Table of Contents

1. **Main**
2. **Introduction**
   + Concepts
   + Textual Syntax
   + Data Types
3. **Elements**
   + Nodes
     - Methods
     - Tasks
     - Task Groups
   + QAFs
     - Min
     - Max
     - Sum
     - All
     - Seq Min
     - Seq Max
     - Seq Sum
     - Seq Last
     - Exactly One
     - Last
     - Sigmoid
   + Resources
     - Consumable
     - Non-Consumable
   + Interrelationships
     - Enables
     - Disables
     - Facilitates
     - Hinders
     - Produces
     - Consumes
     - Limits
     - Multiple Effects
   + Agent
4. **Examples**
   + Getting Dinner
   + Alternatives
   + Non-Local Methods
   + Non-Local Tasks
   + Inter-Agent IRs
   + Making Coffee
5. **Modeling Notes**
   + Rules of Thumb
   + Views
   + Active Tasks
   + Identifying Coordination
6. **Translations**
   + MDP
   + JIL
7. **Pre-Taems**
   + Syntax
     - Variables

- Control
- Functions

+ Examples
  - PCT

8. **Proposed Extensions**
   + Weights
   + Co-occurance
   + Iteration
   + Meta Operations
   + Virtual Nodes
   + Last QAF

References

Appendix

+ Commitments
  - Local
  - Non-Local
+ Schedules
+ Schedule Setting Bundle
+ Scheduler Criteria
+ Inconsistencies

The Taems White Paper

Bryan Horling, Victor Lesser, Regis Vincent, Tom Wagner,
Anita Raja, Shelley Zhang, Keith Decker, and Alan Garvey
Multi-Agent Systems Lab
University of Massachusetts
Amherst, MA 01003

This document is designed to provide the reader with four loosely grouped points of interest about the Taems modeling language. The first is a historical, purpose-oriented overview of why Taems came into existence. The second, comprising the bulk of the information in the document, covers the actual structure of Taems - the textual representation, field contents, semantics and the like. We recommend you read the blocks of texts at the top of each part of this section, and use the field specifications as a reference tool. The third covers how Taems is used, covering modeling examples and sample applications. Finally, we also include information on where Taems is heading, and cover topics closely related to Taems, such as commitments and schedules.

This document is very much intended to be a working paper, describing our specification of the format and semantics of the Taems model, along with thoughts about how Taems is being used to handle real problems. Given that Taems has been around for several years now, this is also an effort to unify the various views on Taems, from both written and implemented sources.

Many individuals have contributed to both the written and intellectual content presented in the document. In particular we would like to thank Keith Decker, who's doctoral thesis this is based upon.

For optimal viewing, we recommend you read this document with a browser that understands tables, frames and cascading style sheets (CSS1). Alternately, you can download a PDF version of the paper (the formatting is close to, but doesn't exactly match, the web version). If you have any questions or comments about this document or its contents, feel free to contact us at www@mas.cs.umass.edu.

Introduction

Overview of Taems: a formal, domain-independent framework

Taems models problem-solving activities of an intelligent agent operating in environments where responses by specific deadlines may be required, where the information required for the optimal performance of a computational task may not be available, where the results of multiple agents' computations (to interdependent subproblems) may need to be aggregated together in order to solve a high-level goal, and where an agent may be contributing concurrently to the solution of multiple goals. The implication of the real-time requirement is that the agents should be able to produce results of varying quality based on available time. Thus, the representation must be able to specify the possibility that there are multiple ways to accomplish a goal that trade off the time (and possibly other types of cost) to produce a result for the quality of the result. The possibility of incompleteness of the local information will, by necessity, lead to agents working in a satisficing mode in which problem solving is structured to operate effectively with missing information. For this reason, the representation includes what we will call "soft" coordination relationships that define the implications, in terms of both result quality and computation duration, of specific information arriving prior to the start of the computational task. Other relationships, describing how activities affect and are affected by resource availability in the operating environment may also be defined. Additionally, the effect of an agent's activities may not be quantifiable from a local perspective; instead it may need to be measured from the perspective of how it contributes to the solution of a high-level goal. Thus, there needs to be a representation of how progress towards the achievement of a goal occurs incrementally as the result of multiple activities. Finally, the representation of agent activity should allow for the possibility that individual agent's activities contribute to different and independent high-level goals.

