2003

Knowledge-Based Task Structure Planning for an Information Gathering Agent

John Phelps

KNOWLEDGE-BASED TASK STRUCTURE PLANNING FOR AN INFORMATION GATHERING AGENT

By

John Phelps

B.S. University of Maine, 1999

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

August, 2003

Advisory Committee:

Thomas Wagner, Assistant Professor of Computer Science, Advisor

Paul Bauschatz, Associate Professor of English

James Fastook, Professor of Computer Science

Roy Turner, Associate Professor of Computer Science

KNOWLEDGE-BASED TASK STRUCTURE PLANNING FOR AN INFORMATION GATHERING AGENT

By John Phelps

Thesis Advisor: Dr. Thomas Wagner

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
August, 2003

An effective solution to model and apply planning domain knowledge for deliberation and action in probabilistic, agent-oriented control is presented. Specifically, the addition of a task structure planning component and supporting components to an agent-oriented architecture and agent implementation is described. For agent control in risky or uncertain environments, an approach and method of goal reduction to task plan sets and schedules of action is presented. Additionally, some issues related to component-wise, situation-dependent control of a task planning agent that schedules its tasks separately from planning them are motivated and discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This thesis describes the design and development of an Internet information gathering agent that assists its users with online trip planning. The agent, called Tripbot, uses and extends a generic agent control infrastructure that has been used on a number of other problems in other domains to control how it gathers and fuses information on the Internet.

Tripbot accepts high level descriptions of trip requirements from its users through a web page. It then processes the user's requirements to create a problem description that it can use to plan and schedule appropriate Internet data gathering and information fusion actions to fulfill the user's trip requirements. Ultimately, Tripbot returns a list of candidate itineraries that fit the user's trip requirements based on the information that it has gathered on the Internet.

This thesis describes the Tripbot information gathering agent architecture and discusses the components that instantiate the architecture in a system. The primary focus of the thesis is task structure planning for information gathering and its interactions with agent-based scheduling and execution monitoring.

Tripbot is a time-constrained and goal-directed agent. This means that the agent was developed to generate plans and schedules of action from goals within a specified time bound. The time-bounded aspect of the agent architecture combined with its use of a generic task structure planning component make the agent an instantiation of an agent architecture that we call the Flexible Soft Real-Time Agent (F-SRTA) architecture.

The task structure planning component that is added to a partial instantiation of the SRTA architecture is called the Design-To-Schedule (DTS) planner since it generates task structures with an objective function that focuses on the expected utility of schedules generated.

1.1 Agent-Orientation

Agent-oriented development represents a software system development approach that stresses the use of domain-independent control components, ontologically grounded domain representation for problem expression, continuous accountability to environmental context that stresses flexibility and adaptability, especially with respect to the context of time. The term "agent" has many definitions because it has often been thought of as an extension of an object, rather than as a model or method of computer system design. The following notions of agents have been offered:

Russell and Norvig[RN91] see agents as computer systems that sense, deliberate, and act, where "deliberation" can be the simplest computational procedure could; i.e., a thermostat could be thought of as an agent.

Rodney Brooks[Bro91] stresses behaviors, situatedness, and embodiment leading to emergent intelligence, with an emphasis on role of the environment in the display of intelligence.

Jennings, Sycara, and Wooldridge [JSW98] stress situatedness, autonomy, and, especially, flexibility — which indicates an agent's ability to adapt to changing environmental conditions.

Wagner and Lesser [Wag00] focus on expressive probabilistic representation and robust combinations of reactive and deliberative planning and scheduling, diagnosis, and learning at the single agent and multiagent levels.

The approach taken to solve the Tripbot problem was influenced by the other work cited, but its lineage is most closely tied to the efforts connected with the Bounded Information Gatherer (BIG) [LHK+00], which we identify as the Task Analysis, Environmental Modelling, and Simulation

(TÆMS) Agent approach. Using the TÆMS agent approach means that the TÆMS language is used to represent the control aspects of the problem internally. It also implies that there exist within the agent system TÆMS parsing and analysis components.

Given that Tripbot is solving a time-bounded problem, there is immediately the question of how it should allocate computational resources to achieve a set of goals. Independent of the way in which a system solves a problem is a problem of which and how much of which computational resources (time and space) should be used to solve it. A decision must be made based on the expected time and space complexity requirements of components required to solve a domain problem about the best allocation of computing time and space.

1.2 Outline of Remaining Chapters

In the remaining sections, we first discuss TÆMS work related to the development of the TÆMS-based F-SRTA architecture, and the Tripbot instantiation of the F-SRTA architecture. We then discuss the DTS planning system in some detail. A short discussion of criteria for a leveled approach to agent schedule failure analysis is then given. Finally, we give and evaluate some preliminary experimental results and conclusions about local task structure heuristic optimization effects on task structure schedule generation and agent execution.

Briefly, the primary result of this research is that the application of heuristic, criteria directed goal reduction can yield significant time speedups versus a complete approach to goal reduction in TÆMS task structure planning. This improvement comes with no statistically significant schedule quality loss with a quality-optimizing heuristic, and no schedule quality density loss with a time-optimizing heuristic.

There is schedule quality gain in complex circumstances associated with the quality-optimizing heuristic and quality-density gain with the time-optimizing heuristic. The favorable results for the incomplete heuristic versus a complete baseline is due to the computational infeasibility of complete scheduling analysis for problems of moderate size, and the heuristic approach of the DTC scheduler which is used to solve the task structures generated by the DTS planner.

1.3 Summary of Contributions

Tripbot/F-SRTA contains many independently developed subsystems that work together, some of which have been developed by other people. This list briefly highlights some of my contributions to the development of Tripbot/F-SRTA:

DTS TÆMS planning language and parser — a language describing how goals are solved through candidate TÆMS task structures; i.e., a domain theory language and a problem definition language, and a parser to translate the information into a form useable by the DTS planner.

DTS TÆMS task structure planner — Hierarchical Task Network planner that includes:

- support for Java reflective computation in any preconditions and axioms;
- support for method performance attributes generally, and TÆMS cost, quality, and duration attributes, specifically, as discrete probability distributions;
- support for encoding plans in TÆMS task structures;
- support for tracking expected performance attributes of each plan encoded in a TÆMS task structure to a current plan performance "horizon";
- extensible, n-attribute tradeoff mechanism, currently supporting TÆMS criteria definition for rating candidate plans and subplans;
- extensible, n-heuristic selection mechanism, currently supporting six selection heuristics based on order of alternatives provided by TÆMS criteria rating: low, high, median, random, fast, extremes, all.

DTC scheduler driver and parser — driver that runs the DTC scheduler and parses the schedules that it produces.

Tripbot/F-SRTA Executor — system that allocates compute resources and that creates, runs, and monitors instantiated schedules, using the following subcomponents:

• MTIM — the Method Type Instance Map is a mapping from method types used by the DTS planner and DTC scheduler to actual methods that can be run by the executor;

• IRT — the Information Resource Table is a table containing new state information generated by actions of the agent in the world, but that does not necessarily imply a world-state model change for the agent, e.g., information about flights produced by an information gathering action.

• MSM — the Method State Map is an experimental component used to identify the world state at a given point in a running schedule to be used for "WHAT-IF" continuous planning in parallel to schedule execution.

Experimental harness — includes a planning problem generator for the Tripbot and Probabilistic Blocks World domains.

Computational complexity versus utility results — results about where planning and scheduling runtimes intersect over problem complexity are given as well as results about the effect of heuristics with different complexity on schedule quality density, assuming a uniform distribution of method attributes and deadlines.

Rescheduling criteria — some rescheduling criteria based on simple schedule statistics are explained; the criteria are being explored in ongoing research to determine the best criteria in a given task situation with regard to recovery compute time reduction.

Complexity analysis of task structure planning and scheduling — results concerning the computational complexity-dominating planning and scheduling operations in Tripbot/F-SRTA, including some results motivating the inclusion of problem characterization for problem solving method selection in agent systems and a TÆMS-based proof that TÆMS task structure scheduling (TSS) is $\in$NP-hard.

Other supporting contributions to F-SRTA/Tripbot — a simple keyword query expansion Wordnet bridge, Tripbot travel domain ontology elements, and information source wrappers supporting trip data gathering.

Chapter 2

TÆMS

TÆMS stands for Task Analysis, Environmental Modeling, and Simulation. TÆMS is a language that models an agent task environment, specifying what action alternatives are available to an agent, how candidate actions relate to one another, and how they may contribute to a measure of single and multiagent utility[Dec95]. TÆMS represents an attempt to achieve maximum tractable generality in a computable language to address the problem of optimal single agent task selection and multiagent task coordination in uncertain environments with special focus on task interrelationships.

The origin of TÆMS is in the quest for general theories in AI [Dec95]. TÆMS exists to spur the development of general hypotheses about single and multiagent behavior as well as strong experimental studies in the domains of single and multiagent action, including studies of ablation and parameter optimization.

TÆMS has proven useful for solving problems in many different domains including:

- Dynamic readiness and repair for airplanes [WGP03a],
- Complex, multipurpose, uncertainty minimizing information gathering [LHK+00],
- An intelligent home prototype [LAH+03],
- Real-time agent sensor network for vehicle tracking [HVM+01],

- Distributed hospital patient scheduling [DL00],
- Supply chain control [WGP03b], and
- Travel planning, described in [WPQ+03] and the present document.

2.1 TÆMS Elements

A TÆMS task structure represents an agent's task and resource environment in a directed graph representation, consisting of TÆMS nodes and TÆMS arcs. Both TÆMS nodes and TÆMS arcs have associated attributes. Additionally, there are standard attributes and extended attributes for method and task nodes, providing a rich language for modeling and simulating complex stochastic processes and options for their control. There are four basic types of TÆMS nodes:

task groups – the roots of task structures,

tasks – the interior nodes of a task structure,

methods – the terminal, actionable nodes of task structure, and

resources – nodes connected to methods that represent a thing that can be used by a limited set of agents at a given time or that has a renewable or nonrenewable capacity.

The attributes that these node types possess vary, but all must contain the following attribute to be referenced between components:

label – a unique name.

Task groups, tasks, and methods have an *agent* attribute, which associates a task with an agent and which can specify a *local* or *nonlocal* agent. If the agent is nonlocal, then the task is also described as a *nonlocal task*, and a coordination episode may be invoked to secure the performance of the task from its associated agent if necessary. Resources do not have an agent attribute since they may often be shared between agents. Resources may be shielded from or exposed to agents within a multiagent system through the use of exporting or importing views of partial or conditioned TÆMS

task structures. Conditioned task structures are task structures that have some modification from the task structure currently bound to an executing agent schedule to explore possible "what-if" schedules.

Task groups, tasks, and methods contain attributes pertaining to their coordination context and the way in which they may be locally scheduled, which include:

arrival time – the time at which a task arrives within the TÆMS task structure which is important for some coordination protocols, e.g., first come first served,

earliest start time – the earliest time that a task may begin in a schedule of tasks, and

deadline – the latest time that a task may complete in a schedule.

Task groups and tasks also contain attributes pertaining to the task decomposition structure, including:

subtasks – a list of the labels of the node's subtasks, and

qaf which describes the type of quality accumulation function that governs the subtask to supertask performance relations on the quality characteristic function. A quality accumulation function defines how the expected utility of a task's subtasks affect its expected utility.

Methods also contain the following attribute:

outcomes – an attribute that describes the expected stateless effects resulting from the execution of a method.

The expected effects from the execution of a method in a particular environmental context are expressed in the method's outcomes through characteristic functions. Presently, these functions return distributions for the method's *cost*, *quality*, and *duration* in one or more possible outcomes. There is an additional outcome function which produces an expected outcome according to a separate, independent outcome distribution. Presently, most of the TÆMS generation and analysis

tools support outcomes and characteristic functions with at least discrete probability distributions, although support for more expressive distributions has been explored in [WL03].

2.2 TÆMS Task Interrelationships

TÆMS nodes are connected by two types of arcs: decomposition arcs and interrelationship arcs. The decomposition arcs indicate both a process and a functional relationship between a supertask and its subtask. The functional relationships between supertasks and subtasks are called characteristic accumulation functions (CAFs). There are presently eleven CAFs defined for the quality attribute. Since most TÆMS components only support analysis of the quality attribute, they are usually referred to as Quality Accumulation Functions (QAFs). The eleven QAFs defined and supported by TÆMS are as follows:

min – specifies that the quality at the supertask is equal to the minimum quality produced at one of its subtasks, i.e., $Q(task_i) = \min(subtask(task_i, j))$ but that in practice this is treated equivalent to a logical *and* over the subtasks;

max – specifies that the quality at the supertask is equal to the maximum quality produced at one of its subtasks, i.e., $Q(task_i) = \max(subtask(task_i, j))$, but that in practice this is treated equivalent to a logical *or* over the subtasks;

sum – specifies that the quality at the supertask is equal to the sum of the quality at the subtasks, i.e., $Q(task_i) = \sum(subtask(task_i, j))$;

all – specifies that the quality at the supertask is equal to the sum of the quality at the subtasks, i.e., $Q(task_i) = \sum(subtask(task_i, j))$ iff all of the subtasks complete successfully;

seq_min – is defined the same as the the min QAF, except that the subtasks must be performed in a defined sequence;

seq_max – is defined the same as the max QAF above, except that the subtasks must be performed in a defined sequence;

seq_sum – is defined the same as the *sum* QAF above, except that the subtasks must be performed in a defined sequence;

seq_last – specifies that the quality at the supertask is equal to the quality at the last performed subtask and that the subtasks must be performed in a defined sequence;

exactly_one – specifies that the quality at the supertask is equal to the sum of one and only one of the subtasks selected;

last – is defined the same as *seq_last* above, except that no order on the performance of the subtasks is given; and

sigmoid – specifies that the quality at the supertasks increases according to a zero-shifted and appropriately and skewed sigmoid function, i.e., $y = \frac{1}{1+e^x}$.

2.3 TÆMS Interrelationships

TÆMS interrelationships are ways of describing interactions between tasks that fall outside of supertask-subtask relations. There are interrelationships between tasks and resources that describe how tasks consume, produce, or limit resources. More detail on those relations can be found in [Dec95, Wag00]. The DTS planner handles task-to-task interrelationships through annotations in task decompositions and through ordering in plan steps. The interrelationships supported by the DTS planner are:

enables — specifies that the run of one task enables the run of another task;

facilitates — specifies that the run of one task can influence positively (in TÆMS criteria-directed utility terms) the expected value of characteristics at another task; and

hinders — specifies that the run of one task can influence negatively (in TÆMS criteria-directed utility terms) the expected value of characteristics at another task.

2.4 TÆMS Criteria

The objective function used to rate solution alternatives for domain independent TÆMS analysis components is specified through TÆMS criteria. The criteria were first developed and applied in the DTC-scheduler [WL01]. The criteria mechanism is flexible enough in principle to include an arbitrary number and organization of dimensions. However, presently it consists of four primary dimensions and one meta dimension that relates the four primary dimensions. The four primary dimensions are:

goodness — the utility of the combination of expected values for the TÆMS characteristics of quality, cost, and duration, where higher quality has higher utility, and where lower cost and duration have higher utility,

certainty — the certainty of the overall schedule,

threshold — the thresholds for expected values of the schedule TÆMS characteristics of quality, cost, and duration, and

threshold certainty — the certainty of obtaining each TÆMS characteristic threshold.

2.5 A TÆMS Example

Figure 2.1 shows portions of TÆMS task structure for Tripbot's itinerary generation. The `Generate Itineraries` task structure is a hierarchical decomposition of a top level goal. The top level goal, or task, has two subtasks which are to `Gather Information` and `Display Itineraries`. Each of these tasks is decomposed into subtasks, e.g., `Trip Query`, and finally into terminal, actionable methods, e.g., `Query Yahoo Weather`. Non-primitive tasks are represented with rounded rectangles, while primitive actions are represented with rectangles in most TÆMS figures.

Notice that the task of displaying trip itineraries, `Display Itineraries`, is facilitated by the actions associated with the `Additional Trip Query` task. The edge that denotes

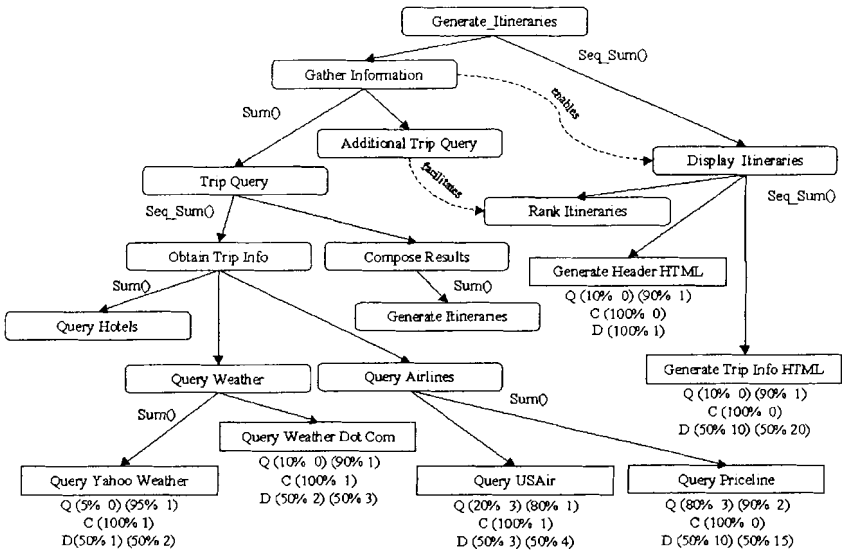

Figure 2.1: An example of TÆMS for the trip planning domain.

the facilitation is denoted by a dashed or dotted arch, annotated with the word *facilitates*. In this model, if the `Additional Trip Query` task is successfully performed before the `Rank Itineraries` task, it will positively augment the quality of the `Rank Itineraries` task, and hence, the `Display Itineraries` task. The *facilitates* is one of the eleven *interrelationships* enumerated above. Interrelationships held between task structures located in different agents provide for *non local effect* (NLE) relationships between the jointly held tasks. NLEs effectively identify points over which agents sharing tasks may coordinate.

All of the tasks represented in the figure have terminal methods, but for sake of simplicity, they are not all represented. For each of the methods that is visible, it should be noted that each method has a quality (Q), cost (C), and duration (D) discrete probability distributions associated with it. This is a simplification from the outcomes and characteristics scheme previously described. It is assumed for the purposes of this example that the expected outcome shown is the only one possible.

One sort of tradeoff generated and analyzed for the kinds of unstructured query optimization that Tripbot provides is illustrated by the `Query Airlines` task in Figure 2.1. For the `Query Airlines` subtask, the relatively lower expected quality `Query USAir` would require 3 to 4 time units (minutes in this example) than the higher expected quality `Query Priceline`, which

is expected to require 10 to 15 minutes. However, since the accumulation of supertask quality in this case is governed by the *sum* QAF, and there are no ordering *interrelationships*, these methods may be run in parallel. In reality, most actions have logical consequences; e.g., data gathering actions of one type enable information generation actions of another. Consequences of actions have implications for both the DTS planner and the DTC scheduler. For the planner, it means exploring a separate subplan branch. For the scheduler, it means exploring separate subschedules. More specifically, if there is a deadline on the `Generate Itineraries` task structure, then the scheduler will need to perform feasibility analysis for each selected subschedule to see whether a complete schedule containing it will meet the deadline.

Criteria definition can be used to modulate the relative importance of TÆMS attributes in candidate plans and schedules, so the importance of action duration is, in this analysis framework, no more important than its cost or quality. It is typical in practice, however, to look at duration slightly differently than the quality and cost attributes, since duration is usually viewed as a hard, physical limit, whereas quality and cost are typically viewed as more soft and intangible limits.

Quality was designed from the beginning to be a "vague" [Dec95] and unitless value into which other domain-specific attribute values may be aggregated. A domain problem solver abstracts domain problems into the language of TÆMS quality, cost, and duration. The problems can then be solved by TÆMS components. It is usually best to think of the TÆMS planning, scheduling, and coordination components generally as quality optimizing components, but, as alluded to above, TÆMS criteria definition can direct optimization in any of the defined dimensions in very many ways.

The ability to model action alternatives for an agent within an individual or social control context is an important attribute of a language and system for any changing and uncertain agent task environment, especially one where information about the world, compute space, or compute time may be constrained. Thus, central goals of research into F-SRTA are to explicitly and generically account for these control and metacontrol attributes and constraints, and to produce mechanisms that are capable of optimizing simultaneously over control and metacontrol problems to provide

the best results for situated agents.

Because the general question is very complex, we narrow our focus to a few aspects of the metacontrol problem for DTS-planning: The question of which task alternatives should be included in task structure planning to optimize options for the DTC scheduler, according to a given task environment. A connected question is for how long each alternative should be explored.

Alternatives are present at several levels of the `Generate Itineraries` task group. The `Obtain Trip Info` task, for example, has three subtasks joined under the *sum* QAF. The generation of such a structure would depend on the the explicit representation of choice in the generating language domain theory as well as appropriate bindings in the problem instance. The DTC scheduler then has the opportunity to include any number of the alternatives under the `Obtain Trip Info` task, as provided for by a computational and task environment context.

2.6 TÆMS Agents

TÆMS agents leverage the expressive, computable language of TÆMS combined with its single and multiple agent task plan and schedule optimization components to solve complex agent system control problems.

A version of the TÆMS agent architecture is given in Figure 2.2. The present research into F-SRTA attempts to generalize the TÆMS language and tools further, pushing the control problem to one of posting goals and defining plan options for their decomposition, rather than complete task structures. This addition of the DTS planner is part of the domain problem solver component depicted in the center of the figure. The planner does not obviate the need for other orthogonal, encapsulated domain representation and problem solving code, but rather adds a tool that can aid in the formulation and solution of particular kinds of domain problems. Specifically those problems amenable to expression in a hierarchical task network.

The impetus for the Design-To-Schedule planner was twofold. The first reason for its creation was to facilitate a higher level of autonomy for TÆMS agents, meaning that a range of possible rational behaviors for the agent can be derived from goals instead of hard-coded task structures.

