain procedures or control procedures and are passed through the Communications System Interface for forwarding to other SOAS components.

3.3 Metaplanner

The Metaplanner can be thought of as a transition function that maps events and goals into tasks (or goals). Periodically, events in the event queue are dequeued *en masse* and processed by the Metaplanner. The Metaplanner employs a dynamic goal base, the events from the event queue, and a set of metalevel planning rules *(metarules)* to select actions (if any) appropriate to each event dequeued from the Metaplanner Event Queue. The Metaplanner produces sets of tasks for execution by the Scheduler, based on the current set of events, goals, and metarules.

Upon each invocation of the Metaplanner, events are read from the Event Queue, goals from the goal base, and metarules relating goals and events to tasks from the Metarule knowledge base; the Metaplanner then forward chains exhaustively on the events to determine the full set of consequent tasks that the events (taken as assertions) "imply."

Metarules take the form

$$\mathcal{E}\mathcal{D} \rightarrow \mathcal{T}$$

where

$\mathcal{E}$ is a set of Events dequeued from the Metaplanner Event Queue

$\mathcal{D}$ is the control Data set, consisting of dynamic, persistent goals and of control *facts*. Control facts are assertions of beliefs currently held about the domain, for example, "Contact S2 could be a submarine".

$\mathcal{T}$ is a set of generated tasks

that is, they map event-goal pairs to a corresponding *task* to execute. The task may be either a base-level (domain level) or metalevel (control level) task, and it is through control level tasks that the goal base is modified and new goals are established. For example, a metarule might be of the form, "If event E_i has occurred and we have goal G_j, then schedule for execution task T_g," where T_g is a task that inserts a new goal g into the goal base.[1]

The Metaplanner forward chains on the events fully until all eligible metarules have fired. Each generated task is assigned a *criticality* derived from the goal that motivated its elaboration. The set of tasks to execute resulting from a single such Metaplanner invocation are queued and held in the Scheduler Input Queue; when the closure of the event set has been computed for this goal and rule set, the Metaplanner flushes the Event Queue, queues its set of derived tasks in the Scheduler Input Queue, and expires.

Although events do not live beyond one Metaplanner invocation, metarules and goals are maintained across Meta-

[1]Note that by requiring goal knowledge base manipulations to occur singly within the context of a task execution rather than at Metaplanning time, we effectively eliminate the order-dependent processing ambiguity that could arise in Metaplanner operations if it were possible to modify the goal set while it was in active use by the Metaplanner.

planner invocations. The metarule and goal knowledge bases may be modified from time to time by control procedures as described above. Most such instances of control procedure execution as well as most executions of domain procedures occur because the Metaplanner identifies the procedures as tasks to be scheduled and executed as already described. However, as Figure 1 indicates, tasks may also be submitted for scheduling directly from the Executor. This occurs when an executing task spawns a child task. Task spawning was implemented as an efficient way to handle such features as the establishment of periodic tasks and partitioning tasks into subtasks. When a task spawns a child task, the parent may assign any goal to the child. The assigned goal is added to the goal base if necessary. In practise, most children are assigned the goal of the parent. Note that task creation of goals and tasks does not violate the conceptual model already described. The same effect could have been achieved by instead implementing tasks which signal an appropriate event instead of directly spawning a new task. The event could then be processed with metarules added to the knowledge base for the purpose. The decision to use direct spawning of tasks simply allowed more efficient implementation.

Metaplanning, essentially the process of deciding what to plan, is an approximate description of the activities engaged in by this, the largest part of the Metacontroller. Whereas the SA Scheduler actually determines which computing activities SA will engage in during the next processing epoch, the SA Metaplanner determines what the Scheduler will have available as its scheduling alternatives. Viewed alternatively, the Metaplanner engages in *explicit planning at the metalevel*, that is, it employs a knowledge-based planning technique to determine what sets of tasks SA should execute as a computational system.

3.4 Scheduler

The Scheduler is a true real-time task scheduler. Each task is a descriptor pairing a procedure identifier with invocation specific parameters. Associated with each procedure in the Procedure Description knowledge bases consulted by the Metaplanner are data characterizing the procedure's anticipated execution time; the Scheduler uses these execution time predictions together with the Metaplanner-specified criticality value assigned to each task to build an execution schedule for all pending tasks, including both the set of tasks dequeued from the Scheduler Input Queue at Scheduler invocation time and the set of previously queued but unexecuted pending tasks already waiting in the Scheduler's Task Queue.