Another way of understanding the basis of Taems is viewing it as model of agent's partial view of a distributed goal tree representation of its activities such as seen in Figure 2.1.

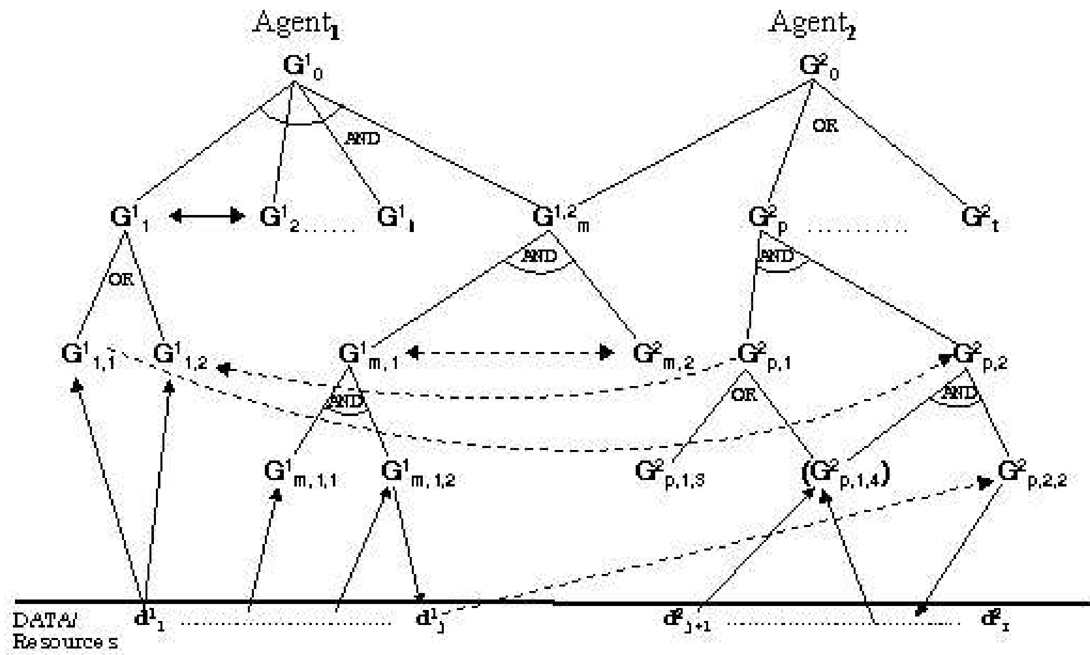


Figure 2.1: An example distributed goal tree.

Taems represents agent activity in terms of task structures at multiple levels of abstraction, each with a deadline. The goal of the agent or agents is to maximize the sum of the quality achieved for each task group before its deadline. A task group consists of a set of tasks related to one another by a subtask relationship that forms a graph (see Figure 2.2). Tasks at the leaves of the tree represent executable methods, which are the actual instantiated computations or actions the agent will execute to produce some amount of quality (in the figure, these are shown as boxes). The circles higher up in the tree represent various subtasks involved in the task group, and indicate precisely how quality will accrue depending on what methods are executed and when. The arrows between tasks and/or methods indicate other task interrelationships, in this case quantitative in character, where the execution of some method will have a positive or negative effect on the quality or duration of another method. The presence of these interrelationships make this an NP-hard scheduling problem.