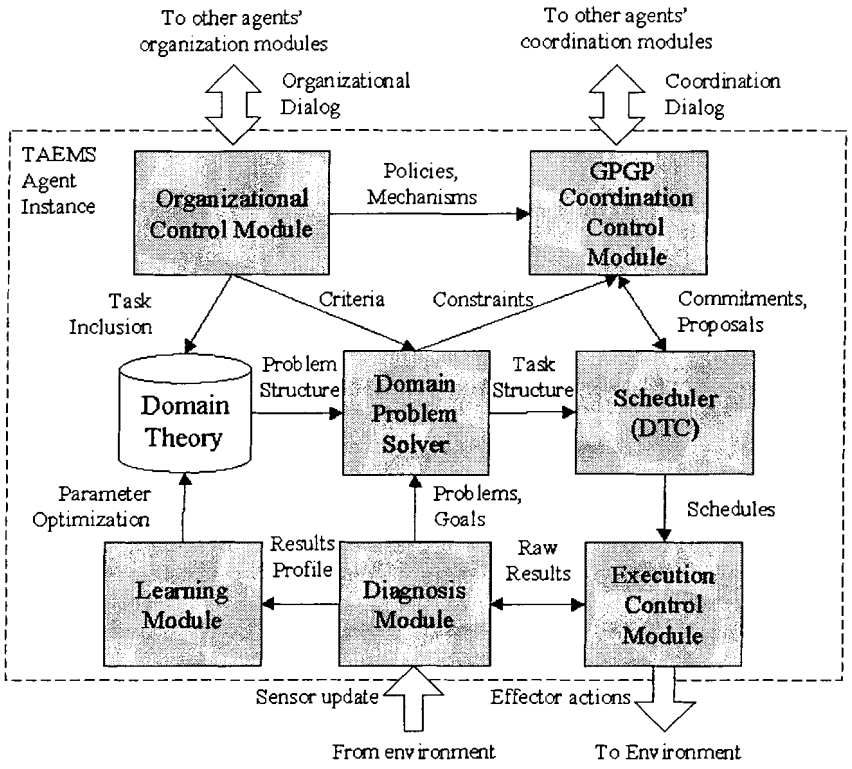

Figure 2.2: The TÆMS agent architecture.

The second reason is to attempt to embed common domain problem solution characteristics into a reusable component. Previous domain problem solvers were nearly invariably not generally reusable components, while the scheduler, the execution control module, diagnosis module, coordination module, learning module, and organizational control module were.

There were two key obstacles to creating a general purpose task structure generator:

- language – what should the TÆMS task structure generation language encode?
- computation – what sorts of computation should the task structure generator be able to do given a problem represented in the desired language?

These questions and others are addressed later in a section just about the DTS task structure generator.

Chapter 3

RELATED WORK

The architecture of Tripbot is loosely based on the Java Agent Framework (JAF)-based Soft Real-Time Agent architecture, which we initially attempted to use as a basis for Tripbot without much success [Hor03, HLVW03].[1] The lexical-semantic processor to expand query terms was conceived of independently, but there is a sizable body of related work, including that of [BS03]. We made use of the WordNet lexical-semantic dictionary to lookup words related to query keywords [MBF+98]. The Design-To-Criteria TÆMS scheduler [WL01] was used without modification. The Design-To-Schedule planner was based on the Simple Hierarchical Ordered Planner (SHOP) [NCLMA99]. The Executor was also loosely based on JAF's Executor[Hor03], making use of Java reflection to instantiate the methods planned by the DTS planner and scheduled by DTC scheduler. To summarize, some of the areas of computer and information science that can be profitably applied to this problem are: agent-oriented software architecture, semantic lexical analysis, unstructured information extraction, knowledge engineering, planning, scheduling, execution control, diagnosis, and learning. The list is to meant to suggest that intelligent autonomy for information systems requires the integration of many disparate problems and solution methods. We now present a summary of the related research, linking their contribution to the F-SRTA architecture and the Tripbot implementation.

[1] The quality of the current JAF distribution has improved significantly since then.

3.1 Agent-Oriented Software Architectures

Agent-oriented software architectures represent a class or style of reusable system designs in the sense described in [GS95, GAO95]. That is, they specify a kind of gross organization of software components and approach to the control of the components. The agent-oriented software architectural approach taken by TÆMS and TÆMS agents attempts to circumvent, to an extent, architectural mismatch [GAO95] by relegating many aspects of system control to a reusable set of components and a control specification language. It thus merges a potentially dizzying variety of abstract interfaces into one that is flexible and expressive: the TÆMS language and evaluation criteria.

System designers using TÆMS and TÆMS agents make assumptions about the control of their system explicit in their TÆMS problem encodings as well as the TÆMS evaluation criteria used to produce control solutions. So in the TÆMS agent-oriented software architecture, the control interface is a computable language capable of expressing the control problem(s). Once posed in such a language, a set of reusable TÆMS components can the solve the problems as stated in the language. The set of TÆMS components is considered a "domain-specific" software architecture as described in [WS03], where the domain of expertise is the class of problems that can be described in the TÆMS language.

Since designing an agent system that would support planning, along with scheduling was a requirement of Tripbot, the Multiagent Planning Architecture (MPA) [WM98] was of interest because of its focus on integration of different control solvers. The MPA focuses on integrating divergent planning approaches into a system capable of solving complex planning problems and is thus a particularly interesting architectural concept, focusing on the multiagent system as a basis for architecture, rather than agent itself.

MPA's goals are similar to Tripbot's since both deal with the integration of disparate approaches to system control. The difference is that MPA seeks component integration at the agent level whereas Tripbot seeks it at the agent component level. In the MPA, agents communicate peer to peer in planning cells, each of which has a cell manager that is responsible for decomposing

a given subset of a planning task, discovering agents capable of handling each decomposition, and distributing the plan subproblems to capable and available agents. In Tripbot, the high level problems are cast as planning goals, and the DTS planner then decomposes the goals (through a hierarchical task network) into alternative solutions encoded within a TÆMS task structure which can then be scheduled by the DTC scheduler. This process could, in principle, be done in parallel for mutually exclusive goals and subgoals.

Although similar in its view of an individual TÆMS agent as a microagent system, JAF takes a very loosely coupled approach to agent component integration and control, providing an agent state holding variables accessable by all other agent components and various possible views of the agent's TÆMS task structure, but not specifying how each component should behave, including how it should interact with other components. JAF uses a round-robin polling loop that provides a rudimentary cooperative time-sharing mechanism for its agent components.

One of the more intriguing aspects of the MPA is a focus present in the system on generating competing alternative plans within planning cells. This idea has direct bearing on the development on the DTS task structure generator. In MPA, as is the case in Tripbot, flexible control policies were available for plan generation and planning and scheduling occurred separately.

However, in MPA, competitive plan generation was facilitated by the existence of two planning level cells, base and meta. The base level cells provide sequential solution generation and the meta level cells employ base level cells to support parallel generation of qualitatively different solutions. A deficiency in MPA is that there is no support for coherently composing subplans generated by the distributed planning agents, each competing plan must be functionally complete to be compared against other generated plans.

The Soft Real-Time Agent (SRTA) control architecture is the motivation for and predecessor to F-SRTA [HLVW03]. SRTA is based on JAF and is far more complex in implementation than Tripbot. SRTA is a design for a system that can efficiently:

- Generate schedules appropriate to resource and time constraints,

- Merge new goals with existing goals and multiplex their solution schedules,
- Efficiently handle deviations in expected plan behavior.

SRTA, at a high level of its control design, contains a loop that operates in the following manner:

1. A domain problem solver that makes use of a TÆMS task library obtains a communication or other sensing action,
2. The problem solver, relying on a TÆMS task library, emits a TÆMS task structure;
3. The task structure is then scheduled by the DTC scheduler, producing a fungible schedule: a schedule that represents a totally ordered plan where the times in the planned (and scheduled) actions can be changed as long as the total order is preserved;
4. A partial order scheduler adjusts and parallelizes the DTC schedule with input from a resource modeler, conflict resolution module, and schedule merging module;
5. A parallel execution module then runs the schedule until completion or failure; on failure, control returns to 1.; and
6. A learning module observes execution and updates the TÆMS task structure library appropriately.

The F-SRTA architecture (instantiated in Tripbot) adds a generic goal reduction and a hierarchical task network solving capability to the the SRTA architecture, giving a domain problem solver additional tools to increase autonomy and flexibility in open, constrained, and uncertain environments.

3.2 Market-Based Control

Although Tripbot does not construct task structures containing plans to bid on parts of trips that it is attempting to compose, such bidding is becoming commonplace and may increasingly become necessary for automated information gatherers. However encoding bidding strategies may turn out to be nontrivial. Other mechanisms to achieve rational bidding may be necessary. Fortunately there

are many. One that allows agents to maintain some flexibility in contracting in a marketplace with asynchronous bids is investigated in [AS98]. This and other research in optimal contract bidding strategies could form the basis of such a component for an information gathering agent such as Tripbot.

A separate set of issues is uncovered when we look at extending Tripbot to handle cooperative multiagent interactions for information gathering. Commitment and decommitment costs are oftent arbitrated for contracts in multiagent systems; this is the standard mechanism for systems employing TÆMS agent control. [SL01] provides analysis of the impact of providing a decommitment cost that allows agents to unilaterally decommit from a task if a better commitment opportunity arises in situations where there may be a range of breaching conditions. This work is extended to include theories of differential information revelation in certain combinatorial auctions in [CS03a]. Some results on the complexity of market cooperation mechanism design are given in [CS03b].

Finally, [GD01] provides theories resulting in policies for communication in multiagent marketplaces. The information source wrapping that was necessary for Tripbot's operation would have been vastly simplified if the necessary information could be queried through XML-based methods. Fortunately, XML is now an increasingly widely used transaction language for online marketplaces; this important role is discussed in [GTM99].

3.3 Information Gathering

Tripbot's objective is information gathering and intelligent data fusion. There is a large quantity of existing research into the general problem of information gathering, from querying multidatabases [ACHK03], to querying global information systems[LSK95], to using softbots as web query interfaces[EW94]. A system that used a planner with some similar capabilities as Tripbots is reported in [BKC$^+$03], which is a theater trip planner with the ability to map the route using heterogeneous information sources.

Although Tripbot does not implement any searches using multiple agents, we are interested in supporting that capability. Using TÆMS for representation and TÆMS agents for single agent

control facilitates the development of larger TÆMS agent systems, since TÆMS was developed with multiagent coordination in mind. [DWS96b] provides guidance in solving the problem of, especially, organizational roles in a multiagent information gathering systems. [DWS96a] provides some insights into the kinds of mechanisms available to provide agent systems with the ability to adapt to dynamic and unpredictable task environments.

We also briefly examined a number of systems that integrate various of the components that we were considering for Tripbot, including complex querying, planning, information source wrapping, and information fusion. We were especially interested in to what extent these systems had integrated such disparate components, and in what way they were tied together in a coherent software architecture. Some of the most interesting are given in the following list:

- Let's Browse, a collaborative browsing system [LDV99] which used Term Frequency Inverse Document Frequency (TFIDF) analysis on the pages that multiple users were viewing (as well as semantically proximate pages on the web) to suggest pages that might be of interest to multiple users simultaneously.

- ShopBot, A shopping Internet interaction agent that has a planner at its core is reported in [DEW97]. ShopBot was interesting for its focus on real-time results and graceful degradation in the case where it was taking too long to produce a result (or if the result could not be parsed) it would provide a link to the page which caused the problem. We used the same approach for the results returned from Tripbot.

- [MTT03] for information on complementing the lexical-semantic approach of query expansion via WordNet, the approach taken by Tripbot, with a standard thesaurus for online information retrieval.

The most influential preceding information gathering project with respect to Tripbot was the Bounded Information Gatherer (BIG), which conceived of information gathering as an interpretation problem [LHK+00]. BIG tackled the complex task of effective information gathering over unstructured information sources.

One key difference between BIG and Tripbot is the existence in BIG of rather sophisticated natural language and general purpose unstructured data parsers. Tripbot uses relatively brittle, simple, handcrafted wrappers to access the information within unstructured sources.

Another key difference is the approach to search. The BIG agent used a customized uncertainty seeker called RESUN, which focused the search in a way to minimize the uncertainty of the presented results in line with user criteria. Tripbot does not have an analogous component.

3.4 Ontologies

Tripbot's ontology was constructed using Java classes, without reference to an existing ontology and without an ontology management system. Although [CCPS99] admonishes that using an established upper ontology is worthwhile, we found that the small number of concepts necessary to model the domain sufficiently prohibited taking a more complex approach. We required less than one hundred classes to represent the domain. Further, the focus of the research was not on ontological inference, so relatively little effort was expended designing for maintenance of large ontologies [STH97]. Given our approach, the caveat is that the few concepts that sufficed to model our domain for the current relatively simple application would likely balloon to a number that would be unmanageable without a better ontology construction and management tool or approach.

3.5 Planning

The focus of the Design-To-Schedule task structure planner was on planning and scheduling with alternatives, probabilistic planning, planning that considers possible replanning, and tractable (fast) planning. In finding an appropriate planner to base our work in planning task structures for Tripbot, several planning systems were surveyed, including Systematic Nonlinear Planner [MR91] derivatives including UCPOP [PW92], Cassandra [PC96], BURIDAN/C-BURIDAN [KHW95, DHW94], and C-BURIDAN, and Hierarchical Task Network (HTN) planners, including Universal Method Composition Planner (UMCP) [EHN94a, EHNT95], O-Plan [CT91], and the Simple Hierarchical Ordered Planner (SHOP) [NCLMA99]. Ultimately, SHOP was chosen as

the basis of the DTS-planner. SHOP is addressed briefly below and in more detail as appropriate in later sections as it applies to the design and implementation of the DTS planner.

3.5.1 STRIPS

Modern AI planning methods may have had their start in the Stanford Research Institute Problem Solver (STRIPS) [FN]. STRIPS introduced a method of planning that is a search for a set of transforms, induced by applicable operators, that moves an initial world model toward a goal world model, where world models are sets of first-order predicate calculus formulas. The standard operator in STRIPS has preconditions that are used to deduce its applicability, and postconditions, that specify how the world model in which it is invoked changes in reaction to the application of the operator. The sequence of operator transformations represents a plan that can then be run in a real world environment, provided, critically, that the world model is valid. Many planners, including Tripbot's DTS planner use a representation for plan operators based on the representation used by STRIPS.

3.5.2 Knowledge-Based, Task Reduction Planning

O-Plan [CT91], was a derivative of Nonlin [Tat77], which was one of the first planners to construct partially ordered plans. Nonline, which was preceded by NOAH [Sac75], addressed the limited control architectures and poor search limiting capabilities of earlier planners. O-Plan is a hierarchical, partial-order planning system with an "agenda-based control architecture" [CT91] where new tasks during planning may be posted to the planning agenda during the search for a plan. O-Plan maintains a plan state which contains the current set of actions, the partial order on those actions, and "flaws" that remain in the plan. O-Plan is also a least-commitment planner [Wel94]. O-Plan also utilizes temporal and resource constraint handling derived from Operations Research to prune the subplan search space. It also uses typed preconditions and a notion of goal structure derived from Nonlin. One of the more remarkable aspects of O-Plan is its "functions-in-context" datastore, which stores histories of operators used to create state changes indexed by world state

information. The world state information is organized hierarchically. Another is the ability of the user of O-Plan to interact with the planner during the planning process.

A very interesting extension of the O-Plan work is in the generation of qualitatively different plan alternatives [TDL98].

3.5.2.1 Simple Hierarchical Ordered Planner

The Simple Hierarchial Ordered Planner (SHOP) [NCLMA99] is an HTN planner that solves a task network through ordered task decomposition. This planner was appealing as a starting point for the development of the DTS planner due to its simple and fast deduction mechanism for testing the applicability of task decompositions and operators, its amenability to mapping its problem solving process into a TÆMS task structure due to its structured, linear decomposition of tasks, and the computational expressiveness supported in preconditions (arbitrary arithmetic computations). The speed of SHOP based on empirical evaluation also encouraged its use [BKS+03].

We explain in a later section some of the extensive modifications we made to the basic algorithms and design used by SHOP, including extending its deduction mechanism to include arbitrary computations at binding points and a heuristic, probabilistic alternative subplan search.

SHOP differs from O-Plan in the following important ways:

1. Complete world state information is not available explicitly at each step of the planning process in O-Plan, so performing arbitrary computations during planning is not immediately feasible and,

2. The means by which plans are constructed is nonlinear in O-Plan, so linearizing the plan steps to provide for alternative generation and interrelationship annotation would be more complicated and might be too computationally expensive than in an ordered task reduction planner.

3.5.3 Graph Planning

Weld [WAS98] identifies two developments as revolutionizing the field of AI planning: two phase Graphplan and methods for compiling planning problems into propositional formulae. The first implementation of Graphplan was for the STRIPS planning representation [BF95]. Graphplan planning occurs in two phases: plan graph expansion and solution extraction. The plan graph expansion phase extends a plan graph forward in time until it has achieved necessary but not necessarily sufficient conditions for plan existence [WAS98]. The solution extraction phase is handled by a search on the graph, that attempts to find a complete plan that solves the problem - specified in STRIPS representation as a conjunct of ground propositions - a goal state or states. Graphplan creates a planning graph starting with a set of the propositions true in the initial world state. Each action that can be applied to that state is applied, creating a second "level" in the graph. The results of each method application are unioned to create a third level in the planning graph. The graph is annotated with preservation and mutex arcs. Then, solution extraction occurs. If there is a solution, planning halts. Otherwise, the next two levels of the graph are created and solution extraction proceeds again. A loop check is necessary to halt Graphplan if there is no feasible plan. The preservation arcs simply mean that a proposition will be preserved through a subsequent action level if no action negates it. These are added primarily to provide for a mutex relation between an action that requires as a precondition the negation of the preserved proposition. Two action instances at a given level are mutex if either:

Inconsistent effects – the effects of one action are the negation of the effect of another action;

Interference – the effect of one action deletes the precondition of another action;

Competing needs – the precondition of one action is mutex with the precondition of another action.

Two propositions are mutex at a given level are mutex if:

Inconsistent support – one proposition is the negation of the other, or if all ways of achieving the propositions are pairwise mutex.

Once new action and proposition levels have been created in the planning graph, the solution extraction phase begins. If the propositions of the goal state exist in the propositions available, Graphplan chooses for each subgoal (proposition literal) one action that achieves the subgoal. If the chosen action is consistent with all other actions at its level, then Graphplan proceeds to the next subgoal; otherwise, it backtracks. Once Graphplan has found a consistent set of actions that produce the subgoals at a given level, it recursively attempts to find a plan for the set of subgoals formed by taking the union of the selected action's preconditions. Planning proceeds until the first level – the initial state – is reached. If no satisfiable plan is found, then the graph is extended. There have been numerous improvements to Graphplan's original formulation as presented in [BF95]. These include for solution extraction forward checking, memoization, and explanation-based learning; and, for planning graph construction compilation of action schemata, regression focusing, and in-place graph expansion [WAS98]. Additional work has been done to augment the expressiveness of Graphplan, including handing uncertainty and sensing actions [WAS98].

There has been a flurry of activity to merge more expressive domain representations with "clasic" Graphplan, including the addition of support for conditional effects [AWS03], probabilistic effects [BL99], and cost sensitivity [DK02].

3.5.4 Buridan

The key difference between Buridan and its predecessors is the fact that Buidan models imperfect information about the initial world state by using a probability distribution over possible world states and a conditional probability distribution to model changes to the world made by plan actions. Buridan's notion of plan construction success changes accordingly from that of producing a plan that provably achieves a goal to one that is sufficiently likely to achieve a goal [KHW95].

So, the process of constructing a plan begins with an intial probability distribution over world states and ends with a final probability over world states in which the goal expression holds with sufficient probability.

Buridan computes a plan-space search in the following manner. Each plan consists of a set of actions, a partial temporal ordering on those actions, a set of causal links, and a set of subgoals to achieve. The set of causal links caches the probability that an enabling proposition produced by one action enables another action. This relationship is denoted as $A_i \rightarrow A_j$, meaning that action A_i produces a proposition, p, consumed by action A_j, or in other words, the probability that p is true at the time A_j is executed. The link is said to provide causal support for A_j. To increase support, Buridan can backward-chain to increase causal support to the triggers of the desired consequence of A_i.

Another approach to probabilistic planning that claims an order of magnitude speed improvement over BURIDEN is MAXPLAN [ML98]. It accomplishes the speedup through a step that transforms the problem into a form that can be solved through highly refined boolean satisfiability methods that make extensive use of dynamic programming.

3.5.5 Hierarchical Task Network Planning

[EHN94b] describes the problem and solutions offered by a very general approach to Hierarchical Task Network (HTN) planning. A method for learning method preconditions for HTN planning is given in [INMAA02]. Kambhampati in [Kam95] compares HTN, or task reduction planning with partial order planning and gives a framework for planning system design tradeoffs in [KKY95]. The DTS task structure generation algorithm is based on the HTN planning done by SHOP [NCLMA99].

3.5.6 Other Relevant Planning Results

See [BN03] for approximate planning in unstructured stochastic spaces, which is especially applicable to the Tripbot domain. See [DK01] for planning as constraint satisfaction. See [DK03]

for multiobjective Graphplan, metrically constrained planning. For compiling planning to SAT problems to solve using fast heuristics, see [EMW97]. A discussion of reviving partial order planning is given in [NK01]. For planning as controller synthesis, see [GO01]. Another approach to planning with incomplete domain information, as is the case in the Tripbot domain, is given in [EWD+92]. Finally, [RM01] discusses decision theoretic planning with temporally extended actions. An early AI planning language that echoed some of Brooks' [Bro91] ideas of behavioral levels was the Reactive Plan Language [McD03]. A description of some techniques from computer learning as applied to planning can be found in [ZK03, EM96]. Least cost plan repair is described in [JP94].