The Scheduling policy is as follows. Each task has a *criticality*, nominally inherited from the parent goal that led to the task's elaboration by the Metaplanner.[2] Most tasks also have a real-time deadline; these are *real-time (RT) tasks*, *vs.* the *non-real-time (non-RT) tasks*, which lack execution deadlines. (It is required that an estimate of execution time be available for tasks having execution deadlines.)

Real-time tasks generally are scheduled to execute before non-real-time tasks, to ensure that real-time execution criteria are satisfied.[3] The RT tasks are scheduled for execution (i.e. mapped onto an execution timeline) based on their required completion time and anticipated execution time. Where two tasks qualify for the same timeslot, criticality is used as a tie-breaker. Once such a schedule has been constructed for RT tasks, non-RT tasks are scheduled; where possible, they are fitted into the interstices between already-scheduled RT tasks, and if no such space exists in the schedule, they are appended to the end of the schedule in order of decreasing criticality.

At the completion of a Scheduler invocation, the Task Queue is a total ordering of tasks ordered by scheduled start time. The first task in the Task Queue is the *next task to execute*.

3.5 Executor

The Executor dequeues the *next task to execute* and assigns the processor to it. Execution is non-preemptive, i.e. the task executes to completion once initiated. From SA's perspective, all task executions in the current prototype are atomic in the sense that they always run to completion and cannot be interrupted by (for example) intervening message deliveries or other internal or external events. The question of whether to implement task interruption was considered early in the process of designing the Metacontroller. Analysis of both options showed that task interruption would vastly increase the complexity of the goal and task reasoning process. We therefore chose to define atomic tasks.

3.6 Data and Knowledge Bases

In addition to the major components of the architecture as described above, SA also contains several data and knowledge bases. They include:

- The *Metarules knowledge base*, from which the Metaplanner obtains its rules for planning task executions based on the current contact (as evidenced by triggered events) and the current goal set.

[2] In some cases, as described above, a task will be presented to the Scheduler for scheduling and subsequent execution by another task. The creating task is permitted to specify any of several criticalities for the created task: that of the creating task, that of some goal to which the created task is intended to bear a causal relationship, or the same criticality as some other identified task in which the new class is intended to have equivalence-class membership for criticality purposes. Under most circumstances, however, we would expect that tasks will be elaborated by the metaplanner and will inherit their criticality from the parent goal that they are being executed to satisfy.

[3] This follows from the principal that SA is a real-time problem-solving system. If satisfaction of execution deadlines were subordinated to some other metric, e.g. task "priority," then SA would be a conventional non-real-time problem-solving system. In a real-time system, the deadlines are incontrovertibly more important than other performance metrics.

- The *SA Goals knowledge base*, containing the assessment goals that will employed by the metalevel to perform metaplanning.
- The *SA Domain Database*, comprising the collection of facts, assertions, and hypotheses about the undersea world that are used to produce assessments. This includes all knowledge concerning the current set of hypothetical contacts in the external world.
- The *Situation Assessment Procedure Description database*, containing a description of each computational procedure that SA's Executor can execute to manipulate domain-level data (i.e. data about vessels, contacts, acoustics, etc.).
- The *Control Procedure Description database*, wherein reside corresponding descriptions of each metalevel computational procedure (those that manipulate input messages, task queues, etc.).
- The *Situation Assessment Procedures database*, containing the bodies of executable domain-level procedures, such as procedures for performing correlation of contacts.
- The *Control Procedures database*, containing the actual bodies of executable metalevel procedures.

3.7 Truth Maintenance Substrate

SA uses a truth maintenance system (TMS) to ensure consistency between goals. The SA TMS is an extension of the JTMS (Justification-based TMS) [6; 2] approach, in which justifications are represented as Horne clauses. The JTMS has been extended to establish and propagate goal criticalities. This capability is essential for reasoning about the importance of goals, and in turn, of tasks.

The TMS represents two types of "assertions": facts and goals. Facts are statements of belief that ultimately justify one or more goals, goals are a state SA is trying to achieve or a question to answer. Similarly justifications are of two types: fact and goal (meaning that the TMS justifies the in-ness of a fact or goal).

A fact justification conjoins a set of facts and goals (i.e. assertions) and represents boolean support for the consequent fact when the conjunction is true. A goal justification conjoins a set of facts and goals to represent boolean support but also represents the importance (criticality) that support wants to lend to the consequent goal. Each goal justification has an associated criticality.