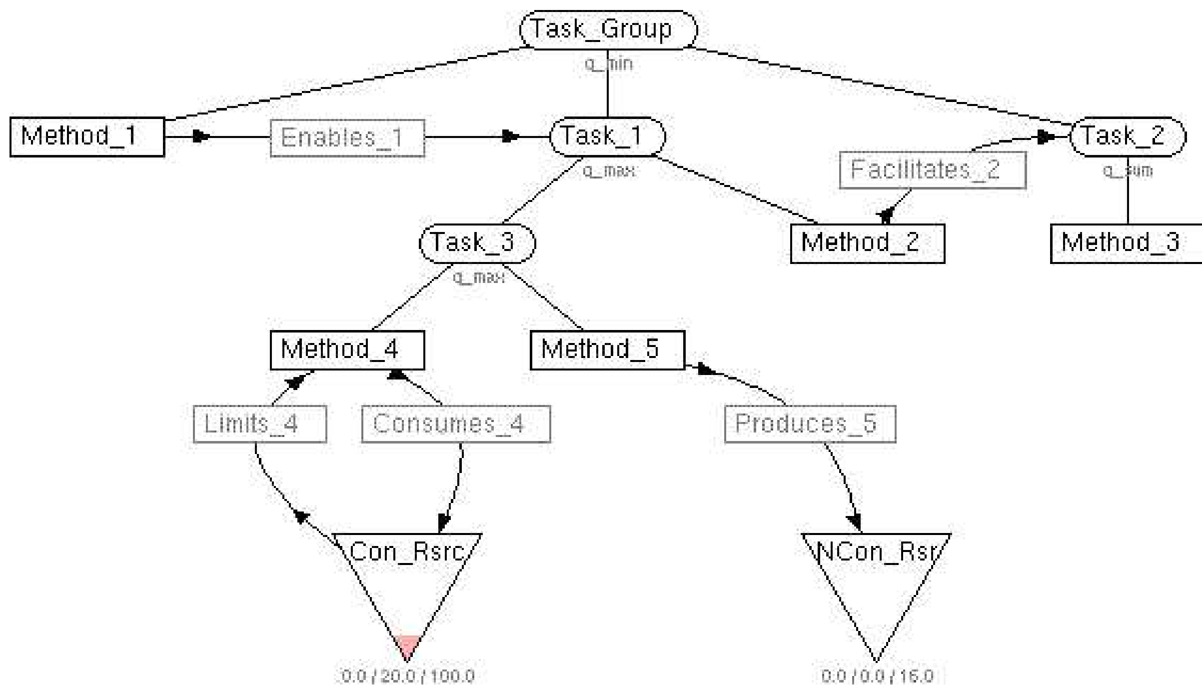


Figure 2.2: An example Taems task structure. [src]

This representation framework allows the following type of domain-independent reasoning to be implemented: predictions about future agent activity (their timing properties and the importance and quality of their output), how activities relate to each other not only in terms of precedence but more complex relationships which effect their timing and output characteristics, how results are aggregated into higher level output, and how the quality of that output relates to the quality of the input. It is this type of reasoning that is crucial in order to construct agent coordination strategies.

Concepts

Taems (also known as TAEMS, TÆMS and Tæms) was designed as a modeling language for describing the task structures of agents. The acronym stands for Task Analysis, Environmental Modeling and Simulation. In this overview we will cover some of the high level ideas that underly this acronym and give Taems its capabilities. There is nothing in Taems that precludes it from being used outside of an agent or multi-agent system, but you will see in this document that most of the distinguishing characteristics of Taems are agent related, including its ability to represent capabilities of remote agents, and explicitly represent the interdependencies that exist between them and those of the local agent.

So what do we mean by "task structure"? In any sophisticated agent system, the designer will inevitably arrive at a point where the agent must reason about its potential actions in the context of its working environment. So, presented with a given situation, what should an agent do? What goals can and should it be trying to achieve? What actions are needed to achieve those goals? What are the implications of those actions, and of actions performed by remote agents, on the agent's local state? There is a whole host of questions that an agent will need to be answered if it is to reason about its situation and act intelligently. Thus, the agent must have some representation of what its capabilities are. One such representation is called a task structure - something which describes the tasks the agent may perform. Taems improves upon conventional task structures by adding such features as quantitative action characterizations, explicit models of local and remote interactions and mechanisms to represent the wide range of ways a particular task can be achieved.

While the details associated with Taems can be daunting, there are in reality just a few simple concepts behind its structure and function. A Taems task structure is essentially an annotated task decomposition tree (actually a graph, but we will refer to is as a tree for simplicity here). The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. Below a task group there will be a sequence of tasks and methods which describe how that task group may be performed. Tasks represent sub-goals, which can be further decomposed in the same manner. Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform.

The structure above will work out to be a tree structure containing goals and sub-goals that can be achieved, along with the primitive methods needed to achieve them. Annotations on a task describe how its subtasks may be combined to satisfy it. Another form of annotation, called an interrelationship, describes how the execution of a method, or achievement of a goal, will affect other nodes in the structure. For instance, the execution of method A may enable the execution of method B. In other words, method B cannot be successfully performed before A is successfully completed. Several types of interrelationships exist to describe various types of situations.