3.6 Scheduling

3.6.1 Design-to-Criteria Scehduling

Design-To-Criteria scheduling [Wag00] handles resource bounded agent control. Resource boundedness refers to the existence of deadlines, cost limitations, or application specific resource limitations (like limited network bandwidth). We are concerned with online reasoning about these limitations when generating plans and schedules because agents often exist in changing environments, both task and nontask. DTC is not hard real-time nor is it fault tolerant, at least in the schedules that it produces, but it does support fault-tolerant reasoning through its probability distributions on quality, cost, and duration. DTC uses a battery of techniques to manage the worst-case exponential complexity possible in TÆMS analysis, including:

Criteria-directed focusing – soft criteria are used to focus the search for schedules;

Approximation – schedule alternatives are used to provide a coarse overview of the schedule space; and

Heuristic decision making – DTC uses a set of action rating heuristics to reduce the exponential complexity of exhaustive schedule construction and analysis; and

Heuristic error correction – corrects errors in suboptimal schedules.

3.6.2 Other Relevant Scheduling Results

Soft real-time scheduler construction is reported in [RS03]. Also, see [BH03], [BM98], and [BDS94] for a discussion of relevant Just-In-Case scheduling. ASPEN is an interesting system with planning and scheduling systems to support space mission operations [CRK+03].

Chapter 4

FLEXIBLE SOFT REAL-TIME ARCHITECTURE

The Flexible Soft Real-Time Architecture (F-SRTA) an extension of the Soft Real Time Agent (SRTA) architecture [HLVW03]. What principally differentiates F-SRTA from SRTA is a domain-independent, time-bounded task structure planning component.

A software architecture, as mentioned previously, is a class or style of system design. And, an agent architecture describes a set of coupled components and the way that the component capabilities can be used together in a configuration that supports a particular kind of agent behavior within the basic agent state trinity of sensing, deliberating, and acting. In many circumstances, in order to act reasonably, an agent must "know" (or be able to figure out) how long to sense, act, or deliberate. An agent's components thus should expose their computational requirements in order for the proper solution time versus solution quality tradeoffs to be made. Some important questions pertain to the way components interact with one another:

reflectivity Does the architecture provide components with the ability to detect their own state and the state of other components in the architecture. For example, does the architecture support components that can query each other's current and expected computational loads?

representation Can the architecture make use of the information provided through its reflective interface? Is it easy for a domain modeler to understand how to use information in a model? How fast can data be transferred between one component and another? Are there complex transformations required?

extensibility How easy is it to replace an old or add a new component to the architecture? Can modules be swapped based on the detection of a problem pattern whose solution will be provided better by one component versus another? Is the problem solving interruptible?

security Are the component interconnections secure? What is the trust and security model between components? How does the agent insure the component does only what it is required to do; i.e., can it run in in a sandbox?

We emphasize the importance of representing computational complexity attributes for component state in order to make continual decisions about their use in an open agent architecture. An agent should be able to tune its components to provide better or faster results as user requirements demand.

TÆMS and TÆMS agents provide a theory flexible and testable general representations and computing systems for control of complex systems. TÆMS began as a language to model and simulate a Distributed Vehicle Monitoring problem which then was being solved by a distributed network of blackboard problem solvers [CLL03]. TÆMS has evolved considerably since then, especially through many iterations of its use in implementation of the DTC scheduler and the Java Agent Framework [Hor03, WL01]. On one end of the spectrum, TÆMS technologies are pushing into the realm of analysis of resource allocation problems within millisecond control loops for hard real-time applications, and, on the other, they are being pushed toward the analysis of motivations for large, complex, agent organizations [HVM+01, WL03]. The addition of the DTS planner adds a primarily generative TÆMS component for the modeling and control of TÆMS agent systems.

The Soft Real-Time Agent (SRTA) control architecture has focused on coordination and harder real-time responsiveness, whereas the focus of the F-SRTA project has been demonstrating if and how the TÆMS formalism and software framework can be extended to support goal and criteria directed, generalized task structure planning to be used in conjunction with DTC scheduling. Thus, the purpose of extending the SRTA architecture is to support goal and criteria directed TÆMS planning in order to achieve a higher level of intelligent autonomy for TÆMS agents, e.g., [PBC+03].

4.1 Tripbot

We now turn our attention to the instantiation of the F-SRTA architecture that we used to solve an online trip planning problem: Tripbot. We first describe in some detail the problems that Tripbot solves. Then, we discuss the operation of Tripbot at a component level in detail.

4.1.1 The Problem

Tripbot, in input/output terms, accepts a desired trip query specification as input and then attempts to produce as output a list of ranked itineraries. In our casting of the problem, there are two basic subproblems between the user's query specification and providing the user a set of ranked itineraries: gathering appropriate data based on the query and fusing the data to generate trip itineraries. These two problems are described as: a *data gathering problem* and an *information generation problem*[1]. Both problems are hard combinatorial optimization problems and balancing system responsiveness and the goodness of the solutions produced is difficult. Other problems encountered in developing the system are:

- Understanding and characterizing the form for user input to the data gathering and information generation problems. This is mostly a problem of mapping user query ideas closely into a form that can be used by the computer system to guide its search for appropriate data and information.

- Generic data gathering based on a partially qualitative user query. Given that we have the user query in a computer useable form, we then have the problem of using it to gather data from an open system. A generic solution to this problem turns out to be very complicated, and not the most important thrust of our research, so we developed an approximation to a generic solution – customized data source wrappers.

- Control solution instantiation for the gathering of data and fusion of information. The components alluded to above – the task structure planning and scheduling components –

[1]See Appendix B for a more formal treatment of the problems.

will generate control solutions, but then the system using them needs to actually instantiate the schedules produced, and control the flow of information produced from running actions.

4.1.2 The Solution

Our discussion thus far has been biased toward an agent-oriented solution, but it is worthwhile to mention that there are other, non-generic solution approaches. A custom set of algorithms could be generated to gather data and generate the information required. However, problems requiring schedules of action, which must operate in dynamic environments, including, especially, environments which demand time constraints on computation and other system utilization, are appropriate candidates for agent-oriented design and development[Wag00]. And, since we are interested in developing generic solutions to a broad class of problems through the use of a reusable solution framework, we adopted the agent-oriented approach.

Tripbot solves its data gathering and information generation problems through a reusable framework of agent-oriented components that we collectively label a TÆMS agent. The TÆMS agent components use heuristic-oriented methods that manage the computationally hard problems of task structure generation, schedule selection, and result combination in an *open*, *uncertain*, and *computationally constrained* environment.

Tripbot is thus an instantiation of the F-SRTA architecture, although some of Tripbot's implemented component interfaces are not generic. The Tripbot agent architecture is given in Figure 4.1. The components of Tripbot:

Query Interface — an HTML page and a Perl CGI script to parse parameters from the HTML page;

Query Processor — Java classes that read the Perl CGI generated parameter file, find appropriate synonym sets (synsets) for some of the query keywords, create a query resource map and a problem definition to send to the DTS planner;

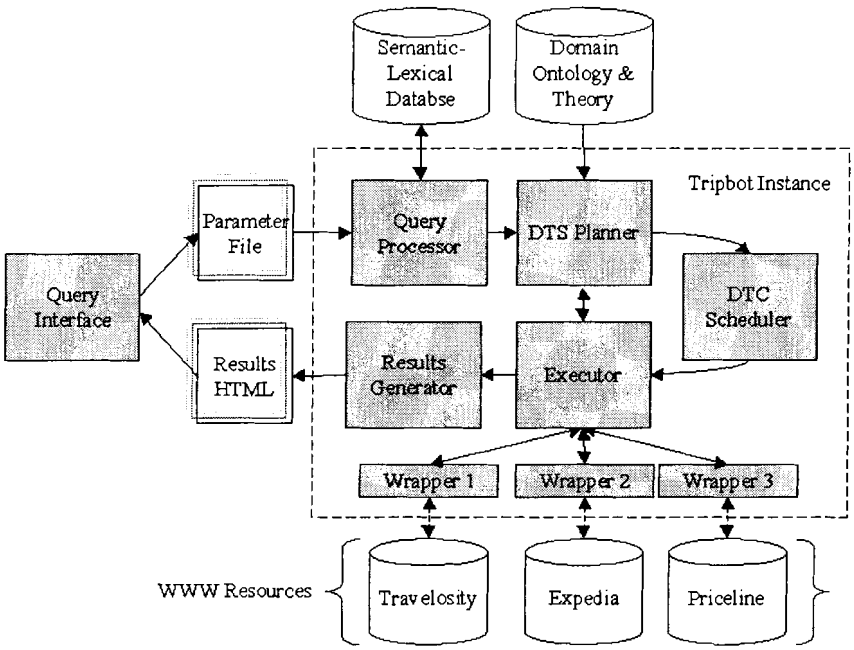

Figure 4.1: The Tripbot agent architecture.

Design-To-Schedule (DTS) Planner — Java classes that generate task structures and Information Resource Tables (IRTs), described later, from Query Processor produced problem definitions and provided domain definitions;

Design-To-Criteria (DTC) Scheduler — a C/C++ stand-alone application that reads in the Textual-TÆMS (TTÆMS) – the human-readable, textual form of TÆMS– task structure produced by the DTS Planner and creates a set of schedules in TTÆMS format;

Executor — Java classes that drive both the planning and scheduling processes, maintain resource flow, and trigger replanning and rescheduling operations;

Wrappers — information resource wrappers that mediate between internal and external information representations; and

Results Generator — Java classes that fuse the disparate results received from information source wrappers into coherent presentations, where in Tripbot, those results are trip itineraries.

Other minor aspects of the architecture, hidden within the Executor, include the IRT subsystem, to handle the aggregation of results from action in the world – a working memory in Belief, Desire, Intention (BDI) architectural terms – and a generic Java reflection instantiation and execution subsystem that is responsible for binding scheduled items to objects that can be run by the Executor.

Most of Tripbot's components are written in Java and run in a Java Virtual Machine (JVM). A separate instance of Java classes runs in a separate JVM for each query performed. To handle multiple queries simultaneously, the Query Interface creates a separate, unique prefix for all output files pertaining to one query run. The files with that prefix are deleted by the Executor once the Results Generator returns from its computation. This is an admittedly primitive system to allow multiple queries, but that problem, although interesting, is not part of our focus. Once the query parameters file has been generated, the Perl CGI side of the Query Interface starts a new JVM whose main application thread is the Executor. The application flows roughly as follows from that point:

1. The Executor instantiates and calls the Query Processor to read in the query parameters and to process them. The input to the Query Processor is a file descriptor.

2. The Query Processor expands the query using the Wordnet lexico-semantic dictionary [MBF+98] as well as some hand-coded relations between query input words and words that will produce good results at web resources. Using a template created specifically for the information gathering domain, it also creates a DTS planning problem definition which it returns to the Executor.

3. The Executor instantiates an Information Resource Table (IRT) and adds resource values obtained from the Query Processor including the query key/value pairs from the query input form, the time bound for the query process to produce results, the number of itineraries to produce, and the DTS planning problem definition.

4. The Executor then initializes the DTS planner with domain information, the planning problem generated by the Query Processor, the initialized IRT, and a time bound for computation. The time bound for the DTS planner, for its use in Tripbot as a solver to the data gathering problem, is the larger of 20 seconds or 30% of the time between the time it is given a problem and the time that the user expects a result. A more justifiable value would be the product of additional controlled experimentation, but remains highly domain- and problem-dependent. For the DTS planner experiments described later, no time bound is given to the planner so that the runs with each alternative selection heuristic run to completion.

5. The DTS planner produces a set of plans and schedules, encoded in a TÆMS task structure. It optionally produces a Method State Map (MSM) which contains a mapping from every given action to a plan worldstate to be used for replanning and rescheduling and a Method Type-Instance Map (MTIM) to be used by the executor to instantiate the methods that are ultimately scheduled. It also adds containers for the results of method execution to the IRT.

6. The Executor then tells the DTC Driver to run the DTC scheduler on the task structure generated by the DTS planner.

7. The DTC scheduler emits a set of schedules and each schedule's rating.

8. The Executor reads in the schedule or schedules, picks the first (best) schedule, and then merges the method identifiers in the schedule with the MTIM to create a runnable action sequence. It also generates some schedule-wide statistics for rescheduling invocation. Rescheduling policies based on these statistics are described in a later section.

9. Once an Action Sequence has been created, the Executor runs it, monitoring actual method characteristics against expected method characteristics, triggering rescheduling or replanning sessions if necessary.

10. Once the Action Sequence has completed information gathering activity, the Executor

passes the IRT to the Results Generator which creates itineraries from the accumulated information results and returns them to the user.

This describes the basic flow of Tripbot control and information. We now detail each component of Tripbot in terms of its control and information flow. Information about componentwise time and space complexity within a given invocation context is required to decide how much computation time and space should be allocated to one component versus another. Ideally, we would be able to closely estimate the computational complexity of components in an arbitrary context of operation, instead of characterizing its worst or average case performance. However, doing this in general is a difficult and likely undecidable task. So, worst case Θ time complexity is also given for each component, motivating a discussion of the parametric combination of components in a solution to the metacontrol problem.

4.1.3 Query Interface and Query Processor

The Query Interface operates in one of two modes, query input or results presentation. In its query input mode, it presents an interface to its user where query parameters can be set. Some of the query parameters are:

- Departure city and destination city, if desired,
- Departure and return dates and times,
- The total trip expense limit, and
- Preferences for airline, car rental, and hotel brands;
- Some keywords describing the kind of trip desired.

In its results presentation mode, the Query Interface presents the users of Tripbot with a set of possible trip itineraries, from airline reservations to car rental reservations, to day to day bookings for their stay at the destination and along the way.[2]

[2]There are some booking sources for which this approach is somewhat more limited, e.g., Priceline, where a commitment to fly the flight that is found for you must be made before the specifics of the flight are known.

The Query Processor takes the "raw" query from the user and then performs two major functions:

Query to problem translation — uses query values from the interface to:

- Set up the information resource table template — the template used by the DTS planner to generate a table with typed entries for gathered data, and
- Generate problem instances — an initial state and goal(s) specification.

Lexical-semantic keyword processing — uses user-specified keywords to generate synsets for queries against data sources that provide trip itinerary subsolutions based on keywords, e.g., state park sites that have search capability and that will provide a list of state parks with beaches based on a "beach" query.

Wordnet [MBF+98], a lexical-semantic dictionary, is used in the Query Processing phase to expand an initial set of query keywords given by the user. Using a lexical-semantic dictionary to expand a keyword search is necessary to cover the cases where semantically relevant information might not be directly referenced in an initial keyword set. However, with respect to the expanded set of keywords, there is an obvious question about which keywords to use in which keyword queries.

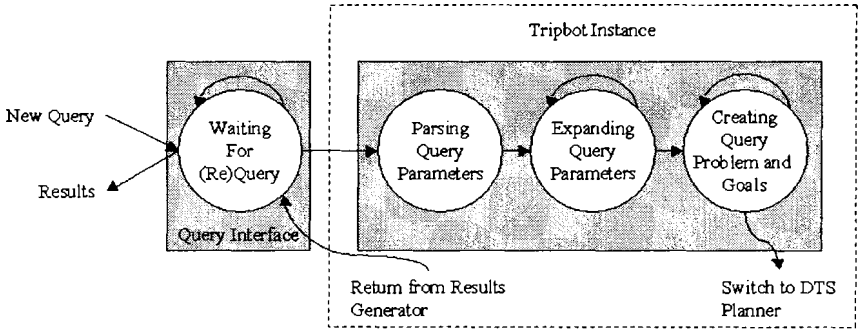

Figure 4.2: Query Processor states and control flow.

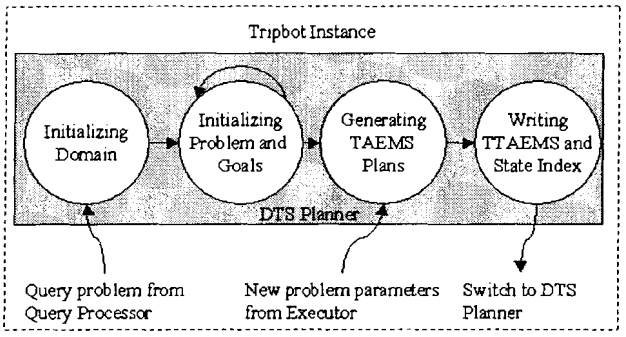

Figure 4.3: DTS Planner states and control flow.

The use of Wordnet requires, upon its first use, an initialization step that usually requires less than 1 second to complete. Further, information related to a given query word is usually returned from the Wordnet interface in less than 1 second. So we treat each Wordnet query operation as a $\Theta(1)$ operation in the average case. Interface generation and presentation of itineraries resulting from data gathering and information generation activities are $\Theta(n)$ operations in the average and worst cases, being linearly dependent on the number of itinaries requiring presentation.

A basic states, control, and information flow view of the Query Interface and Query Processor is given in Figure 4.2.

4.1.4 DTS Planner

The DTS planner takes as input a query problem and an information gathering goal from the Query Processor. The DTS planner, when invoked with a new problem, loads the Tripbot domain theory — a collection of task decomposition rules and state axioms. The DTS planner then generates plans encoded in a TÆMS structure and emits them to be scheduled by the DTC scheduler. It also writes to a file an IRT for the gathered information and a MSM, if requested[3]. A states and control flow view of the DTS planner is given in Figure 4.3. Given a problem, domain theory initialization is a $\Theta(n)$ operation in the size of the domain.

[3]The MSM adds considerable space to the planning operation, so it is excluded for large problems. More experimentation is required to determine if and when the MSM is necessary.

The hardness of a given DTS planning problem is governed by five factors:

- The task network (decomposition) structure, including the alternative decomposition branching factor (a parameter varied in tests described later),
- Alternative bindings in the domain for a given task decomposition state,
- Ordering constraints on task decomposition;
- The number of variables instantiated at terminal tasks (methods), and
- The size (at binding query time) of the problem state.

Since the DTS planner is at its core an ordered HTN planner that supports variables, previous analysis of HTN planning applies, especially [EHN94c, EHN94b], which establishes its EXPSPACE complexity and semidecidability. DTS does add an alternative subplan selection to SHOP's HTN planning approach, where alternatives can be enabled by the same preconditions. This additional functionality does not affect the EXPSPACE result that applies to SHOP since the alternative point is just a special type of task decomposition node. Clearly in the case where DTS is tasked with providing an optimal selection of n subplan alternatives at such an alternative point, the number of subplan decomposition operations at each alternative point increases from n to 2^n. To verify its optimality the task structure must be scheduled. TÆMS task structure scheduling is hard [WL01] (and proved NP-hard using TÆMS formalism in the Appendix).

4.1.5 DTC Scheduler

A states and control flow view of the DTS Planner is given in Figure 4.4. Although task structure scheduling is NP-hard, some cases, such as the case where methods have discrete duration and monotonically increasing cost inversely proportional to duration, the solution is approximable in polynomial space to within $(3/2)\log(l)+(7/2)$, where l is the the ratio of the maximum allowed duration of any activity to the minimum allowed non-zero duration of any activity [Sku97].[4]

[4] [Wag00] gives an upper bound approximation to the time complexity for worst case disjunctive method scheduling, by Stirling's approximation, as $o(n^n)$.

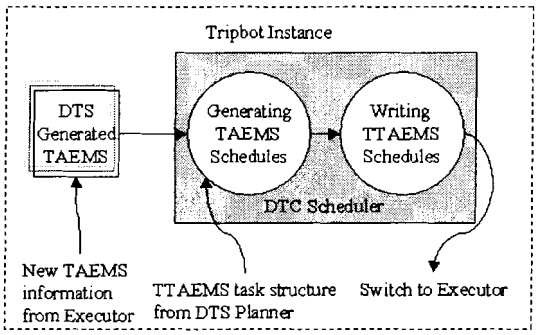

Figure 4.4: DTC Scheduler states and control flow.

4.1.6 Executor

The executor is where the control and information flow come together. A states and control flow view of the Executor is given in Figure 4.5. This component represents the nexus of unified information flow in task networks of the kind described in [WDS96]. Specifically, in Tripbot,

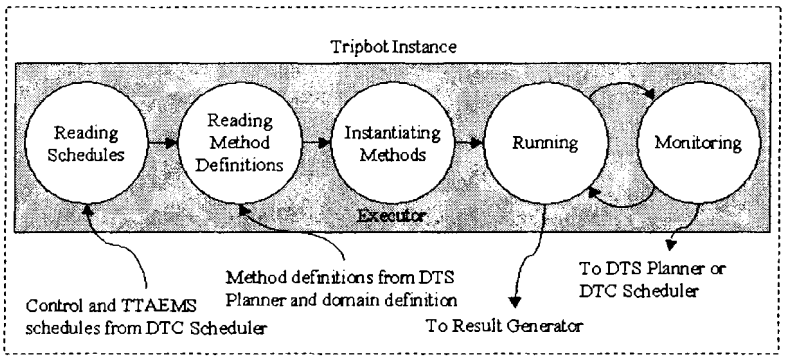

Figure 4.5: Executor states and control flow.

the Executor merges abstract schedules with type information in the MTIM to create runnable actions, stores data gathered through the instantiated actions in the IRT, and invokes rescheduling or replanning actions when schedule failure is detected according to explicit criteria. In both the average and worst cases, the time complexity of the merge operation is $\Theta(n)$, where n is the number of methods in the agent's currently scheduled task structure, since method information in

the MTIM and IRT is accessible through hashtables in average time $\Theta(1)$, and worst case time $\Theta(n)$, with the hash table implementation used[CLR90].

4.1.7 Results Generator

A states and control flow view of the Results Generator is given in Figure 4.6. The results generator presently uses a constraint satisfaction technique reported in [Qia02].

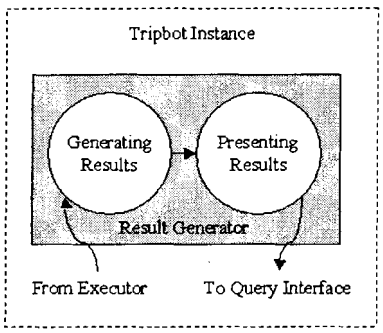

Figure 4.6: Results Generator states and control flow.