The criticality of a goal is assigned in one of two ways. If the goal is an assumed goal (i.e., a premise) a criticality is assigned at the time of the assumption. Other goals are derived goals, and obtain their criticality from the criticality of justifications. The criticality of the supporting justification is used as the criticality of the goal. When a goal has more than one justification, the one with the maximum criticality is used to assign the criticality.

4 Test Results

TheMetacontroller was tested as part of the SOAS prototype. Tests were performed in a distributed computing environment consisting of networked workstations. Each major component of SOAS, e.g., the SA component, resided on a single workstation and communicated with other components on other workstations through a communication substrate based on ISIS. An additional workstation on the network supported a simulator which exercised the SOAS prototype by providing simulations of: sensor data, data on ownship (e.g., ownship course and speed), and environmental data such as sound-velocity profiles. This distributed system was tested against several contact-behavior scenarios by personnel with both computer science and Navy operational experience.

Through the use of the Metacontroller, SA was able to meet about 80% of its real-time deadlines in the midst of executing optional tasks opportunistically. We expect that the 80% figure could be improved upon through improved use of the existing metacontroller, for example, by replacing unbounded-time algorithms with any-time versions.

SA uses about 30 metarules and typically maintains goal sets containing between 20 and 100 goals. The metacontroller runs efficiently. Approximately 5% of the total processing time typical SA scenarios is spent in metacontroller processing. The metacontroller is implemented in Common Lisp; we plan to port the metacontroller to the C language.

5 Conclusions

The development of the Metacontroller demonstrates two points:

- A feasible method for integrating domain and resource-based reasoning about control of a knowledge-based procedure.
- The practicality of creating solutions to difficult AI system design problems by coupling several well-known AI techniques.

An obvious problem in building real-time AI systems is the fact that control of the system typically depends on two largely unrelated factors:

- Time deadlines and related computing resource constraints.
- The control decisions inherent in the problem-solving domain.

Each of these factors has been the subject of investigation, individually. The first factor has been studied in some depth [5]. Concepts such as any-time algorithms and the use of interrupts in AI tasks have been investigated [3; 7]. The second factor has led to well-known concepts such as blackboard systems and goal-based reasoning. Although the general problem of integrating these two factors has been discussed (e.g., [4]), there appear to be few published sources describing detailed mechanisms for such combined reasoning. Typically, much of the expertise in a given problem domain has to do with reasoning about what to do next.

This inference process cannot be discarded when real-time constraints are imposed. Instead, it must be integrated with the complimentary process of reasoning about deadlines. The Metacontroller design provides a mechanism to do this integration.

The second point demonstrated, that several AI techniques can be usefully coupled, is important in advancing the success of AI as an engineering discipline. The Metacontroller combines:

- Rule-based reasoning to form goals,
- Goal-directed processing,
- TMS-based criticality propagation,
- Real-time scheduler.

The noncontrol elements of SA integrate additional techniques as well.

While each of these techniques is based on well-known concepts, its not typical to find them coupled in a single integrated function. More frequently, we see systems built using one or two techniques (e.g., a diagnostic system built around a TMS [1]). Such homogeneous designs are appropriate for basic research in AI techniques and for engineering applications where they happen to work. However, our experience has led us to believe that the engineering potential of the existing body of AI research cannot be properly exploited unless multiple techniques are integrated. In the case of the Metacontroller, we showed that integration of multiple techniques produced very promising results on a problem which appeared beyond the means of any single techniques with which the authors are familiar.

References

[1] de Kleer, Johan, and Williams, Brian, Diagnosing multiple faults, *Artificial Intelligence*, 32:97–130, 1987.

[2] Doyle, Jon, A Truth Maintenance System, *Artificial Intelligence*, 12:231–272, 1979.

[3] Erman, Lee D. ,Ed., *Intelligent Real-Time Problem Solving: Workshop Report*, Santa Cruz, CA, November 8 and 9, 1989.

[4] Hayes-Roth, B., Washington, R., Hewett, R., Hewett, M., Seiver, A, Intelligent Real-Time Monitoring and Control, Technical Report No. KSL 89-05, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

[5] Laffey, Thomas J., Cox, Preston A., Schmidt, James L., Kao, Simon N., Read, Jackson Y., Real-Time Knowledge-Based Systems, *AI Magazine*, Spring, 1988, Vol. 9, No. 1, pp. 27-45.

[6] McAllester, David, A Three-Valued Truth Maintenance System S.B. Thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1978.

[7] Sharma, D.D., and Narayan, Srini, An Architecture for Intelligent Task Interruption, *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems*, Detroit, MI, August 20, 1989.