Now that we can represent the local capabilities of an agent, we also need to be able to model potential interactions with other agents. Above we conceptually framed a single task tree structure - now envision a series of such structures. Tasks may be shared between the trees - indicating that one agent has knowledge of what another agent's capabilities are. Interrelationships may also span the trees, indicating for instance that a particular method agent A can perform enables the achievement of a task that a different agent B might choose to work on. These "nonlocal" interrelationships implicitly describe points where negotiation or coordination may be desired - if one agent can affect another, then it might make sense to explicitly exploit or avoid those interactions.

Much of what is seen in this structural view is also quantitatively described. Methods have expected execution characteristic descriptions. The effects of interrelationships are given with various numerical distributions. Resource usage, and the negative effects arising from a constraining limit of that resource, is also provided, as are the specific ways that the qualities of subtasks are combined to derive the quality of the parent task.

Taems then was designed to provide a way to model the sophisticated capabilities, alternative methods and activity-related effects that an agent may posses or exhibit. The structural view gives is agent a clear notion of how these traits are organized, while the quantitative aspects allow the agent to compare and contrast possible plans, predict their effects, and reason about the need for coordination with other agents. With this amount of detail in hand, it is appropriate to use Taems both in real agents and also as a support for simulated environments. Taems provides the information agents need to make informed decisions while also storing the detail necessary to effectively simulate the environment and agent actions.

Textual Syntax

You'll notice from the start that the syntax used to describe Taems is quite Lisp-ish, which reveals its original design platform. Just to get us started, here's a small example of what the textual representation (conventionally called *Textual Taems* or *TTaems*) looks like:

```
; The Task Group
(spec_task_group
   (label Root)
   (agent Agent_A)
   (subtasks Method_1)
   (qaf q_min)
)

; The Method
(spec_method
   (label Method_1)
   (agent Agent_A)
   (supertasks Root)
   (outcomes
      (Outcome_1
         (density 1.0)
         (quality_distribution 10.0 1.0)
         (duration_distribution 6.0 1.0)
         (cost_distribution 5.0 1.0)
      )
   )
)
```

Example 2.2.1: TTaems representation of a simple task structure.

The above is probably the simplest complete task structure you'll run across. It consists of a single root node, which we'll call a *task group*, which is the parent to a single child, which we'll call a *method*. The semantics behind these names and structures will be covered later in this document.

As in Lisp, most everything important is wrapped with a pair of parentheses. In general, most elements in TTaems follow this rough attribute/value form:

```
(field_name field data)
```

...where `field_name` is an unbroken word or phrase (that is, it contains no whitespace characters - it is a Symbol, as defined in the Data Types page) describing the contents of the element, and `field data` is the actual data or value associated with that name. In general, the data itself may contain whitespace characters but no newlines, although in some cases (such as the `outcomes` seen above) complex values may require a multi-line definition.

So, in Example 2.2.1, the `label` of the task group is `Root`, and the `duration_distribution` of `Outcome_1` of method `Method_1` is "`6.0 1.0`".

Also included in the above example are a couple lines that start with a semicolon. Again, as in Lisp, these are comments. Any line starting with a semicolon (optionally preceeded by whitespace) is considered a comment and will be thrown out when the file is parsed. In general, whitespace is also ignored outside of the field data, so you can indent and space out elements as you see fit. In some cases, however, whitespace may be recognized and incorporated in field data (it will typically be clear when this is the case).

A TTaems description consists of a number of objects, where each object is described within a self-contained `(spec_* ... )` block, and has a label unique within the structure. There are two such objects shown in Example 2.2.1. Objects may appear in any order within the description, and fields within the objects may contain forward and back references to other objects.

The relative ordering of fields within objects is fixed, so parsers are not guarunteed to correctly parse your textual specification if you provide fields out of order. The correct ordering will be implied by the given ordering of the fields in the table specifications.

All field names are case sensitive. Unless specified otherwise, you should also consider field data to be case sensitive as well.

Data Types

In many of the sections in this white paper, you will run across specification tables, describing the different fields associated with a structure, the types of data they may contain and weather they are optional or not. Within the data type column we will use several predefined keywords to describe the type of data stored in those fields. These keywords are defined on this page.