4.2 Complexity, Compute Time, and Quality

Giving average- and worst-case computational complexities for most of the agent components is somewhat misleading, since the overall complexity of F-SRTA/Tripbot's operation is dominated by the DTC scheduler, whose problem is NP hard, and the DTS planner, which, at least in theory, may never halt.[5] Further, in terms of its value to runtime in a specific problem context, it does not prove very useful [TH03]. In an attempt to address the complexity of of problem solving methods in an application context, [TH03, HT98] propose a formal framework for characterizing the expected performances of anytime algorithms. This analysis does not apply well to the DTS planning method or the DTC scheduling method. It doesn't apply to either because they meet, on a given problem instance, at best only the consistency property of Zilberstein's desirable properties

[5]Fortunately, in practice, this unfortunate condition can be guarded against.

of anytime algorithms: interruptability, monotonicity, measurable quality, diminishing returns, consistency, recognizable quality, and preemptability. The DTS planner, given the same duration to compute on a given problem, will consistently return the same result. The same can be said of the DTC scheduler on simple problems, but not on large problems, if it is running in a heuristic mode, since it uses random sampling techniques to generate subschedules in those instances.

The attempt to produce an accurate runtime characterization is additionally thwarted by the high variability of runtime across "random" problem instances. This is due partly to the structure of the problems and partly to the use of heuristics to generate solutions. For instance, in the DTS planner, the inclusion of one additional primitive goal decomposition can increase the plan runtime exponentially.

There is no obvious magic bullet to solve this problem, since detecting nontrivial attributes of problem instances posed in a sufficiently powerful language — one that is a subset of the set of recursively enumerable languages — is undecidable [HU79].

Chapter 5

DESIGN-TO-SCHEDULE PLANNING

This chapter describes the Design-To-Schedule (DTS) planning algorithms that generate task structures to solve problems in uncertain domains. DTS planning[1] seeks to provide an effective set of plan families, encoded within a hierarchical task structure, to a TÆMS characteristic criteria optimizing scheduler. Specifically, the DTS planner generates a TÆMS task structure which can then be analyzed by a Design-To-Criteria task structure scheduler. Thus, a key problem solved by DTS is the transformation of goals within the context of a problem into TÆMS task structures which can then be scheduled and run by an agent.

The complication in performing these actions are the constrained computational environment of the agent and the usual need for timely action. The deliberative approach adopted by DTS complements reactive planning approaches[Tur03]. Since the time and space complexity of the DTS planner and DTC scheduler dominate the time and space complexity of other F-SRTA components, the key question explored in DTS is the question of how to balance complexity, computation, and utility between its own solving process and that of the DTC scheduler.

This section describes the DTS planner language and procedures in some detail, including the interfaces between the DTS planner and other F-SRTA/Tripbot components. First, the application of planning algorithms to TÆMS task structure generation is motivated. Second, the language

[1]Task structure generation is herein referred to as planning, since its task, naively, is goal reduction to partially ordered plans, and the core of its present capability is a hierarchical planning algorithm. The problem is more general than planning, however, since it may deal with domains where planning approaches are inappropriate.

and representation used by the DTS planner are described. Then, our focus turns to a detailed explication of the DTS planning procedures.

5.1 Applying Planning to TÆMS Task Structure Generation

One problem with existing approaches to TÆMS agent control is the limited generic flexibility available in current approaches to goal-directed TÆMS agent control, specifically through task structure generation. To date, TÆMS agents have generated task structures through many approaches, but, in all cases, the generation approach has not been generic nor particularly reusable in the same way that the DTC scheduler has been.

TÆMS can encode a very large number[2] of nontrivial schedule alternatives within one task structure, but the fact that the task structure remains largely static, disregarding method completion annotations, finally limits the schedule alternatives available to an agent. A key problem is that once a TÆMS task structure has been generated, there is no generic mechanism through which new, nontrivial schedule alternatives may be generated.

If an agent was capable of creating a perfect schedule at the beginning of its mission, the limited alternatives set available for rescheduling would not matter. However, that is usually not the case. Often in real-world domains, partly due to the fact that a domain may never be static or able to be modelled perfectly, an agent will require an adjustment to initially conceived schedule of action and, possibly, plans to schedule from to solve a problem.

To demonstrate: consider a problem example from the probabalistic blocks world (PBW) domain. This is a simple domain that we created to test the interaction between the DTS planner and the DTC scheduler, since doing so using the Tripbot domain, which is much more complex, was prohibitive in the time required to plan and schedule single instances of the problem.

[2] In fact, an *infinite* number of schedule alternatives can be encoded in a TÆMS task structure under circumstances where no task deadlines exist, however little practical value nearly all of the schedules in such a set might have. Hence, we qualify the alternatives that are of interest as *nontrivial* alternatives.

A PBW problem is a 4-tuple (W, T, S_1, G_F). W is a set of world elements. In this example:

$$W = \{BLOCK_1,\ BLOCK_2,\ BLOCK_3,\ TABLE\}$$

T is a domain theory. The domain theory provides a set of functions on logical predicate sets in the domain; the functions map one predicate set to a set of predicate sets, which may be the null set. PBW differs from classic blocks world, which is described in [RN91] in that it provides a subset of three methods for moving a block that are probabilistically characterized: $fast - and - shaky$, $average$, and $slow - and - steady$. The $fast - and - shaky$ method is faster than the other methods, but will fail more often than not. The $slow - and - steady$ method is slower than the other methods, but will nearly certainly not fail. Finally, the $average$ method has average speed and average reliability.

S_1 is an initial state, which is a set of predicates. In this example, again, the initial state is defined as:

$$S_1 = \{(on\ BLOCK_1\ TABLE)\ (on\ BLOCK_2\ BLOCK_1)\ (on\ BLOCK_3\ TABLE)\}$$

G_F is a final, goal, state. In this example it is also specified a set of predicates:

$$G_F = \{(on\ BLOCK_3\ TABLE)\ (on\ BLOCK_2\ BLOCK_3)\ (on\ BLOCK_1\ BLOCK_2)\}$$

For the DTS planner, this state would undergo a trivial transformation to a list of achieve goals:

$$G_{achieve} = \{(achieve\ G_F)\}$$

The problem is to determine at least one sequence of functional transformations consistent with T that maps S_1 to S_F. The minimal satisfying result is a sequence of grounded transformation

functions, $\{f_1, f_2, \ldots, f_n\}$. Grounded functions are functions whose variables are parameterized with ground substitutions. A ground substitution is a sequence of substitutions that ends with the substitution of a variable term with a predicate literal; for our purposes, predicate literals are elements of W or functions from T whose terms are grounded.

Abstractly, the ability to generate agent task structures from goals to perform a mission adds an element of flexibility and autonomy to the agent, but then concerns alluded to above expand to include design-time tradeoffs between the expressivity, generality, and performance of the interoperating systems. As mentioned previously, and echoing [Wd03], the use of hierarchical task network (HTN) planning as the basis of the approach seemed to provide the best balance of these concerns. Using an HTN planning approach would provide sound ground for a solution to the task generation requirements of the Tripbot data gathering and information generation problems, and it would also lend itself to reuse within the F-SRTA agent architecture. This is so because of several benefits of choosing HTN planning as the basis of DTS:

- The similarity between the intermediate structures used in generating HTN plans and TÆMS task structures.
- The flexibility of the domain representation.
- Performance against other types of planning systems[BKS+03, LFS+03].

As mentioned previously, hierarchical, ordered planning with a formalism at least as expressive as STRIPS presents us with EXPSPACE complexity in the worst case — a high computational cost for completeness guarantees in complex, real-world domains. To expediently provide a rough sort of control over the computational cost of task structure planning, the algorithm was modified to support time-bounded computation. There are many ways to support this, ranging in sophistication. Due to the fact that the planning algorithm that DTS inherited from SHOP does ordered task decomposition, there is a natural agent-based, anytime character to its plan search. So, as a first cut, DTS simply stops computation within a Δt bound of the compute deadline it is given. Some questions that arise given a soft real-time constraint to provide a result by time t are:

- How much time should be spent planning given a criteria and sampling policy?
- How should alternatives be sampled given a criteria and a time constraint?
- How much time should be spent scheduling for a given criteria?
- Are there criteria that produce better schedules for a given time constraint?
- Are there task structures generated under criteria and time constraints by a DTS planner that allow the DTC scheduler to produce better schedules for a given time constraint?

The planning phase can encode a large number of plans in a TÆMS task structure. To illustrate this, however, let's look at two simple structures that could be generated in a DTS planning phase.

The first task structure, depicted in Figure 5.1, is a task structure generated for the Probabilistic Blocks World (PBW) domain and contains no alternatives. Next, we look at what we can achieve

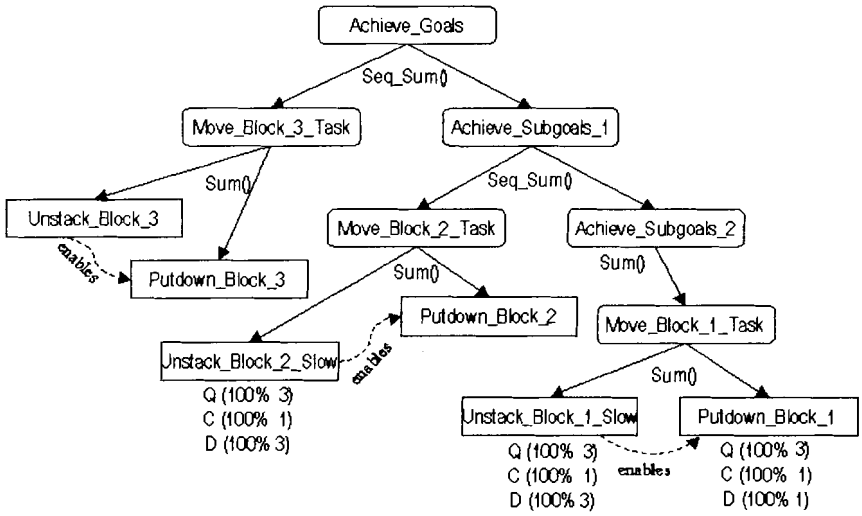

Figure 5.1: Probabilistic Blocks World TÆMS task structure with ***no*** alternatives.

if we spend more time planning. Planning can expand the task structure in a number of ways. Presently, planning focuses on two means to offer the scheduler more alternatives:

Alternative method inclusion – this is a zero-cost operation, which simply entails indexing multiple methods with a form that can be identically unified with an outstanding form.

This guarantees more choice at the cost of higher computational complexity — larger state space and longer runtime.

Alternative plan generation – this is a potentially costly operation. If there are multiple means of performing a task in a plan state, the planner can opt to explore those alternatives. There is no guarantee that a complete alternative will be found, hence there is some uncertainty in this method.

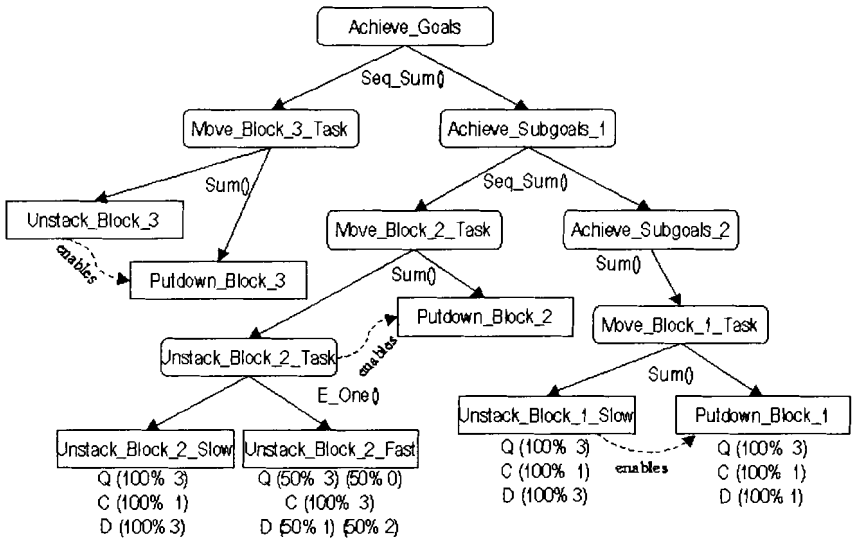

Figure 5.2: Probabilistic Blocks World TÆMS task structure with alternatives.

With the second task structure, depicted in Figure 5.2, we have an additional option that can be encoded in a separate schedule, providing for an on-line alternative, if the opportunity to reschedule an alternatives arises. An important problem in the alternative generation stage of task structure planning is deciding which alternatives to include. In the case where no appropriate task alternative exists in the task structure, a replanning session can be used to generate a new task structure based on knowledge of the current state.

5.2 Language and Representation

There are three parts to every search problem: representation, objective, and evaluation function [MF99]. Two extensions to the TÆMS language were created to provide the planner with a representation sufficient for defining the planning domain and for defining a planning problem. Further, the previously discussed IRT and MTIM extensions were required to allow a plan to be run after being scheduled.

5.2.1 Domain Definition Language

The domain definition language is the language that encodes an agent's options to affect the world. In the case of DTS planning, because DTS planning is ordered, hierarchical task network planning, the domain definition language provides for tasks. Tasks are the principle task decomposition mechanism. Tasks are the interior nodes of the task decomposition network, whereas methods are the terminal nodes, representing the primitive or atomic actions that the agent can take, and axioms - predicates, i.e., functions that take arbitrary input and that return a Boolean value as their result. From a base of HTN-style planning, DTS adds heuristic search capability, reasoning about the expected value and variance of attribute tuples and time-boundedness, meaning that DTS will generally provide the DTC scheduler a larger schedule space if more time is given to its search.

5.2.1.1 Tasks

Tasks represent compound actions for an agent in a given domain. Important parts of a task representation are its form and decomposition.

The Backus Naur Form (BNF) for tasks is given in Figure 5.3. Tasks essentially populate task networks, providing complete decompositions for goals. The DTS planner performs a recursive, depth-first search over the task network induced by task decompositions. All choice points in the planner's search space are encoded within tasks. A short description of each element of the task BNF follows.

```
<TASK>            ::= '(' spec_task <DOMAIN_ATTRIB> <TASK_FORM>
                      <DECOMPOSITIONS> ')'
<TASK_FORM>       ::= '(' form (' <LABEL_STRING> <VAR_STRING> (' '
                      <VAR_STRING>)* ')' ')'
<METHOD_FORM>     ::= '(' '!'<LABEL_STRING> <VAR_STRING> (' '
                      <VAR_STRING>)* ')'
<DECOMPOSITIONS>  ::= '(' decompositions ( '('<LABEL_STRING>
                      <DECOMPOSITION> ')' )+ ')'
<DECOMPOSITION>   ::= '(' <LABEL_STRING> <PRECOND>
                      <INTERRELATIONS> <TASK_DECOMP> ')'
<TASK_DECOMP>     ::= '(' decomposition
                      (<TASK_FORM>|<METHOD_FORM>)* ')'
<DOMAIN_ATTRIB>   ::= '(' domain <LABEL_STRING> ')'
<LABEL_STRING>    ::= ([A-Z,a-z])+([`_',0-9,A-Z,a-z])*
<PREDICATE_LIST>  ::= '(' (<PREDICATE>)* ')'
<PREDICATE>       ::= '(' <LABEL_STRING> <VAR_STRING>
                      (' ' <VAR_STRING>)* ')'
<VAR_STRING>      ::= '?'<LABEL_STRING>
<PRECOND>         ::= '(' preconditions <PREDICATE>* ')'
<INTERRELATIONS>  ::= '(' interrelations <INTERRELATION> ')'
<INTERRELATION>   ::= '(' <IR_TYPE>  <IR_LIST> ')'
<IR_TYPE>         ::= 'ENABLES' | 'DISABLES' | 'FACILITATES'
                      | 'HINDERS'
<IR_LIST>         ::= '[ [0-9]+(2)+ ]'
```

Figure 5.3: DTS Planner task definition in Backus Naur Form.

TASK — the start element

TASK_FORM — a form in a task list

METHOD_FORM — a form in a task list preceded by a special character

DECOMPOSITIONS — the available task networks that may solve a task

DECOMPOSITION — the structure of one subtask that may solve a task

TASK_DECOMP — a subtask list to solve the parent task

DOMAIN_ATTRIB — the TÆMS domain of the task

LABEL_STRING — the task's name (appended with a gensym when instantiated)

PREDICATE_LIST — a list of logical predicates

PREDICATE — a logical predicate

VAR_STRING — a string beginning with a '?', signifying a variable

PRECOND — a decomposition applicability precondition

INTERRELATIONSHIPS — TÆMS interrelationships between subtask decompositions

INTERRELATIONSHIP — one type of interrelationship and lists of nodes for which it holds

IR_LIST — lists of at least two task indices, that indicate, for each list, that an interrelationship exists starting with the first index and ending with each of the remaining indices

```
(spec_task
     (domain tripbot)
     (form (fly ?c1 ?c2))
     (decompositions
          (decomposition_0
               (preconditions
                    (city ?c1) (city ?c2)
                    (need-flight-from ?c1)
                    (us-flight ?c1 ?c2 ?attribs-us)
                    (delta-flight ?c1 ?c2 ?attribs-delta)
                    )
               (decomposition
                    (or [(!fly-delta ?c1 ?c2 ?attribs-delta)
                         (!fly-us ?c1 ?c2 ?attribs-us)]))
          )
          (decomposition_1
               (preconditions
                    (city ?c1) (city ?c2)
                    (need-flight-from ?c1) (us-flight ?c1 ?c2 ?attribs))
               (decomposition (!fly-us ?c1 ?c2 ?attribs))
          )
          (decomposition_2
               (preconditions
                    (city ?c1) (city ?c2)
                    (need-flight-from ?c1) (delta-flight ?c1 ?c2 ?attribs))
               (decomposition (!fly-delta ?c1 ?c2 ?attribs))
          )
     )
)
```

Figure 5.4: DTS Planner task definition example.

To illustrate the task definition concretely, Figure 5.4 is provided, which contains a simplified version of a domain task definition for the Tripbot domain. In Figure 5.4, the domain is specified as `tripbot`. The form is given as `(fly ?c1 ?c2)`, which can be unified with a supertask's bound decomposition element. Then, the decompositions are given. This task decomposition specification checks to see if a flight is needed from the value of `?c1` first, in each decomposition case. Then, it first checks to see if there exists a USAir flight, `us-flight ?c1 ?c2 ?attribs-us)`, as well as a Delta flight, `(delta-flight ?c1 ?c2 ?attribs-delta)`, in order to reduce the `fly` goal to an alternative point where a selection computation and decision can be made. If both types do not exist, but one type does, then it is chosen. If no `fly` decomposition is applicable in the world state, the decomposition for this branch halts.

5.2.1.2 Methods

Methods represent primitive actions in a given domain. Important parts of a method's representation are its form, preconditions, delete list, add list, and outcomes. The BNF for methods is given in Figure 5.5.

```
<METHOD>           ::= '(' spec_method
                         <DOMAIN_ATTRIB>
                         <METHOD_FORM>
                         <PRECOND>
                         <ADDLIST>
                         <DELETELIST>
                         <OUTCOMES> ')'
<METHOD_FORM>      ::= '(' '!'<LABEL_STRING> <VAR_STRING> (' '
                         <VAR_STRING>)* ')'
<ADDLIST>          ::= <PREDICATE_LIST>
<DELETELIST>       ::= <PREDICATE_LIST>
<RESOURCETYPE>     ::= <LABEL_STRING>
<OUTCOMES>         ::= '(' outcomes (<OUTCOME>)* ')'
<OUTCOME>          ::= '(' <LABEL_STRING>
                         <COST_ATTRIB>
                         <QUALITY_ATTRIB>
                         <DURATION_ATTRIB> ')'
<COST_ATTRIB>      ::= '(' cost <DISTRIBUTION> ')'
<QUALITY_ATTRIB>   ::= '(' quality <DISTRIBUTION> ')'
<DURATION_ATTRIB>  ::= '(' duration <DISTRIBUTION> ')'
<DISTRIBUTION>     ::= ( <POSITIVE_INT> <0-1FLOAT> )+
<DOMAIN_ATTRIB>    ::= '(' domain <LABEL_STRING> ')'
<PRECOND>          ::= '(' preconditions <PREDICATE>* ')'
<LABEL_STRING>     ::= ([A-Z,a-z])+([' ',0-9,A-Z,a-z])*
<PREDICATE_LIST>   ::= '(' (<PREDICATE>)* ')'
<PREDICATE>        ::= '(' <LABEL_STRING> <VAR_STRING> (' '
                         <VAR_STRING>)* ')'
<VAR_STRING>       ::= '?'<LABEL_STRING>
```

Figure 5.5: DTS Planner method definition BNF.

Since methods are the terminal nodes in task networks, they provide actions that can be scheduled and run by an agent. Methods also have TÆMS performance characteristic distributions that indicate with some probability how much quality the method will produce, how long it will take to run, and how much it will cost. A short description of each element of the method BNF follows.

METHOD — the start element

METHOD_FORM — a form in a task list

ADDLIST — a predicate list of additions to a state resulting from the inclusion of the method in a plan or schedule

DELETELIST — a predicate list of deletions from a state resulting from the inclusion of a method in a plan or schedule

RESOURCETYPE — the type of information resource produced when the method is run

OUTCOMES — TÆMS outcomes

To illustrate the use of planning domain methods, Figure 5.6 contains an example of a method definition. This method is for the same simplified Tripbot domain as the above task. The form is what distinguishes one method from another when planning. The form for this method is

$(fly\ ?c1\ ?c2)$. This means that the method accepts two logical atoms from the world state. Typically, these would be bound through a task decomposition, e.g., in the above example decomposition of the $(fly\ ?c1\ ?c2)$ task has one decomposition $(fly - us\ ?c1\ ?c2\ ?attribs - us)$. The binding — a set of substitutions — for $?c1$ and $?c2$ in the decomposition is found in a matching phase of the *precondition* evaluation. That matching binding is then passed to the task reduction and used to bind the matching variables in the method form. The outcome distributions given are an example, and are not meant to indicate realistic values.

```
(spec_method
    (domain tripbot)
    (form (!fly-delta ?c1 ?c2 ?attribs))
    (preconditions)
    (addlist (in ?c2) (flight-delta-booked ?c1 ?c2))
    (deletelist (in ?c1) (goal (in ?c1)))
    (resourcetype AirFlight)
    (outcomes
        (outcome_1
            (cost 1.0 1.0)
            (quality 1.0 1.0)
            (duration 1.0 1.0)
        )
    )
)
```

Figure 5.6: DTS Planner method definition example.