Symbol
: A single arbitrary word containing no whitespace, composed of the characters [A-Za-z0-9] and symbols "_" or "-".
  Ex. `foobar`, or `foo_bar`, or `23-foo-bar-lane`.

Agent Symbol
: This type follows the same construction as a regular symbol, with the additional directive that it should match the given name of an agent residing in your system or known to the owner of the Taems structure.
  Ex. `Agent_A`, or `Joe`, or `007`.

Predefined Symbol
: This type follows the same construction as a regular symbol, but the allowed list of symbols is fixed and predefined. An enumeration of the permitted symbols will accompany the field description.

Integer
: An integer, plain and simple. Reasonable bounds on this seem to be [-32768...32767]. A -1 is frequently used in this field to indicate that the value is unknown. You should be able to fill optional integer fields with a -1 and get the same results as if the field were ommited.
  Ex. `-1`, `13`, `42`.

Float
: A float value. In some implementations this might actually be stored as a Double.
  Ex. `1.0`, `0.3333333`, `-15.7`.

Distribution
: A series of float values, arranged logically in pairs. The first value in each pair represents a data point, while the second is the density or weight associated with that data point. The sum of the densities should be 1.0. The typically meaning to this structure is "V1 will happen P1 per cent of the time, V2 will happen P2 per cent of the time", and so on. So, in the example below, the value 15 will occur 30% of the time, while 11.2 is expected to occur 70% of the time. Implicit in this description is the idea that at runtime, one will see each of the given values instantiated, proportional to their respective densities, as a result for whatever field the distribution describes.
  Sometimes, in examples, we will refer to a distribution field as though it were a single number, e.g. "*if distribution field X has a value of 23.4...*". When this is done, we are referring to the case where X contains a distribution of one or more value pairs, and the specific value 23.4 has been selected from that list to instantiate the current situation.
  Ex. `15 0.3 11.2 0.5 -2.75 0.2`

* List
: A list type is a series of whatever type is given, separated by whitespace characters. Multiple whitespace characters are ignored.
  Ex. Symbol List -> `a b  c    d e`

Special

When indicated, the format of the data type is particular to the specific field it is being used in. A complete description of the data type will appear in the field description.

None

A field with this tag has no data, only the name of the field will appear. Conceptually, fields of this type represent flags, and indicate a true boolean state for whatever attribute they describe if they are present.

Taems Elements

A Taems tasks structure is made up of a number of items, such as methods, tasks, interrelationships, and resources, which I'll refer to collectively here as *elements*. In this section we will look at each of the elements that serve as the building blocks of a Taems structure, how it is used and what its specifications and semantics are. This is ment both to serve as a reference and as a primer for the different possible uses of each element.

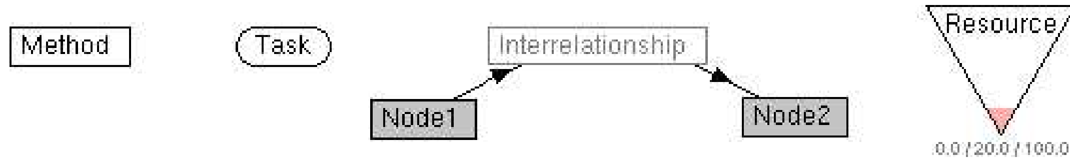


Figure 3.1: The four kinds of Taems elements.

In this section we also look at two "markup" items used in Taems, quality accumulation functions and the agent object. These are not first class elements in the sense described above, but are important and prominant enough to warrant close inspection and discussion regardless.

The four structural elements in a Taems structure share some common characteristics that will be described here in lieu of repeating the information for each object. The Agent object also shares the `spec_attribute` and `label` fields.

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no

Figure 3.2: The standard element fields.

Field Overview

`spec_attributes`

The attributes field is space available within the element where you can store arbitrary data. It is organized in the same manner as other fields in the object, as an attribute/ value pair, where the attribute is a Symbol. The value can be any data when in binary form, although certain restrictions apply to field in TTaems, notably that the textual form of the value cannot span multiple lines or contain mismatched parentheses (for parsing reasons).

In general, one cannot assume that any particular data will be available in the attributes field, although you may define ad hoc standards or conventions in your agents where important data is stored here. In these cases it is perfectly acceptable (even reccomended) that you use attributes to mark up your task structure in a way that makes Taems a better representation for the information relevant to your particular problem.