In addition to the preconditions, the method definition includes the add list and delete list specifications. These are the representational requirements for a STRIPS planning domain definition [JSW98]. The add and delete lists are the specifications for transformation from one state to another in a plan search space. The space is the set of all possible world states. New world states are created by adding or deleting logical atoms from the current world state.

A method's `resourcetype` specification specifies the resources produced by this action if executed as part of an instantiated schedule. In Tripbot, the instantiated schedule is called an ActionSequence. The ActionSequence contains instantiated method types from the MTIM, references to to entries in the IRT, and fields for storing actual method characteristics, much like a TÆMS method in [Hor03]. The resource type specified is that of an AirFlight. AirFlight is part of Tripbot's ontology, which is implemented in Java.

5.2.1.3 Axioms

Axioms for DTS are an extension of the axioms used in SHOP [NCLMA99], which are a set of horn clauses with some logical control over how they are evaluated [NMAC+03]. DTS planner axioms support arbitrary computations callable through a Java reflection mechanism. Axioms are an important method for specifying constraints on bindings. Important parts of an axiom's representation are its form and its truth conditions. The BNF for axioms is given in Figure 5.7, and a short description of each element of the BNF follows.

```
<AXIOM>           ::= '(' spec_axiom
                        <DOMAIN_ATTRIB>
                        <AXIOM_FORM>
                        (<TRUTHCONDITIONS>)? ')'
<AXIOM_FORM>      ::= <METHOD_FORM> ::= '(' <LABEL_STRING>
                        <VAR_STRING> (' '
                        <VAR_STRING>)* ')'
<TRUTHCONDITIONS> ::= '(' truthconditions (<TRUTHCONDITION>)* ')'
<TRUTHCONDITOIN>  ::= '( <LABEL_STRING>
                        '(' condition <PREDICATE_LIST> ')' ')'
<DOMAIN_ATTRIB>   ::= '(' domain <LABEL_STRING> ')'
<LABEL_STRING>    ::= ([A-Z,a-z])+([`_',0-9,A-Z,a-z])*
<PREDICATE_LIST>  ::= '(' (<PREDICATE>)* ')'
<PREDICATE>       ::= '(' <LABEL_STRING> <VAR_STRING> (' '
                        <VAR_STRING>)* ')'
<VAR_STRING>      ::= '?'<LABEL_STRING>
<PRECOND>         ::= '(' preconditions <PREDICATE>* ')'
```

Figure 5.7: DTS Planner axiom definition BNF.

AXIOM — the start element

AXIOM_FORM — a form in a precondition

TRUTHCONDITIONS — a list of truth conditions that are evaluated as an $if \ldots elseif \ldots else$ structure

TRUTHCONDITION — a conjunct of predicates

The form of the axiom specifies how it is referenced from within precondition blocks. Borrowing the SHOP [NCLMA99, NMAC+03] convention, truth conditions work as an $if \ldots elseif \ldots else$ structure. The axiom is true if the first condition is true, or if the first is false and the second is true, or if the first two are false and the third is true, and so on. Figure 5.8 is given to illustrate the use of an axiom in the simplified Tripbot domain.

```
(spec_axiom
  (form (need-flight-from ?x))
  (truthconditions
     (condition_1
          (condition  (in ?x)
                   (goal (in ?y))
                   (different ?x ?y))
     )
  )
)
```

Figure 5.8: DTS Planner axiom definition example.

5.2.2 Problem Definition Language

The problem definition language is a language in which a world state and goal may be encoded. The world state is expressed in STRIPS style predicates at a given point in time. Figure 5.9 has the BNF specification for textual DTS problem definition. A brief description of the elements of the BNF follows.

```
<PROBLEM_DEF>              ::= '(' <LABEL_ATTRIB> <DOMAIN_ATTRIB>
                               <STATE_SPEC> <GOALS_SPEC> ')'
<LABEL_ATTRIB>             ::= '(' label <LABEL_STRING> ')'
<DOMAIN_ATTRIB>            ::= '(' domain <LABEL_STRING> ')'
<STATE_SPEC>               ::= '(' state <GROUND_PREDICATE_LIST ')'
<GOALS_SPEC>               ::= '(' goals '('achieve-goals '['
                               <GROUND_PREDICATE_LIST> ']' ')' ')'
<GROUND_PREDICATE_LIST>    ::= '(' (<GROUND_PREDICATE>)* ')'
<GROUND_PREDICATE>         ::= '(' <LABEL_STRING> <LABEL_STRING> (' '
                               <LABEL_STRING>)* ')'
<LABEL_STRING>             ::= ([A-Z,a-z])+([''_',0-9,A-Z,a-z])*
```

Figure 5.9: DTS Planner problem definition in Backus Naur Form.

PROBLEM_DEF — the start element

AXIOM_FORM — a form in a precondition

STATE_SPEC — an initial state, expressed as a set of true predicates about the world

GOALS_SPEC — a list of achieve goals

This concludes our discussion of the specification of all complex and atomic operators in planning on world states with task networks. Since hierarchical task network planning with variables is semi-decidable, care must be taken to ensure that methods provide a bijective mapping between sets of world states; i.e., a world state should never map to itself, otherwise a loop may result.

5.3 DTS Algorithms

In this section we will discuss the algorithms used by the DTS planner to generate task structure for the DTC scheduler. DTS planning merges of Hierarchical Task Network (HTN) planning, based principally on the SHOP planners[NCLMA99, NMAC+03], and criteria-directed search and sampling, most like the DTC scheduler [Wag00, WL01]. Switching from the PBW domain, we will discuss the details of the operation of the DTS planning algorithms mostly in the context of an example from the Tripbot domain. Tests of the DTS planner were conducted in both the Tripbot and the PBW domains.

The planning process begins with reading in the definitions of the domain and problem specifications that are created in accordance with the syntax discussed previously[3] The planning problem starts with the initial world state, S_1, and a list of goals, G_F, to complete in the world. Each world state, including the first world state, S_1, consists of a list of augmented first-order predicate calculus propositions.

Each state proposition is augmented in that it may have a special last argument, recognized as such by the problem parser, that is a list of TÆMS characteristic outcome distributions. Any atom or binding tree in the domain may thus carry a value associated with a task's characteristic accumulation functions.

This means that in addition to the performance characteristics for method types, which have a form like $(fly - delta?c1?c2?attribs)$, characteristic distributions may also be propagated from

[3]Alternatively, the planner may be given instantiated domain theory and problem definition structures.

specific method bindings, say of the form

$$(fly{-}delta\ bangor\ boston\ [(quality\ [5.0\ 0.9\ 10.0\ 0.1])(cost\ [10.0\ 0.2\ 0.0\ 0.8])\ (duration\ [30.01.0])])$$

. This capability adds flexibility to the representation for problem instances in a way to specifically leverage TÆMS criteria-directed analysis. The means of passing characteristic function information for performance-related facts in the domain also enables the DTS planner a choice between making decisions on task alternatives based on updated task performance information versus averaged or learned characteristics over a given set of method bindings.

The initial goals are formulated as achieve goals using the syntax described previously. An example from the Tripbot domain is for achieving a trip from Bangor, Maine to Orlando, Florida:

$$(achieve - goals\ [(trip\ bangor\ orlando)])$$

The problem for the planner is then to find a task network and associated TÆMS task structure that contain plans to achieve the goal from the initial world state represented in S_1 to a state that is a superset of the states represented in the achieve goals state descriptions, $G_{initial}$. Its principle means of performing the search is a depth-first task decomposition. Figure[?] depicts this search.

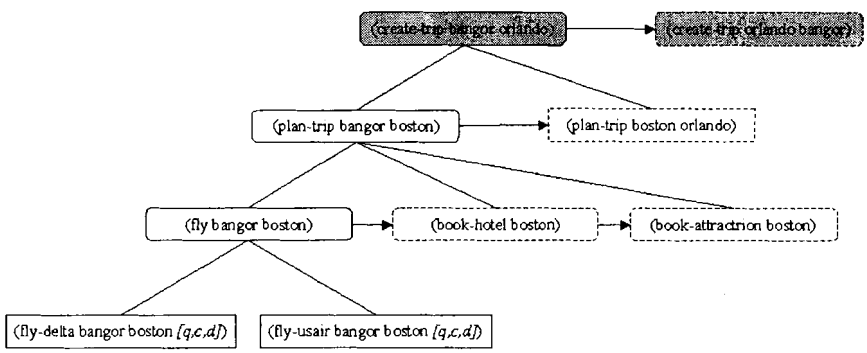

Figure 5.10: Task structure decomposition.

The search begins with the first goal. If the form of the task (achieve-goal) matches ground terms in the current world state, then the task is reduced to its decomposition. In the DTS planner, the intermediary task networks, augmented with TÆMS accumulation functions, interrelations, and other necessary TÆMS attribues, are stored as TÆMS task structures. Goals are decomposed into tasks, and tasks into more primitive tasks, and finally methods when no more task decompositions apply. The applicability of decomposition steps and the order of their application are specified in the domain theory. This simply means that, in contrast to other types of planners, including partial-order and graph-based planners, goal states are not pursued nondeterministically in theory.

5.4 Generating Task Structures

While task network planning is at the core of the DTS planner, its objective is to create TÆMS task structures. The planner is thus structured not to produce a single plan, but rather to produce a set of plans encoded in TÆMS.

Goal reduction and task decomposition in the DTS planner begin in the **GENERATE-TASK-GROUP** method. There is an intermediary plan state, PS_1, derived from the problem specification and containing the planning problem's initial state S_1, empty add and delete lists, and the problem's goals. Plan states PS_i are used to track the world state at every point in the task structure's creation to determine which decompositions can apply. A TÆMS task group node, *task_group* is also created. The plans, encoded in the subtasks of *task_group* are then generated by recursively seeking and composing partial plans in the **GENERATE-TASK-STRUCTURE** method.

```
   GENERATE-TASK-GROUP(Domain d, Problem p)
1  task_group = new TaskGroupNode(SUM);
2  GENERATE-TASK-STRUCTURE(task_group, p.goals(), p.state());
3  return task_group;
```

In **GENERATE-TASK-GROUP**, at line 1, the *SUM* arguments indicates that the node's quality attribute is governed by the *sum* QAF. When ***GENERATE-TASK-STRUCTURE*** returns, the instantiated *TaskGroupNode* is returned from GENERATE-TASK-GROUP. The *TaskGroupNode*, in Tripbot's operation, is then serialized to TTÆMS and scheduled.

GENERATE-TASK-STRUCTURE and its two supporting methods, **GENERATE-FROM-GOAL** and **GENERATE-FROM-GOAL-LIST**, contain the logic to do the depth first traversal of each applicable task decomposition structure in the domain theory.

```
    GENERATE-TASK-STRUCTURE(TaemsNode task_node, GoalList goals, State s)
1    if(task_node.compDeadline() - currentTime() < MIN_DELTA_T or goals.empty())
2      return;
3    else if(goals.first() is not a GoalList)
4      GENERATE-FROM-GOAL(task_node, goals.first(), goals.rest, s);
5    else
6      GENERATE-FROM-GOAL-LIST(task_node, goals.first(), goals.rest(), s);
```

In **GENERATE-TASK-STRUCTURE**, task_node.compDeadline() is the deadline for completion of the planning phase of computation; the planner repeatedly checks to see if it is within a Δt bound to continue planning. If it is within the bound — approximately the time required for some worst case bookkeeping — it stops planning and writes out the in-memory task structure. This structure can be scheduled if only one terminal method has been reached in its depth-first search, since the scheduler will ignore nonterminal task nodes in the serialize task structure. If a goal in the list of goals is a goal list, this is treated as an alternative point by **GENERATE-FROM-GOAL-LIST**. And, in the case where the next element to be expanded is a singular goal, **GENERATE-FROM-GOAL** is invoked.

```
    GENERATE-FROM-GOAL(TaemsNode task_node, Goal first, GoalList rest, State s)
1   if(first can be achieved by a Method)
2    method_node = new TaemsNode(SUM);
3    GENERATE-METHOD-STRUCTURE(first, method_node, s);
4    if(method_node.fail())
5     return;
6    else
7     subtask_node = new TaemsNode(SUM_ALL);
8     subtask_node.addNode(method_node);
9     GENERATE-TASK-STRUCTURE(subtask_node, rest, s);
10    if(subtask_node.fail())
11     task_node.addNode(method_node);
12     return;
13    else
14     subtask_node.addIR(ENABLES, method_node, e_one_node);
15     task_node.addNode(subtask_node);
16  else
17   eone_subtask_node = new TaemsNode(E_ONE);
18   reductions=GENERATE-GOAL-REDUCTIONS(first, s)
19   while(not(reductions.empty()))
20    foreach(r in reductions)
21     subtask_node = new TaemsNode(SUM);
22     GENERATE-TASK-STRUCTURE(subtask_node, r.goals(), s)
23     if(subtask_node.fail())
24      continue;
25     else
26      eone_subtask_node.addNode(subtask_node);
27    reductions =
28     GENERATE-GOAL-REDUCTIONS(first, s, reductions.type())
29   task_node.addNode(eone_subtask_node);
```

GENERATE-FROM-GOAL receives as arguments a task structure, a current goal to pursue, a list of the remaining goals, and a projected worldstate. In addition to logical propositions describing the projected worldstate, the State structure includes the state of a plan encoded in an exclusive branch of the task structure. The structure also contains the derivation of the plan state for use in backtracking to old plan states when an infeasible subplan branch is discovered.

At line 1 **GENERATE-TASK-STRUCTURE** checks to see if the goal can be achieved by a method in the domain theory. If it can, than a new method node (and plan state) is created with the

results of **GENERATE-METHOD-STRUCTURE**, which modifies the original TÆMS method node and plan state through the application of a method type to the current worldstate.

While we won't give pseudocode to detail **GENERATE-METHOD-STRUCTURE**, called at line 3 in **GENERATE-FROM-GOAL**, finding an applicable method involves looping through each method type available in the domain theory and determining whether each method can be applied to the primitive task form and current worldstate. If the task has no variables or has variables that are fully bound, then a simple match is performed. If there is an unbound variable or function in the form, then a more sophisticated matching operation is performed, where all applicable terms in the current world state consistent with the axioms in the domain theory are unified with the form. The match operation returns a substitution set $SS = \{SUB_1, \ldots, SUB_k\}$ for a matching method. Both the substitution set, SS, and the method go through a process or ensuring their uniqueness in the binding chain for this branch of the search space.

Each substitution returned contains a list of instantiated variable-term bindings, if there is a set of substitutions that satisfies all the preconditions for the method. If there is no such assignment of ground terms to variables, then the matching procedure returns an empty list. The operation halts and returns a failure condition which causes the search on this branch to halt.

If a reduction to method structure succeeds, the method is applied to the current worldstate and included in the generated task structure. The application of methods to worldstates is the principle means by which the depth-first search of feasible world states is conducted. The state passed into **GENERATE-METHOD-STRUCTURE** is modified in accordance with the method's add and delete specifications. If the add and delete specifications are not empty and do not cancel each other, there is a net change in the world state information. A new world state is thus created and task decomposition continues.

If a method reduction fails, a goal to task reduction is attempted. The workhorse of finding applicable task reductions for goals is the **GENERATE-GOAL-REDUCTIONS** method. This method is called if no method structure can achieve the goal with a goal form GF_i and a reference to the current state of the world. To perform the goal reduction to applicable task structures, the

planner attempts to find a substitution, S_i, that matches a goal form to the task decomposition form, TDF_i. If there is a match in the list of task decomposition forms, then the matching reductions are returned, with any variables bound according to their applicability to the task structure in the current world state. Task structures are applied to goals in a conditional structure that works as an $if \ldots elseif \ldots else$ control structure; i.e., the preconditions of the control structure are evaluated in order, until one precondition succeeds, given the current substitution and axioms. If none succeeds, an empty list is returned to **GENERATE-GOAL-REDUCTIONS** to indicate failure. When a given task decomposition precondition, PRE_i, matches a goal form GF_i with

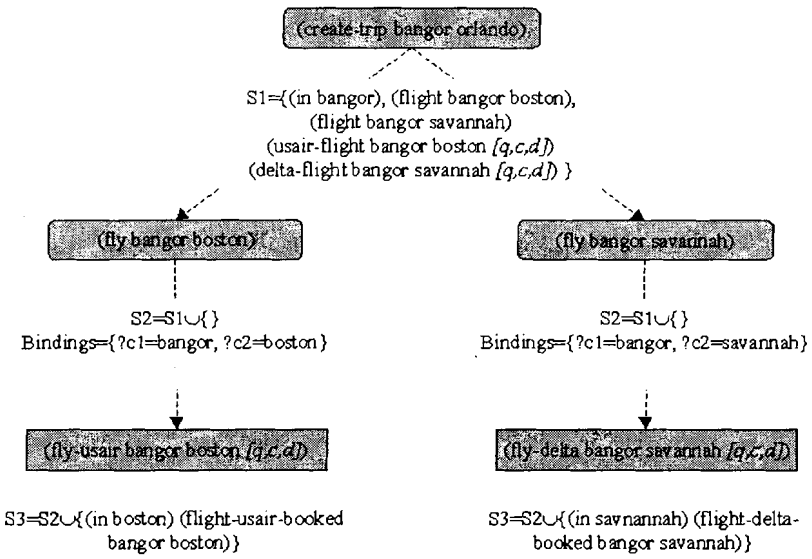

Figure 5.11: Task structure decomposition and binding data flow.

a substitution derived from the initial match between the form of the goal, GF_i, and the form of the task decomposition form, TDF_i, the precondition succeeds. There may be more than one substitution from the initial match between GF_i and TDF_i, so each substitution is applied to the task decomposition to provide a reduction for each substitution.

The matching procedure is invoked to find satisfying substitutions between one form and another, given an existing substitution set and a worldstate. The matching procedure finds a substitution list, SL_i, if one exists, from a starting form SF, a target form TF, and a state S. If

preconditions to the match exist, they are evaluated in the context of a current substitution list and worldstate.

To match a set of preconditions, each predicate of the precondition, PRE_i, including arbitrary computations invoked through Java reflection, is evaluated for a match — in the case of computations, a nonempty value indicates a match. There are several nonreflective prefix operators borrowed from SHOP [NCLMA99, NMAC+03] that can be used in a precondition predicate: arithmetic — $-$, $+$, $-$, $*$, $/$, and the equivalence — $-$, $>$,$<$, $>=$, $<=$, and $=$. Bindings of a successful precondition computation may be used in following computations. If a complete precondition computation is successful, the bindings of the computation are passed to the task or method instance for which they were a test.

If the match preconditions predicate is not a computation, then a predicate match is attempted given a state of the world and a (possibly empty) substitution set. For example, if the fly task decomposition precondition is $(flight\ ?x\ ?y)$ and a current substitution's most general unifier is $\{(?x\ bangor)\ (?y\ atlanta)\}$, then this check attempts to find a predicate in the current world state of the form $(flight\ bangor\ atlanta)$, indicating that there exists a flight from Bangor, Maine to Atlanta, Georgia. The substitution that permitted the match is then unioned with the original substitution and returned. The search for an applicable binding, given a substitution, is currently implemented as a sequential search through the worldstate. This operation has complexity $O(m)$. It is required for each substitution provided, thus yielding time complexity $\emptyset(m * n)$, where m is the number of states in S and n is the number of substitutions provided.[4]

If no ground predicate in the worldstate matches the precondition predicate and it is not a computation, then the application of one of the domain theory's axioms to PRE_i is attempted. Each of the domain theory's axiom's forms, AF_i, described above, is matched against PRE_i in succession. If AF_i matches PRE_i, then the substitution that allowed the match is unioned with the initial substitution, and a proof attempt is made at the axiom's body. The proof of the axiom's body is the same as the proof of a task or method type's preconditions.

[4] A state storage model that hashed state representations would improve the speed of this operation considerably.

Substitutions are combined in the following way. Say there are two substitutions, SUB1 and SUB2. Each substitution contains a list of variable, term pairs, e.g., $SUB_1 = \{(?y\ bangor),\ (?x\ florida)\}$ and $SUB_2 = \{(?y\ atlanta),\ (?z\ boston)\}$. To union the substitutions, a worst-case $O(n^2)$ operation is performed where the variable parts of the variable term pairs are compared, and variable term pairs of the added substitution with the same variable as a variable term pair in the original substitution are excluded. A substitution union is thus a union based on the variable part of the variable, term pair. Figure 5.11 depicts this process.

Task decomposition continues until no task's precondition unifies with the current world state. Tasks with equivalent forms, but with different decompositions and decomposition strategies for completing a task or task list, create separate reductions, each of which can expand the search space by one branch. The total search space is thus increased by the distance from the task reduction to the ground literal base of its reductions with each equivalent task reduction.

```
    GENERATE-FROM-GOAL-LIST(TaemsNode task_node, GoalList first, GoalList rest,
    State s)
1   if(first is a disjunctive alternative point)
2    goal_characteristics_set = new GoalCharacteristicSet();
2    foreach(g in goals)
3     goal_characteristics_set.add(new GoalCharacteristics(g));
4    selected = GENERATE-SELECTED(goal_characteristics_set, task_node);
5    eone_subtask_node = new TaemsNode(E_ONE);
6    foreach(selected_goal in selected)
7     subtask_node = new TaemsNode(SUM);
8     GENERATE-TASK-STRUCTURE(subtask_node, rest.add(selected_goal), s)
9     if(subtask_node.fail())
10     continue;
11    else
12     eone_subtask_node.addNode(subtask_node);
13   task_node.add(eone_subtask_node);
14  else
15   subtask_node = new TaemsNode(SUM);
16   GENERATE-TASK-STRUCTURE(subtask_node, first, s)
17   if(subtask_node.fail())
18    continue;
19   else
20    task_node.addNode(subtask_node);
```

5.5 Creating Alternative Subplans in Task Structures

Creating alternative plans and, hence, schedules creates new search branches for the planner. Each alternative must be planned for separately because, in general, each included method may have different postconditions that enable other and different tasks and methods than its alternative method. In cases where one method's post conditions are the same as another method's, it may be included without significant additional computational cost to the task structure.

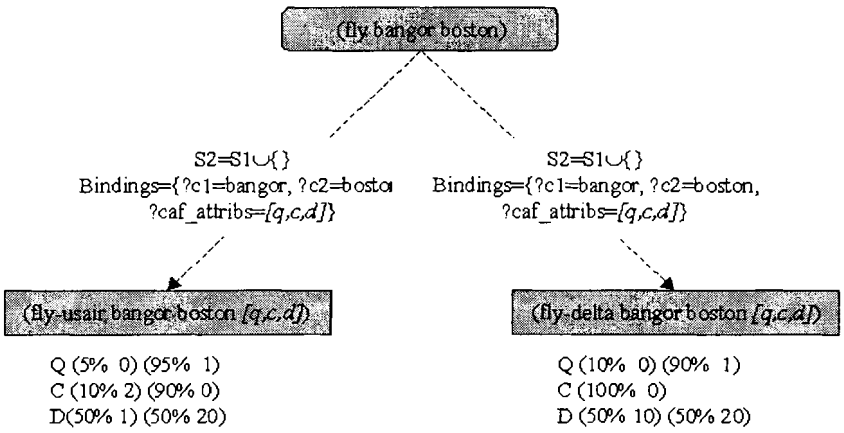

Figure 5.12: Task decomposition binding options.

When the number of applicable disjunctive methods is large, it is essential that the planner constrain its search space. It does so through the use of a heuristic, criteria-directed sort. Using a characteristic evaluation vector comparator, a total order is placed on the applicable tasks at an alternative choice point. The total order is derived from the application of TÆMS criteria rating functions [WL01].

Figure 5.13 depicts the combination of subplans and alternatives to produce a rating for each alternative. Associated with a task structure point is a set of feasible subplans and their associated performance distributions. In Figure 5.13, those subplans are depicted as the set $\{PS_1, PS_2, PS_3\}$. To obtain a raw rating, the distributions from each subplan are combined with a alternative candidate to produced a new set of performance distributions. Each alternative is then rated according to the TÆMS criteria definition relative to the other alternatives.

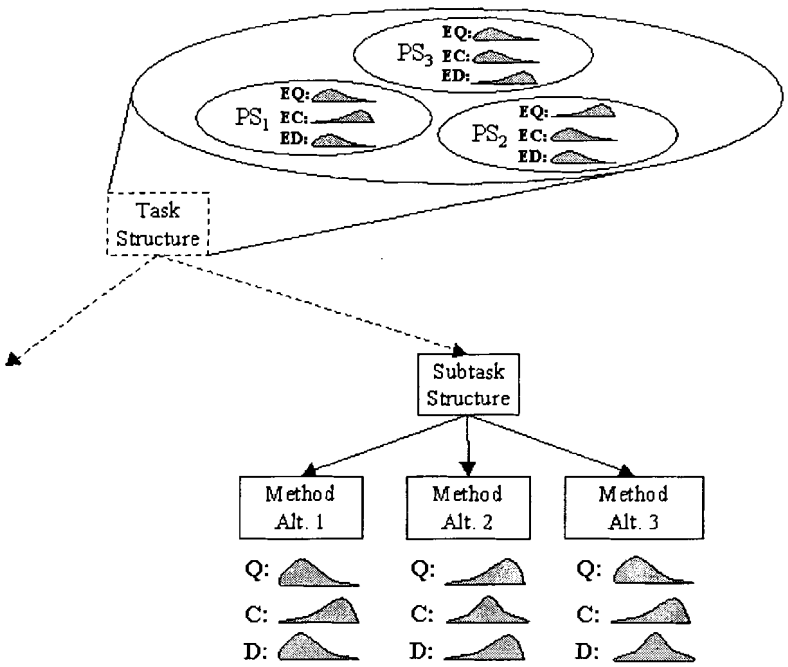

Figure 5.13: Rating subplan alternatives.

A subset of the alternatives is then chosen. The subset is determined according to an adjustable selection function. The selection function produces one set from the possible sets of n options, and 2^n possible sets. The DTS planner makes one of the following selections: *median*, *extremes*, *best*, *worst*, *random*, and *all*. Selecting from the set of alternatives enables the planner to ideally provide an optimal distribution of method alternatives based on preferred characteristics as specified in the characteristic comparison criteria and selection heuristic.

```
    GENERATE-SELECTED(GoalCharacteristicsSet set, TaemsNode task_node)
1   foreach(gc in set)
3     UPDATE-RATING(gc, task_node.allOutcomeSets());
4   SORT-ON-RATING(set);
5   SELECT-SUBSET-BY-HEURISTIC-TYPE(set, HEURISTIC_TYPE);
6   return set;
```

To clarify, there are two points at which alternative set selection comes to play a role. One is where there exists more than one method that has an applicable binding. The other is where there is more than one applicable task reduction. The planner could select a set at this point as

well, but unless it is a reduction to a method with defined characteristics there is no information available upon which to base a decision unless some preprocessing is done as is the case in the DTC scheduler[WL01]. The only available option is selecting a subset of a determined size in order to facilitate planning within a time constraint. Since our experimentation includes some exploration into sampling from alternative methods with all reductions, reduction sampling has not been implemented — all reductions are explored unless time constraints are violated.

Chapter 6

EXECUTION MONITORING

Planning and scheduling are fundamental aspects of the agent architecture. They offer the agent explicit control over its actions, which allows it a degree of flexibility unachievable without their analogue. The Design-To-Schedule task structure planning performed by the F-SRTA architecture is described in detail in a separate section. The Design-To-Criteria scheduling performed as it pertains to planning is described in the planning section. Extensive details on the scheduler can be found in [Wag00].

6.1 Expected Characteristic Rescheduling

There are cases where a schedule will fail in one manner or another. A schedule can be considered to fail if any of its methods fails in duration or non-duration characteristics. TÆMS provides for numerous interrelations between methods, so either duration or non-duration failures may not impact the schedule; e.g., if a parallel method that only $facilitates$ fails to meet a deadline, one might think that this should not cause the schedule to fail in its entirety, but this is largely dependent on how failure is measured. TÆMS criteria provide the appropriate metric for plan and schedule failure, since they are so tightly coupled to plan and schedule generation.

We now focus on duration monitoring for rescheduling. Our focus here is equivalent to evaluating schedule failure from within a TÆMS criteria evaluation context where the 100% of the evaluation weight is put on duration certainty goodness. We are experimenting with three rescheduling policies:

- Schedule Deadline Deviation Constraint (SDD),
- Method Deadline Deviation Constraint (MDD), and
- Adjusted Method Deadline Deviation Constraint (AMDD)

We will explain each of these in turn immediately following. Each of the constraint policies can have one of three consequences, however, that are now noted:

- Shuffle-down rescheduling, where each method's expected duration characteristics are incremented or decremented by some constant factor,
- Full rescheduling, where actual runtime characteristics are are used to annotate the TÆMS method descriptions, and the DTC scheduler is reinvoked on the task structure, and
- Full replanning and rescheduling, where the current state of the world is loaded from the MSM before the failed action and replanning is done from that world state followed by scheduling.

The design of experiments currently being conducted to determine the appropriate conditions for each rescheduling condition are discussed in a later section.

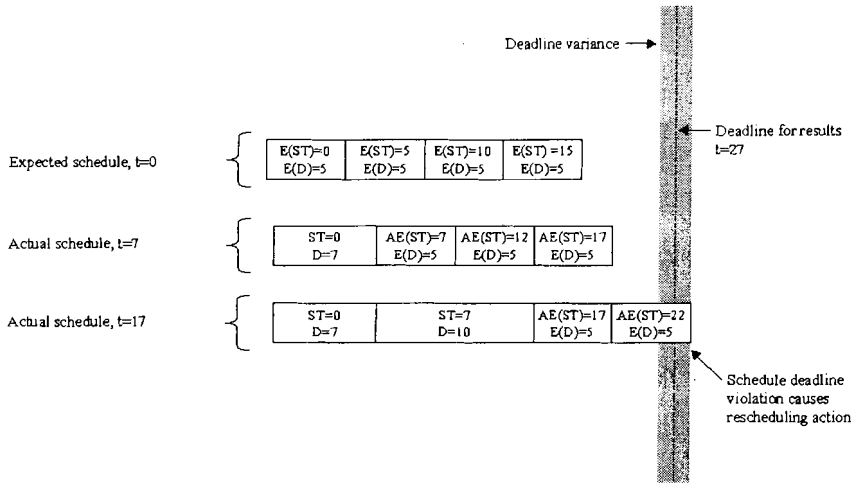

Figure 6.1: An illustration of the SDD constraint.

6.2 Schedule Deadline Deviation Constraint

The SDD policy calculates the combination of method deadline deviations for a given schedule. Each method's duration is characterized by a random variable X with an associated discrete probability distribution. The deviation of X, Dev(X), is calculated as follows:

$$Var(X) = \sum (x_i - E(X))^2 * P(x_i) \tag{6.1}$$

$$Dev(X) = \sqrt{Var(X)} \tag{6.2}$$

That is, as Equation 6.2 shows, that the deviation is the square-root of the sum of the differences between each value of the discreet probability distribution that X can assume, squared, and then multiplied by the probability of assuming that value. It obtains the full schedule deadline from the DTC scheduler and then combines the deviations of the methods included in the schedule to produce the deadline deviation – deviation from which will cause a rescheduling or replanning operation. This policy is depicted in Figure 6.1. The other constraints are variations on this constraint.

6.3 Method Deadline Deviation Constraint

The MDD policy calculates the deviation on each method included in a schedule produced by the DTC scheduler. If any method's actual duration lies outside the duration's deviation, it will cause a rescheduling or replanning operation. This policy is depicted in Figure 6.2.

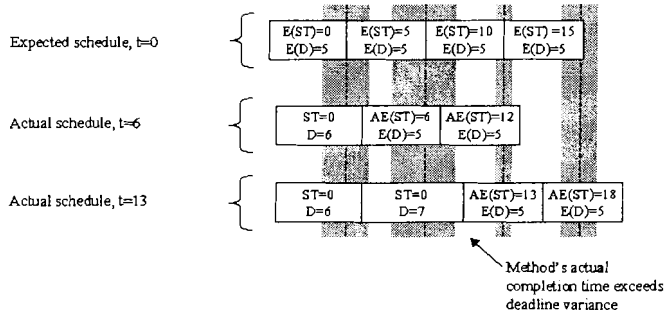

Figure 6.2: An illustration of the MDD constraint.

6.4 Adjusted Method Deadline Deviation Constraint

The AMDD policy calculates the deviation on each method included in a schedule produced by the DTC scheduler. If any method's actual duration lies outside the duration's deviation, it will cause a rescheduling or replanning operation. However, in the adjusted version of the MDD policy, actual durations can immediately cause a simple "shuffle down" rescheduling operation or a more complex operation, depending on the expected performance of the remainder of the schedule. This policy is depicted in Figure 6.3.

6.5 Monitoring Parallel Schedules with Deviance-Based Constraints

There seem to be two fundamental approaches to monitoring schedules with parallel actions. One approach is appropriate when what matters is the performance of the parallel actions in the

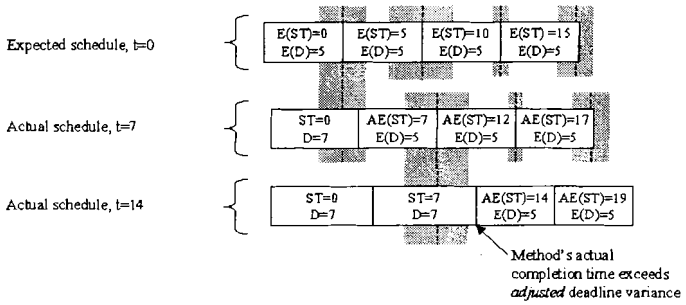

Figure 6.3: An illustration of the AMDD constraint.

aggregate, i.e., each method's deviation is important to the overall performance of the application. Another approach is appropriate when the performance of parallel actions effect the application's overall performance with different weights. In this case, each branch of the parallelized schedule could be targeted for monitoring, replanning, and rescheduling actions when a constraint was violated. Each approach may have its merits in different domains.

Chapter 7

EXPERIMENTS

Several experiments were conducted to derive statistics explaining the effect of criteria-directed goal reduction heuristic choice on planning time and schedule quality. Our ultimate research goal is to be able to appropriately characterize an agent's task environment in terms that allow us to choose heuristics effectively and efficiently. In the presented research, we are especially interested in time and complexity constraints. So, in the context of viewing the agent control problem as a problem of compute time allocation, the experiments attempt to find a map between task and environment characteristics and task structure generation heuristics.

The key features of the solution design space that are explored in the experiments are:

- The effects of the use of a goal reduction policy on scheduling options and compute time,
- The appropriate schedule failure metric based on method performance statistics, and
- The correct compute time allocation for a given task environment.

7.1 Comments on Domain Complexity

A measure of expected required computation time and space of each operation required to produce a result is necessary to make the correct decision of how to balance the allocation of time and space for computing in each situation. We need to know the expected quality of additional compute time spent on one operation versus another. We focus solely on the operations of the DTS

planner and the DTC scheduler in the experiments reported. Further, we focus on compute time, rather than the compute space, since the operations in the cases reported are run sequentially. Such a focus simplifies the analysis.

The important gauges of complexity in the DTS planning domain are:

- The number of choices in each subplan,
- The number of supporting subplans, and
- The expected number and characteristics of failing plans.

The DTS task structure planner can instantiate methods under a given task to support subplan choice. It can thus create tasks that include a number of alternative supporting subtasks. To support a number of different *exactly_one* branches based on changes in the agent's perception of plan state, there is an interplay between the domain definition and the problem definition. There has to be a number of equivalent subplans to pursue, where there are enabling interactions between one action and a sequence of successive actions.

7.2 Alternative Selection Heuristic Effects

We need to determine the effect of the choice of a reduction heuristic across sets of:

- Task environment classes,
- Domain complexities,
- Task structure generation and scheduling criteria, and
- Time to compute.d

The number and kind of varying parameters in experiments and analysis of tradeoffs between TÆMS planning and scheduling are quite daunting; they include:

- Task structure,
- Method performance characteristics,
- Performance characteristic model types,
- Selection heuristic dimension,
- Planning and scheduling criteria and application,
- Compute time constraints,
- Domain complexity, and
- Problem complexity.

Both the planning and scheduling compute time are of interest in these cases, since what is ultimately of interest is the expected utility as a function of compute time for a task structure and its schedules which may incorporate contingencies, where the probability of failure in a given schedule varies. However, we do not focus on that element of the problem presently, partly since the the planner version used in the experiments supports anytime computation naively and the scheduler version used in the experiments does not support anytime computation.

7.2.1 Selection Heuristic Effect on DTS Planner Runtime

The effects of a given alternative selection heuristic on planning runtime is dependent on several aspects of the domain theory. One is, b, the branching factor for the the number of alternatives that are available, on average, and explored per selection. Another is how many alternative points are part of the search, on average, to complete a task network, n, the depth of a complete search. This yields time complexity of $\Theta(b^n)$ for a complete search of the plan space, if the problem is decidable.

The DTS planner uses alternative rating and selection heuristics at each alternative point to choose a subset of applicable alternatives to include in the task network and associated TÆMS task structure. We now examine the runtime performance of the different selection heuristic types on planning runtime. We compare the planning runtime with the runtime required to schedule the generated TÆMS task structure.

The experiments were run in the Probabilistic Blocks World (PBW) domain, with three applicable method alternatives at each alternative point – the ability to stack a block in three different ways, with differing cost and duration profiles, but with uniform quality profiles.

Figure 7.1 shows run times for the DTS planner when using a heuristic that explores one alternative at alternative selection points and the DTC scheduling runtime on the resulting TÆMS task structure for domains of increasing complexity. In the "select one" heuristics, one alternative of

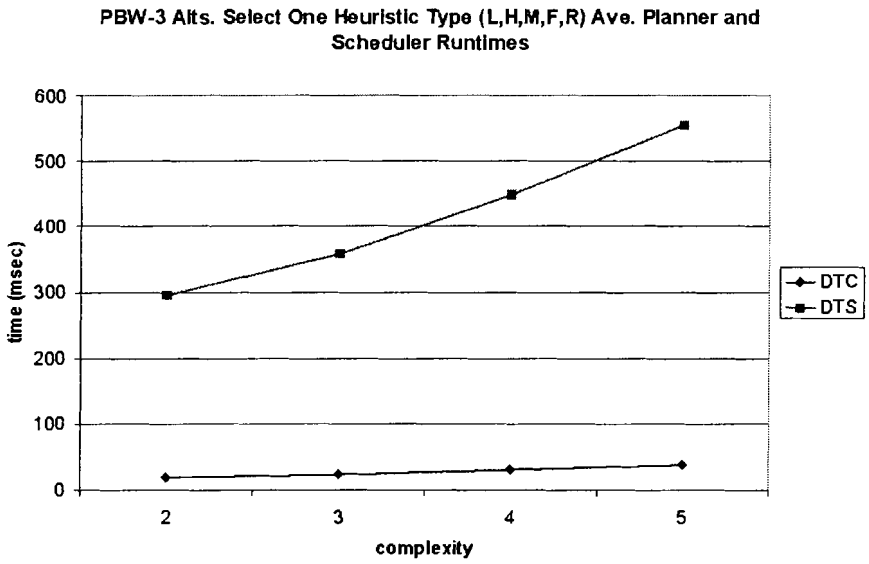

Figure 7.1: DTS planner and DTC scheduler runtimes for select one heuristics over increasing domain complexity.

the three applicable alternatives available is selected. There are five heuristics which select only one of the alternatives. First, recall that the alternatives are ranked according to a TÆMS criteria rating that calculates each alternative's rating based on the relative and proportional value of its characteristics across existing plans and the other alternatives.

The select one heuristics are, more precisely, as follows:

high — choose the highest rated alternative;

low — choose the lowest rated alternative;

median — choose the median alternative: if A is the list of alternatives, choose $\lfloor \frac{|A|}{2} \rfloor$;

random — choose a random alternative: if A is the list of alternatives and R is a random variable with a uniform distribution over the set $[0, 1)$, choose $\lfloor R * |A| \rfloor$.

fast — choose the fastest alternative.

The select one heuristics, minus the selection processes described above, perform the same operations, and so have effectively identical runtimes.

The average runtime curves for both the DTS planner and the DTC scheduler are given in Figure 7.1 over increasing domain complexity, meaning that there were more blocks to move from one configuration to its inverse configuration — the sort of problems used for these tests. Figure 7.1 shows clearly that the total computing runtime to produce a schedule of action from a goal is dominated by the planning process when only one alternative is included at each alternative point in task network generation and, hence, in the resulting TÆMS task structure. This shows that the DTC scheduler handles a task structure with sparse alternatives well. This is to be expected.

Figure 7.2 shows run times for the DTS planner when using a heuristic that explores two alternatives at alternative selection points and the DTC scheduling runtime on the resulting TÆMS task structure for domains of increasing complexity. In the "select two" heuristics, two alternatives of the three applicable alternatives available are selected. There is only one heuristic that selects two alternatives, the *extremes* selection heuristic. The *extremes* heuristic combines both the *high* and *low* selection heuristics to explore both the highest and lowest rated alternatives.

We see now that the runtime of the DTC scheduler is beginning to overtake the runtime of the DTS planner at domain complexity between 4 and 5 blocks. This is not due to inherent complexity in this instance of the scheduling problem but, rather, in the generalized manner in which the DTC

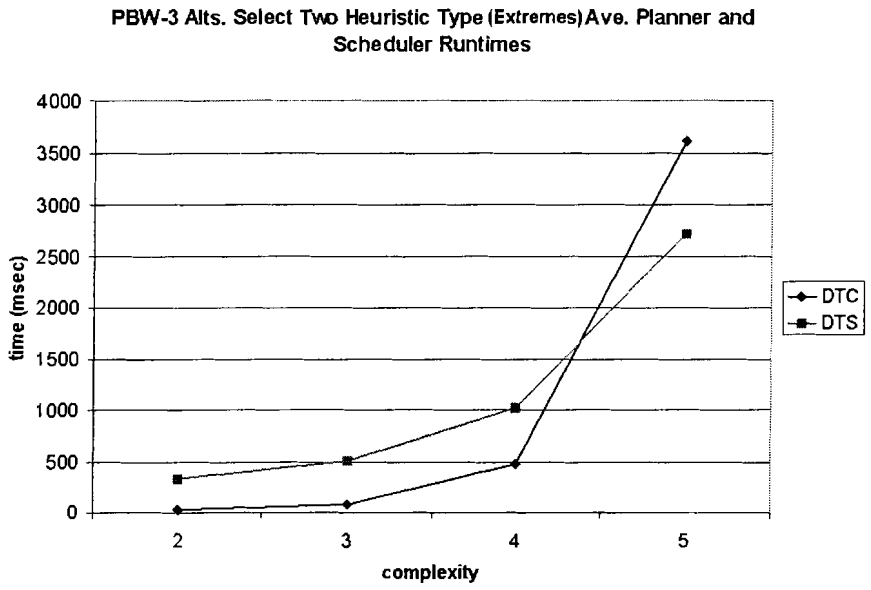

Figure 7.2: DTS planner and DTC scheduler runtimes for select two heuristics over increasing domain complexity.

scheduler schedules. Thus, current research is being conducted to match appropriate scheduling algorithms to task structure scheduling problems such as the one we have here.

Figure 7.3 shows run times for the DTS planner when using a heuristic that explores all alternatives at alternative selection points and the DTC scheduling runtime on the resulting TÆMS task structure for domains of increasing complexity. Here, "all" equals three alternatives.

Clearly, there is significant computational overhead with the exploration of plan and schedule alternatives — exponential in space, in the worst case. In the *all* alternative selection case above, we see the scheduling time dramatically overtake the planning time.

Fortunately, as we will see shortly, it turns out that, given a uniform distribution of method characteristics and deadlines, extra computation will not typically be needed unless deadlines are very tight and meeting them precisely is important for the domain problem, if schedule quality density is our measure, i.e., $\frac{Q(t)}{t}$.

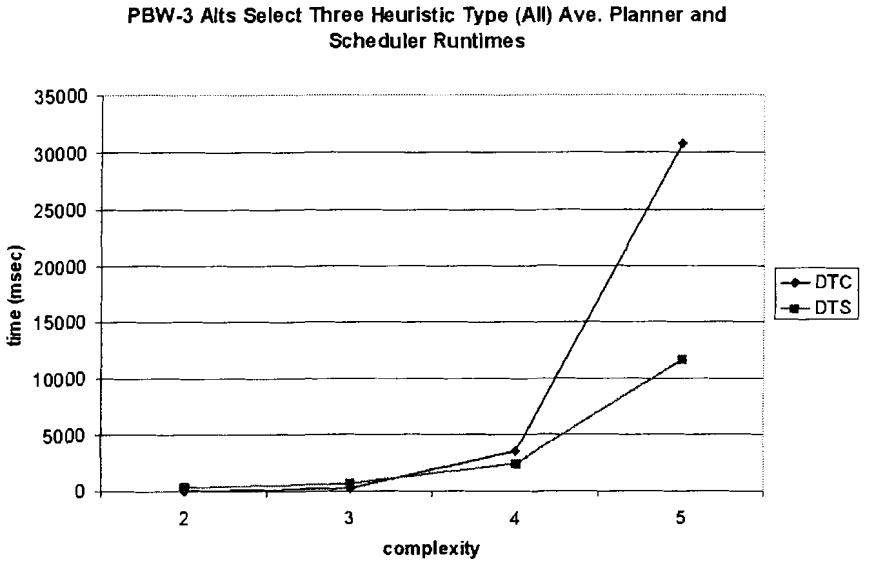

Figure 7.3: DTS planner and DTC scheduler runtimes for select three heuristics over increasing domain complexity.

7.2.2 Selection Heuristic Effect on Schedule Quality

We decided to run an experiment to test whether and when expending additional computational resources will lead to benefits in terms of schedule quality density. We ran each heuristic on a set of random domain planning problems. The results are displayed in Table 7.1.

Batch	Low	High	Extremes	Median	All	Random	Fast
0	0	0	0	0	0	0	0
1	0	0.7500	0.7500	0.5316	0.8418	0.5867	0.6867
2	0	1	1	0	1	0	1
3	0	1	1	0	1	1	1
4	0	1	1	0.4316	1	0.4316	0.8418
5	0	1	1	0	1	0	1
6	0	0	0	0	0	0	0
7	0	1	1	0.3345	1	0.6689	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	1
10	0	0	0	0	1	1	0.5847
11	0	0.5000	0.5000	0.5000	0.5000	0.2475	0.5000
12	0	1	1	0.1263	1	0.5000	1
13	0	0.8333	0.8333	0.6867	0.8418	0	0.6867
14	0	0.8756	0.8756	0.5042	0.8756	0.5083	0.8798
15	0	0.8756	0.8756	0.7554	0.8756	0	0.6418
16	0	0.8756	0.8756	0.3997	0.7620	0.2753	0.7620
17	0	1	1	0.5453	1	0.4729	0.7596
18	0	1	1	0	1	0	1
19	0	1	1	0	1	0	1
20	0	1	1	0	1	0	1
21	0	0	0	0	1	0	1
22	0	1	1	0.4752	1	0.5954	0.7620
23	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0
25	0	0	0	0	1	1	0.7706
26	0	0	0	0	0	0	0
27	0	1	1	0.4763	1	0.4252	0.7620
28	0	0	0	0	1	0	0.5459
29	0	0.8756	0.8756	0	0.7620	0	0.7620
Sums	0	17.5859	17.5859	5.7668	22.4589	7.7117	19.9458
Means	0	0.5862	0.5862	0.1922	0.7486	0.2571	0.6649

Table 7.1: Schedule Quality Density Achieved By Plan Alternative Selection Heuristics

In this experiment, each of the seven selection heuristics was run on a set of 30 randomly generated Tripbot domain planning problems. The problems involve generating plan alternatives for a trip from one location to another with no stops between the two. Each location has five flights to it and five flights from it. Each flight is randomly assigned method characteristics from a uniformly distributed set of nine method characteristic types that vary stepwise along the axes of Quality, Cost, and Duration. If the first letter represents the expected value of the Quality characteristic, the second the Cost characteristic, and the third the Duration characteristic, the set is $\{HLH, HLM, HLL, MLH, MLM, MLL, LLH, LLM, LLL\}$. H represents a high expected value, L represents a low expected value, and M represents an expected value halfway between L and H. The cost of each action is the same, L. This simplification just allows us to focus on a two value tradeoff; the results for this analysis extend to n characteristic tradeoffs.

The values in Table 7.1, pertaining to schedule quality density, indicate a relative rating $[0, 1]$ of schedule quality density, so a 1 value indicates that it obtained the highest rating. A 0 value in the table indicates that no feasible schedule was found. The fact that the *low* heuristic has all zeros means that there was always a way to overshoot the deadline placed on the trip schedule.

An ANOVA test shows that there are statistically significant differences between the selection heuristics with $F_{critical} = 2.143451638$, $F = 18.34052918$, and $p = 5.49E - 17$. Pairwise T-tests show which heuristics differ significantly.

The results of pairwise t-Tests, displayed in Table 7.2, show that there is no statistically significant difference between *high*, *fast*, *extremes*, and *all* selection heuristics a the p=0.05 rejection level when the relative schedule quality density is our metric. In practice, the *high*, *extremes*, and *all* heuristics will produce higher quality schedules overall, but with higher duration and cost.

Comparison	T-test p
Low VS High	1.36703E-07
Low VS Extremes	1.36703E-07
Low VS Median	0.000334274
Low VS All	2.98376E-11
Low VS Random	0.000294858
Low VS Fast	9.19267E-11
High VS Extremes	1
High VS Median	0.000194064
High VS All	0.150658595
High VS Random	0.00289412
High VS Fast	0.471150131
Extremes VS Median	0.000194064
Extremes VS All	0.150658595
Extremes VS Random	0.00289412
Extremes VS Fast	0.471150131
Median VS All	4.51598E-08
Median VS Random	0.411768114
Median VS Fast	0.471150131
All VS Random	3.48128E-06
All VS Fast	0.471150131
Random VS Fast	4.21245E-05

Table 7.2: Pairwise T-Test p values for Heuristic Selection Quality Density Comparisons

Chapter 8

ONGOING RESEARCH

This section discusses some ongoing research in F-SRTA and Tripbot control. The focus of our work is on determining the best cases for replanning versus those for rescheduling only. In both cases there are a range of actions that can be taken to avoid failure in the future, and we focus on tuning the planning and scheduling search criteria to provide the best control solution based on the expected runtime characteristics of the available actions.

8.1 Determining and Responding to Schedule Failure

TÆMS schedule failure can occur on any of 2^n expected attribute subsets. When a failure occurs, a determination must be made to pursue a course of action to rectify the failure. The following options are available to the agent:

Retry — retry the method,

Reschedule — generate a new schedule,

Replan — generate a new task structure.

There is a special failure case in the dimension of time. A determination of failure is necessary when no further evidence of failure than lack of results within an expected time is available. The expected values of the the characteristics for each method are provided in the output of the scheduler. However, additional information, especially about the expected variance on the duration of

the method, could be helpful in this case, to determine whether one of the above recourses is appropriate, or whether one additional recourse is appropriate:

Wait — wait to see if the action completes.

When considering when to reschedule, the computation should not be taken lightly, since the determination of whether a feasible schedule still exists after a failure is a new scheduling problem, which is, in the general case, computationally hard This means that it is no small proposition under certain circumstances to make a rescheduling decision. The same can be said for a decision about generating a new task structure.

Things are further complicated by considering parallel actions, as is the case in the Tripbot domain. In cases where task alternatives are grouped under a *sum* QAF, and there are no NLEs precluding it, the tasks may be run in parallel. We now add the problem of any of the 2^n task subsets failing in any of their 2^n expected characteristic values.

8.2 Recovery Compute Time Reduction

An obvious focus point for this research is on how failure recovery compute time can be minimized within stated utility bounds. We are exploring classification of the type of agent task environments that favor more time spent planning up front rather than in reaction to failed actions, according to the best of several alternative failure criteria previously identified.

The current experiment determines if or when it is better to plan for a set of contingencies and then to schedule those contingencies or whether it is more efficient to regenerate task structures and then to reschedule. In the situation where this consideration must be made, there are two types of deferrals:

- One is to defer analysis of densely interconnected task structures to the scheduler;
- Another is to defer selection of combinations of method disjunctions that are not interconnected.

There is, of course, a notion of providing a "selection" for the scheduler in these deferrals, making them only partial deferrals of computation since the planner must do some of the computation done by the scheduler to produce the appropriate subsets. Key questions are:

- What is the complexity of the operation deferred?
- What is the complexity of the sampling operation?

We hypothesize that the following will be important factors in the test:

- The number of choices in each subtask,
- The number of supporting subtask, and
- The average number of failing methods.

A domain problem generator creates "synonymous" methods under a given task to support subplan choice. It then creates tasks that includes the number of supporting subplans. To support a number of different $exactly\_one$ branches based on changes in the perceived state of a plan, there is an interrelationship between the domain definition and the problem definition. There needs to be a number of equivalent subtasks to pursue, where there are enabling interactions between one action and a sequence of successive actions.

Chapter 9

CONCLUSION

This work has produced a number of interesting artifacts, including a new agent architecture and system with an integrated probabilistic TÆMS task structure planning component. As stated previously, the major contributions were the development of the DTS TÆMS task structure planner, the DTC scheduler driver and parser for use in the Tripbot/F-SRTA system, the Tripbot/F-SRTA system Executor, an experimental harness for generating problems and analyzing results to test hypotheses regarding the interactions between Tripbot/F-SRTA system components, runtime complexity versus utility results, several rescheduling criteria, some computational complexity identifications regarding the Tripbot/F-SRTA generic control problem, and other supporting contributions to the F-SRTA/Tripbot system. This research made the importance of accurate problem-dependent performance characteristics of related problem solving methods starkly apparent. Current and proposed research attempts to refine and generalize the solutions to problem solving method selection and online agent-based control within the TÆMS agent framework.

BIBLIOGRAPHY

[ACHK03] Yigal Arens, Chin Chee, Chun-Nan Hsu, and Craig Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Cooperative Information Systems*, pages 127–158, 1993 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/arens93retrieving.html.

[AS98] Martin Andersson and Tuomas Sandholm. Leveled commitment contracts with myopic and strategic agents. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 615–640, 1998.

[AWS03] Corin Anderson, Daniel Weld, and David Smith. Conditional effects in graphplan. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, 1998 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/anderson98conditional.html.

[BDS94] John Bresina, Mark Drummond, and Keith Swanson. Managing action duration uncertainty with just-in-case scheduling. pages 19–26, 1994.

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann, 1995.

[BH03] Sanjoy Baruah and Mary Hickey. Competitive on-line scheduling of imprecise computations. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, Volume 1: Software Technology and Architecture*, 1996 [cited August 20, 2003]. Available from: http://www.cs.unc.edu/ baruah/Papers/baruahHickey1996.pdf.

[BKC+03] Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. Theaterloc: A case study in building an information integration application. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence Workshop on Intelligent Information Integration*, 1999 [cited August 20, 2003]. Available from: http://www.isi.edu/info-agents/papers/barish99-iii.pdf.

[BKS+03] Fahiem Bacchus, Henry Kautz, David Smith, Derek Long, Hector Geffner, and Jana Koehler (organizers). Fifth international conference of AI planning and scheduleing competition. Available from: http://www.cs.toronto.edu/aips2000/, 2000 [cited August 20, 2003].

[BL99] Avrim Blum and John Langford. Probabilistic planning in the graphplan framework. In *Fifth European Conference on Planning*, pages 319–332. Springer-Verlag, 1999.

[BM98] Maroua Bouzid and Abdel-Illah Mouaddib. Cooperative uncertain temporal reasoning for distributed transportation scheduling. In *Proceedings of the Third International Conference on Multiagent Systems*, pages 397–398. IEEE Press, 1998.

[BN03] Jiri Baum and Ann E. Nicholson. Dynamic non-uniform abstractions for approximate planning in large structured stochastic domains. In *Pacific Rim International Conference on Artificial Intelligence*, pages 587–598, 1998 [cited August 20, 2003. Available from: http://citeseer.nj.nec.com/225314.html.

[Bro91] Rodney Brooks. Intelligence without representation. In *Artificial Intelligence*, number 47, page 139159. Elsevier, 1991.

[BS03] Richard C. Bodner and Fei Song. Knowledge-based approaches to query expansion in information retrieval. In *Canadian Conference on Artificial Intelligence*, pages 146–158, 1996 [cited August 20, 2003. Available from: http://citeseer.nj.nec.com/bodner96knowledgebased.html.

[CCPS99] Paul Cohen, Vinay Chaudhri, Adam Pease, and Robert Schrag. Does prior knowledge facilitate the development of knowledge-based systems? In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 221–226. AAAI Press, 1999.

[CLL03] Norman Carver, Victor Lesser, and Qiegang Long. Distributed sensor interpretation: Modeling agent interpretations in dresun. In *UMass Technical Report, UMCS 93-75*, 1993 [cited August 20, 2003]. Available from: http://mas.cs.umass.edu/pub/paper_detail.php/164.

[CRK+03] Steve Chien, Gregg Rabideau, Russell Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, Tara Estlin, Ben Smith, Forest Fisher, Tony Barrett, Gary Stebbins, and Daniel Tran. Aspen - automated planning and scheduling for space mission operations, 2000 [cited August 28, 2003]. http://citeseer.nj.nec.com/373541.html.

[CS03a] W. Conen and Tuomas Sandholm. Differential-revelation vcg mechanisms for combinatorial auctions. 2002 [cited August 20, 2003. Available from: http://citeseer.nj.nec.com/conen02differentialrevelation.html.

[CS03b] Vincent Conitzer and Tuomas Sandholm. Complexity of mechanism design. 2002 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/534262.html.

[CT91] Ken Currie and Austin Tate. O-plan: the open planning architecture. In *Artificial Intelligence*, pages 49–86. AAAI Press, 1991.

[Dec95] Keith Decker. *Environment Centered Analysis and Design of Coordination Mechanisms*. PhD thesis, University of Massachusetts, Amherst, 1995.

[DEW97] Robert Doorenbos, Oren Etzioni, and Daniel Weld. A scalable comparison-shopping agent for the world wide web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48. ACM Press, 1997.

[DHW94] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 31–36. AAAI Press, 1994.

[DK01] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

[DK02] Minh Do and Subbarao Kambhampati. Planning graph-based heuristics for cost-sensitive temporal planning. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling*, pages 3–12. AAAI Press, 2002.

[DK03] Minh Do and Subbarao Kambhampati. Sapa: A scalable multi-objective heuristic metric temporal planner. 2003 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/455880.html.

[DL00] Keith Decker and Jinjiang Li. Coordinating mutually exclusive resources using gpgp. *Autonomous Agents and Multi-Agent Systems*, 3(2):133–157, 2000.

[DWS96a] Keith Decker, Mark Williamson, and Katia Sycara. Intelligent adaptive information agents. In *Proceedings of the National Conference on Artificial Intelligence Workshop on Intelligent Adaptive Agents*. AAAI Press, 1996. Tech Report WS-96-04.

[DWS96b] Keith Decker, Mark Williamson, and Katia Sycara. Modeling information agents: Advertisements, organizational roles, and dynamic behavior. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence Workshop on Agent Modeling*, 1996.

[EHN94a] Kutluhan Erol, James Hendler, and Dana Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 249–254. AAAI Press, 1994.

[EHN94b] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1123–1128. AAAI Press, 1994.

[EHN94c] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. Technical Report 94-44, University of Maryland, 1994.

[EHNT95] Kutluhan Erol, James Hendler, Dana Nau, and Reiko Tsuneto. A critical look at critics in HTN planning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1592–1598. Morgan Kauffmann, 1995.

[EM96] Tara Estlin and Raymond Mooney. Hybrid learning of search control for partial-order planning. In *New Directions in AI Planning*, pages 129–140. IOS Press, 1996.

[EMW97] Michael Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1177, 1997.

[EW94] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *Communications of the ACM*, 37(7):72–76, 1994.

[EWD+92] Orin Etzioni, Daniel Weld, Denise Draper, Neil Lesh, and Mark Williamson. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 115–113. Morgan Kaufmann, 1992.

[FN] Richard Fikes and Nils Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence*, volume 2.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard? In *IEEE Software*, volume 12, pages 17–26. IEEE Press, 1995.

[GD01] Piotr Gmytrasiewicz and Edmund Durfee. Rational communication in multi-agent environments. *Autonomous Agents and Multi-Agent Systems*, 4(3):233–272, September 2001.

[GO01] Carlos Guestrin and Dirk Ormoneit. Robust combination of local controllers. In *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 178–185. Morgan Kaufmann, 2001.

[GS95] David Garlen and Mary Shaw. An introduction to software architecture. In *IEEE Software*, volume 12, pages 17–26. IEEE Press, 1995.

[GTM99] Robert J. Glushko, Jay M. Tenenbaum, and Bart Meltzer. An xml framework for agent-based e-commerce. *Communications of the ACM*, 42(3):106–114, 1999.

[HLVW03] Bryan Horling, Victor Lesser, Regis Vincent, and Thomas Wagner. The soft real-time agent control architecture. Technical Report TR02-14, University of Massachusetts at Amherst, 2002 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/horling02soft.html.

[Hor03] Bryan Horling. A reusable component architecture for agent construction. In *University of Massachusetts/Amherst CMPSCI Technical Report 1998-49*, 1998 [cited August 20, 2003]. Available from: http://mas.cs.umass.edu/pub/paperdetail.php?id=118.

[HT98] Frank Harmelen and Annette Teije. Characterising approximate problem-solving by partial pre- and postconditions. In *Proceedings of ECAI'98*, pages 78–82, Brighton, August 1998.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[HVM+01] Bryan Horling, Regis Vincent, Roger Mailler, Jiaying Shen, Raphen Becker, Kyle Rawlins, and Victor Lesser. Distributed sensor network for real time tracking. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 417–424. ACM Press, 2001.

[INMAA02] Okhatay Ilghami, Dana Nau, Hector Muoz-Avila, and David Aha. Camel: Learning method preconditions for HTN planning. In *Proceedings of Sixth International Conference on Artificial Intelligence Planning and Scheduling*. AAAI Press, 2002.

[JP94] David Joslin and Martha Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1004–1009. AAAI Press, 1994.

[JSW98] Nick Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[Kam95] Subbarao Kambhampati. A comparative analysis of partial order planning and task reduction planning. In *ACM SIGART Bulletin*, 1995.

[KHW95] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.

[KKY95] Subbarao Kambhampati, Craig Knoblock, and Qiang Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76(1-2):167–238, 1995.

[LAH+03] Victor Lesser, Michael Atighetchi, Bryan Horling, Brett Benyo, Anita Raja, Regis Vincent, Thomas Wagner, Ping Xuan, and Shelley XQ. A multi-agent system for intelligent environment control. In *Proceedings of the Third International Conference on Autonomous Agents*, 1999 [cited August 20, 2003]. Available from: http://dis.cs.umass.edu/pub/paperdetail.php/120.

[LDV99] Henry Lieberman, Neil Van Dyke, and Adrian Vivacqua. Let's browse: a collaborative web browsing agent. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 65–68. ACM Press, 1999.

[LFS+03] Derek Long, Maria Fox, David Smith, Drew McDermott, Fahiem Bacchus, and Hector Geffner (organizers). Sixth international conference of AI planning and scheduleing competition. Available from: http://www.dur.ac.uk/d.p.long/competition.html, 2002 [cited August 20, 2003].

[LHK+00] Victor Lesser, Bryan Horling, Frank Klassner, Anita Raja, Thomas Wagner, and Shelley Zhang. Big: An agent for resource-bounded information gathering and decision making. In *Artificial Intelligence Journal, Special Issue on Internet Information Agents*, volume 118, pages 197–244. Elsevier, 2000.

[LSK95] Alon Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems - Special Issue on Networked Information Discovery and Retrieval*, 5(2):121–143, 1995.

[MBF+98] George Miller, Richard Beckwith, Christiane Fellbaum, Derie Gross, and Katherine Miller. Wordnet: An on-line lexical database. MIT Press, 1998.

[McD03] D. McDermott. A reactive plan language. Technical Report CSD-RR-864, 1991 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/mcdermott93reactive.html.

[MF99] Zbigniew Michalewicz and David Fogel. How to solve it: Modern heuristics. Springer Verlag, 1999.

[ML98] Stephen Majercik and Michael Littman. Maxplan: A new approach to probabilistic planning. In *Artificial Intelligence Planning Systems*, pages 86–93. AAAI Press, 1998.

[MR91] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 634–639. AAAI Press/MIT Press, 1991.

[MTT03] Rila Mandala, Takenobu Tokunaga, and Hozumi Tanaka. Complementing wordnet with roget's and corpus-based thesauri for information retrieval. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, 1999 [cited August 20, 2003].

[NCLMA99] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–973. AAAI Press, 1999.

[NK01] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 459–466. Morgan Kaufmann, 2001.

[NMAC+03] Dana Nau, Hector Muoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001 [cited August 20, 2003. Available from: http://www.cs.umd.edu/

[PBC+03] Barney Pell, Douglas Bernard, Steve Chien, Erann Gat, Nicola Muscettola, Pandurang Nayak, Michael Wagner, and Brian Williams. A remote agent prototype for spacecraft autonomy. In *Proceedings of the SPIE Conference on Optical Science, Engineering and Instrumentation*, 1996 [cited August 20, 2003].

[PC96] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[PW92] Scott Penberthy and Daniel Weld. Ucpop: A sound, complete, partial order planner for adl. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 103–114. Morgan Kaufmann, San Mateo, California, 1992.

[Qia02] Yuhui Qian. A csp approach to information fusion. Master's thesis, University of Maine, 2002.

[RM01] Khashayar Rohanimanesh and Sridhar Mahadevan. Decision-theoretic planning with concurrent temporally extended actions. In *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 472–479, San Francisco, CA, 2001. Morgan Kaufmann.

[RN91] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, New Jersey, 1991.

[RS03] John Regehr and John Stankovic. Hls: A framework for composing soft real-time schedulers, 2001 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/regehr01hls.html.

[Sac75] Earl Sacerdoti. The nonlinear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 206–214. Morgan Kaufmann, 1975.

[Sku97] Martin Skutella. Approximation algorithms for the discrete time-cost tradeoff problem. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–508. ACM Press, 1997.

[SL01] Tuomas Sandholm and Victor Lesser. Leveled commitment contracts and strategic breach. In *Games and Economic Behavior*, volume 35, pages 212–270. AAAI Press, 2001.

[STH97] Kilian Stoffel, Merwyn G. Taylor, and James A. Hendler. Efficient management of very large ontologies. In *Proceedings of the The Fourteenth National Conference on Artificial Intelligence*, pages 442–447. AAAI Press, 1997.

[Tat77] Austin Tate. Generating project networks. In *Proceedings of the Fifth International Joint Congress on Artificial Intelligence*, pages 888–893. Morgan Kaufmann, 1977.

[TDL98] Austin Tate, Jeff Dalton, and John Levine. Generation of multiple qualitatively different plan options. In *Proceedings of the Fourth International Conference on Artificial IntelligencePlanning Systems*, pages 27–34. AAAI Press, 1998.

[TH03] Annette Teije and Frank Harmelen. Describing problem solving methods using anytime performance profiles. In *Proceedings of the International Joint Conference on Artificial Intelligence Workshop on Ontologies and Problem Solving Methods*, 1999 [cited August 20, 2003]. Available from: http://www.cs.vu.nl/ frankh/abstracts/IJCAI99-PSM.html.

[Tur03] Roy Turner. Orca: Intelligent adaptive reasoning for autonomous underwater vehicle control. In *Proceedings of the FLAIRS International Workshop on Intelligent Adaptive Systems*, 1995 [cited August 20, 2003]. Available from: http://cdps.umcs.maine.edu/Papers/1995/FLAIRS/body.html.

[Wag00] Thomas Wagner. *Toward Quantified Control for Organizationally Situated Agents.* PhD thesis, University of Massachusetts, 2000.

[Wag03] Thomas Wagner. Conversation about the complexity of TÆMS task structure scheduling. Personal Conversation, July 10, 2003.

[WAS98] Daniel Weld, Corin Anderson, and David Smith. Extending graphplan to handle uncertainty and sensing actions. In *The Fifteenth National Conference on Artificial Intelligence*, pages 897–904. AAAI Press, 1998.

[Wd03] David Wilkins and Marie desJardins. A call for knowledge-based planning. In *AIPS Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning*, 2000 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/wilkins00call.html.

[WDS96] Mike Williamson, Keith Decker, and Katia Sycara. Unified information and control flow in hierarchical task networks. In *Proceedings of the AAAI Workshop on Theories of Action, Planning, and Robot Control: Bridging the Gap*, pages 142–150. AAAI Press, 1996.

[Wel94] Daniel Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[WGP03a] Thomas Wagner, Valerie Guralnik, and John Phelps. A key-based coordination algorithm for dynamic readiness and repair service coordination. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, page To Appear. ACM Press, 2003.

[WGP03b] Thomas Wagner, Valerie Guralnik, and John Phelps. Taems agents: Enabling dynamic distributed supply chain management. In *To appear in the Journal of Electronic Commerce Research and Applications*. Elsevier, 2003.

[WL01] Thomas Wagner and Victor Lesser. Design-to-criteria scheduling: Real-time agent control. *Lecture Notes in Computer Science*, 1887:128, 2001.

[WL03] Thomas Wagner and Victor Lesser. Relating quantified motivations for organizationally situated agents. In *Intelligent Agents VI: Agent Theories, Architectures, and Languages*, 1999 [cite August 20, 2003]. Available from: http://dis.cs.umass.edu/ wagner/htmlpapers/mq/.

[WM98] David Wilkins and Karen Myers. A multiagent planning architecture. In *Artificial Intelligence Planning Systems*, pages 154–163. AAAI Press, 1998.

[WPQ+03] Thomas Wagner, J. Phelps, Y. Qian, E. Albert, and G. Beane. A modified architecture for constructing real-time information-gathering agents. In *Proceedings of Agent-Oriented Information Systems*, 2001 [cited August 20, 2003]. Available from: http://www.aois.org/2001/Wagner-Abstract.html.

[WS03] William Waite and Anthony Sloane. Software synthesis via domain-specific software architectures. Technical Report CU-CS-611-92, 1992 [cited August 20, 2003]. Available from: http://citeseer.nj.nec.com/1778.html.

[ZK03] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. In *AI Magazine*, 2003 [cited August 20, 2003]. Available from: http://rakaposhi.eas.asu.edu/learn-plan-aimag.pdf.

APPENDICES

Appendix A

FORMAL PROBLEM DEFINITIONS

The following sections present a more formal definition of the two key problems solved by Tripbot: the data gathering problem and the information generation (or fusion) problem.

For Tripbot, the data gathering problem was that of deciding on the best data sources to query to obtain the data necessary to generate the required information results. Tripbot, for this part of the problem, is given a set of data subject areas to gather data from, a set of data sources, constraints on the cost and duration of query operations, and constraints for the resulting trip itinerary information. This part of the solution attempts to find the best set of information sources to query to produce the best data, which, in turn, will provide the best, most complete set of subsolutions in the information generation solution phase.

The problem is depicted graphically in Figure A.1. We view this gathering problem, more precisely, as an iterative decision problem over a 4-tuple, $(\Psi, \tau, \gamma, \epsilon)$, where Ψ is set of vectors, of which for each the first element is a unique subsolution generator function and the remaining elements are *characteristic functions*, which are discussed in more detail later, but which are essentially distributional functions characterizing the subsolution generator random variable and that return a distribution characterizing an attribute of the associated subsolution, i.e., for our purposes, cost, quality, or duration. The subsolution part of each element of Ψ is a function which produces, with some probability, some data (for example, within the Tripbot domain, a set of potential airline reservations) — there is almost always a small probability that no data will be produced, but a

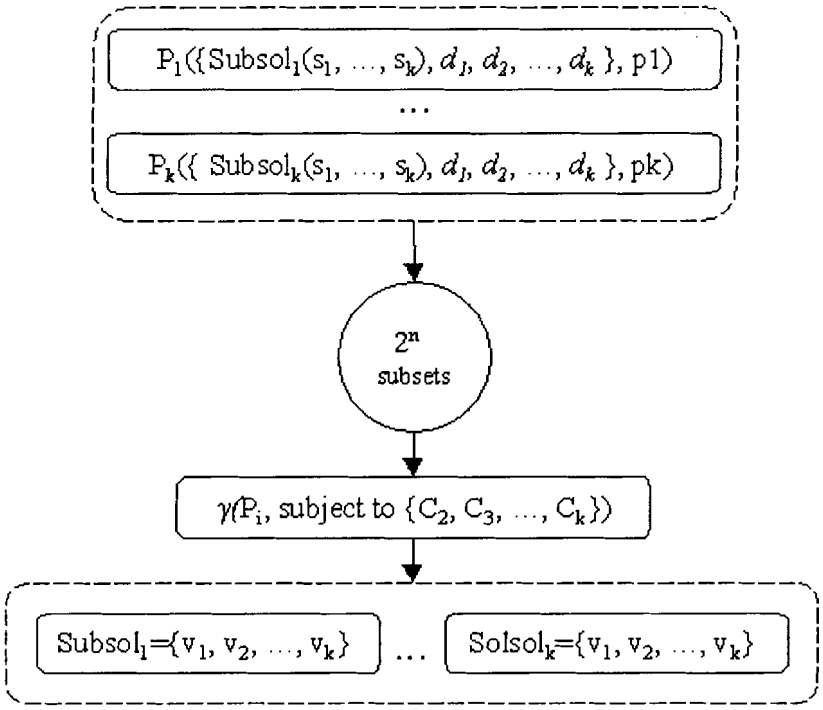

Figure A.1: The data gathering problem.

much larger probability that the data will be returned in more time than the expected duration.

τ is a set of sets over the powerset of Ψ that characterize "complete" data gathering solution sets, i.e., those sets that will produce all the required data to generate the information required by the query.

γ is an objective function for the inclusion of individual possible subsolutions in the probabilistic search over Ψ[1]. γ is thus used to decide which elements of τ will be included in query actions by the system.

Finally, ϵ provides a time constraint on the search of τ. The problem here is to maximize expected value of γ within the specified ϵ and γ constraints of the expected environmental conditions.

Generating trip itineraries for Tripbot clients means consolidating the results returned in the data gathering phase — although additional data gathering actions may be necessary. There may be

[1]In our solution, we use the TÆMS criteria definition, discussed in more detail later.

many combinations of destinations, intermediary stops, transportation choices, recreation choices, and accommodation choices generated from data gathering. Tripbot must use the information provided to produce feasible trips that meet the customer's requirements within a specified ϵ−time bound.

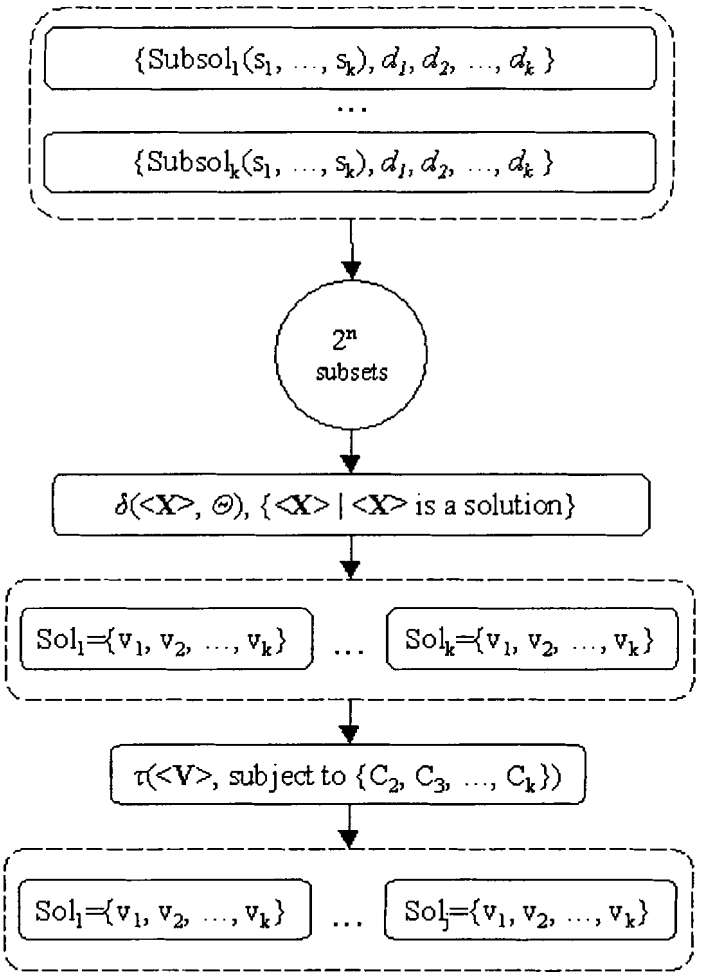

Figure A.2: The information generation problem.

The problem is depicted graphically in Figure A.2. The information generation problem, is an iterative decision problem over a 4-tuple, $(\Gamma, \xi, \gamma, \epsilon)$, where Γ is a set of vectors, defined as Ψ above, except that the first element in each vector is a unique subsolution of a solution. For Tripbot this solution is part of the query result — a part of an itinerary, such as a potential hotel reservation.

ξ is a partial order on the elements in Γ — this partial order specifies the order in which parts of the trip may take place; e.g., a flight is taken from the origin city to a destination city before a flight is taken from a destination city to the origin city. This ordering enables the establishment of completion conditions for the information generation problem. A complete trip starts at the origin location, visits the destination location, and then returns to the origin location.

γ is an objective function for the inclusion of individual possible subsolutions in the probabilistic search over Γ. The same objective function is applied to both the data gathering and information generation phase because both seek to maximize the value of the returned itineraries.

ϵ, as defined above, is a time constraint on the total computation time available to solve an instance of the information generation problem. The problem is to maximize the value of γ.

Notable differences between the data gathering problem and the information generation problem are that:

- The solution to the data gathering problem produces new data, i.e., new state information that is not derived from existing state information; and
- There is an additional source of uncertainty in the data gathering problem about the quantity and expected characteristics of the new data produced.

It is the presence of uncertainty in unstructured data gathering operations that particularly distinguishes Tripbot's problem from database query optimization. The difficulty in the data gathering problem is to generate *likely* good subsolutions for a *likely* good solution. The problem in the information generation problem is deciding upon which subsolutions to fuse together to generate the best result possible, within a set of constraints on the characteristics of the solution and within an ϵ time bound. This contrasts distinctly with database query optimization whose techniques are index and subquery reordering, which are complex in their own right but which always return complete, meaning optimal results, at least if the parameter of solution time is disregarded.

Appendix B

COMPUTATIONAL COMPLEXITY

Here we prove that TÆMS task structure scheduling is NP-Hard. [Wag03] stipulates that checking the validity of a schedule derived from an arbitrary TÆMS task structure should be a low-order polynomial operation (or better), so a proof of NP-completeness is likely possible. Checking the optimality of a schedule in the worst case requires checking each of the following number of schedules, where where n is the number of methods in the TÆMS task structure:

$$\sum_{i=1}^{n} \left(\begin{array}{c} i \\ \\ k \end{array} \right) !$$

Theorem B.0.1 *TÆMS task structure scheduling (TSS) is NP-hard.*

Proof: We transform SAT to TSS. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set of boolean variables and let $C = \{c_1, c_2, \ldots, c_n\}$ be a finite set of disjunctive clauses. Each clause c_i contains a set of variables. The SAT problem is to find a truth assignment for V that satisfies all the clauses in C.

For the purposes of this transformation, we assume that an arbitrary total order exists on C. We also assume that an arbitrary total order exists on the variables for each clause $c_i \in C$. $v_{i,j}$ then corresponds to the j^{th} variable of clause c_i.

We must construct a TÆMS task structure such that a schedule of length $|C|$ exists if and only if C is satisfiable. For each $c_i \in C$ there is a TÆMS task, t_i, with $|c_i|$ child tasks of method type.

Each method $m_{i,j}$, that is a child of t_i exclusively corresponds to variable $v_{i,j}$

The methods $\{m_{i,1}, \ldots, m_{i,k}\}$ that are children of task t_i each have an identical TÆMS method characteristics, where the outcome density is 1.0, there is one outcome, and the quality, cost, and duration characteristic distribution functions return the real value 1.0 with complete certainty.

The quality accumulation function governing the relation between task t_i and each of its child methods, $m_{i,j}$ is the $Max()$ function, which functionally is interpreted as a logical *or*.

Each task t_i is then made a child of a TÆMS task group node, $t_{satisfy\_all}$. The quality accumulation function governing the relation between $t_{satisfy\_all}$ and each of its child tasks, t_i is the $All()$ function, which functionally is interpreted as a logical *and*.

The TÆMS *disables* interrelationship is used to model variable negation. We now assume that there is a lookup table, LUT that contains a mapping from every variable $v_{i,j}$ to its corresponding method $m_{i,j}$ and that also contains the reverse mapping. The following procedure is then used to create the disables to model variable negation in the TÆMS task structure rooted at $t_{satisfy\_all}$ New terminology is introduced: $C[i]$ returns the i^{th} clause of the totally ordered set C, and the variable construct $v_{i,j}$ is assumed to have two fields, where $v_{i,j}.value$ returns the element of V for which $v_{i,j}$ is a copy, and $v_{i,j}.negated$ is a boolean field that returns **true** if this variable copy is negated.

The procedure **GENERATE-DISABLES** generates the disables links in the TÆMS task structure from logically opposing variable instances for each $c_i \in C$. It does this in quadradtic time, approximately $\theta(\frac{|C|*(|C|-1)}{|C|})$.

GENERATE-DISABLES(V, C, LUT, $t_{satisfy\_all}$)

```
1  for(i=0 to size(C))
2   for(j=0 to size(C[i]))
6    for(k=j+1 to size C[i])
7     if(v_{i,j}.value==v_{i,k}.value)
8      if(v_{i,k}.negated!=v_{i,j}.negated)
9       t_{satisfy_all}.addDisables(LUT.getMethod(v_{i,j}), LUT.getMethod(v_{i,k}), BI);
10   for(l=i+1 to size(C))
11    for(m=0 to size C[i])
12     if(v_{i,j}.value==v_{l,m}.value)
13      if(v_{l,m}.negated!=v_{i,j}.negated)
14       t_{satisfy_all}.addDisables(LUT.getMethod(v_{i,j}), LUT.getMethod(v_{l,m}), BI);
```

The transformation from SAT to TSS is given in Figure B.1. The construction can be clearly accomplished in polynomial time. All that remains to be shown is that C is satisfiable if and only if $t_{satisfy\_all}$ produces a schedule of length $|C|$.

First, suppose that S is a schedule of length $|C|$, scheduled from a task structure $t_{satisfy\_all}$ that was generated in the manner that we described above. In order for such a schedule to exist, there would have to exist in the schedule at least one method, $m_{i,j}$ from each task t_i. And, due to the $Max()$ quality accumulation function governing task t_i's relation to its child methods, there could be at most one method included in a schedule that is a child of task t_i; in the case where more than one method could be included, one is picked at random.

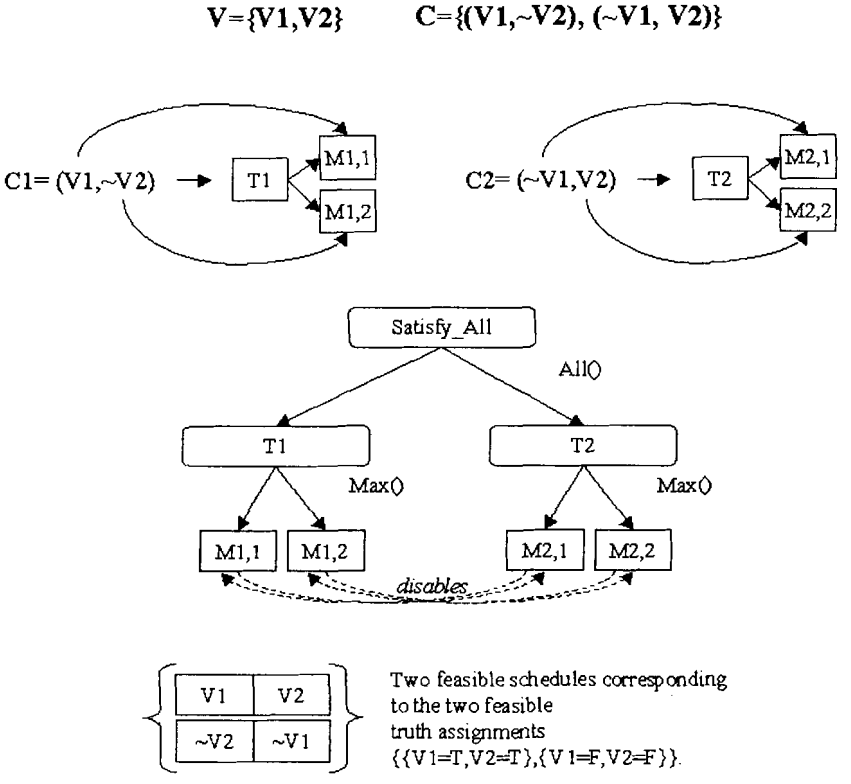

Figure B.1: Depiction of SAT to TÆMS task structure transformation.

The inclusion of method $m_{i,j}$ under task t_i is equivalent to the satisfaction of clause c_i by a truth assignment to variable $v_{i,j}.value$ that makes $v_{i,j}$ evaluate to **true**. Now, for any method $m_{i,j}$ corresponding to variable $v_{i,j}$ included in the schedule, $disables$ interrelationships between $m_{i,j}$

and any method $m_{l,m}$ corresponding to a variable $v_{l,m}$, where $v_{l,m}.negated$!=$v_{i,j}.netgated$, prevent the method $m_{l,m}$ from being included in a schedule containing $m_{i,j}$. This means that method $m_{i,j}$ corresponding to a truth assignment for variable $v_{i,j}.value$ will be included in the schedule in correspondence with one and only one truth assignment.

The assignment of truth values to $v_{i,j}.value$ corresponding to the inclusion of $m_{i,j}$ ensures that each variable has only one truth assignment. The $Max()$ quality accumulation function governing the task subtask relation ensures that one and only one method from each task is included in the schedule. This corresponds to the inclusion of one and only one **true** variable from each clause in SAT. Finally, the length of the schedule, $|C|$, indicates that the assignment of truth values that make each variable $v_{i,j}$ corresponding to $m_{i,j}$ in the schedule **true** in c_i satisfies C, since C is the conjunct of the disjuncts c_i.

Now, conversely, suppose that $A : V \rightarrow \{true, false\}$ is a satisfying truth assignment for C. This means that each clause $c_i \in C$ has at least one variable $v_{i,j}$ that evaluates to **true**. The corresponding TÆMS task structure includes one and only one method $m_{i,j}$ under each task t_i derived from each $c_i \in C$, corresponding to the variable $v_{i,j}$ that evaluates to **true**. Since there are $|C|$ clauses containing one such variable, this will produce a schedule containing $|C|$ methods, i.e., a schedule of size $|C|$.

BIOGRAPHY OF THE AUTHOR

John Phelps was born in Ellsworth, Maine. He graduated from John Bapst Memorial High School. He received his Bachelor of Science degree in Computer Science from the University of Maine in May, 1999. He is a candidate for the Master of Science degree in Computer Science from the University of Maine in August, 2003.