In some implementations (notably our current Java implementation), the value field of attributes has

a special form, facilitating the translation of the data. A field in this form will look like this:

```
(field_name field_type field_data)
```

where `field_type` is a recognized conversion type such as `Integer`, `String`, or `Vector`, and `field_data` is data of that type encoded in the expected format. The type and encoding format are those used by utilities.Converter, you may consult the API documentation for that Object for more information.

`label`

This is the "name" of the object in question. This label should be unique within the structure it is contained in, although in some implementations this may not be enforced. A triple consisting of an object's label, agent and type should always be unique within a structure.

`agent`

The agent field defines the owner of the element in question - for instance, the agent which is capable of performing that task or method. Elements in a structure may have an agent name different from that of the agent scheduling or coordinating over the structure. This indicates that the scheduling agent posesses some knowledge of what other agents' capabilities are, and how those capabilities can affect its local execution. The connectivity and relationships between remote and local methods can affect how the local agent acts and communicates (e.g. I have to ask someone to do this method for me).

For instance, in Figure 2.2, suppose that the entire structure is owned by Agent_A, with the exception of Method_1, which has an `agent` field of Agent_B. Since Method_1 enables Task_1, if Agent_A hopes to accomplish Task_1, it must first ensure that Method_1 has been sucessfully performed. In this case, it can use the `agent` field to determine which remote agent can perform the action, and coordinate with that agent to ensure that Method_1 is done in a timely manner.

Alternate interpretation: The agent field indicates who's view of a particular method it is. This permits a scheduling component to recognize similarly named elements in a structure and differentiate correctly between them.

Nodes

Although the term is sometimes used differently under different circumstances (notably the Java Taems implementation), we will refer to those elements residing in the tree/graph-like portion of the Taems task structure as *nodes*. Depending on your point of view, there are three kinds of nodes, or just two kinds, one of which has a different name under certain circumstances. You can see examples of these nodes in the following figure.

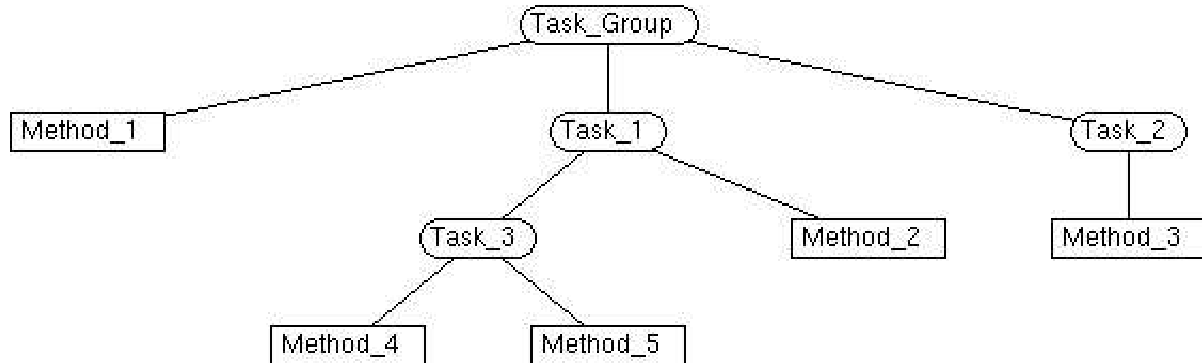


Figure 3.1.1: Examples of node types. [src]

We'll look at the different node varieties from a bottom-up perspective, starting at the bottom of the Taems structure and working our way up to the root. At the lowest level of the structure you will find *methods*, which represent primitive executable actions. In graphical representations of Taems, methods are generally depicted as rectangles. These represent things you can actually *do*, while the remainder of the nodes serve to organize the methods in some coherant fashion.

One or more methods will in turn be organized under a *task*, as is seen under `Task_3`. Tasks may also be parents to tasks themselves, so that a hierarchy of tasks and methods may be constructed, as under `Task_1`. Conceptually, a task is some action or sub-goal that can be achieved in the astract, while the methods or tasks below it represent the more specific elements required to achieve that task. The exact manner in which a task's children may be executed, and the semantics behind their results, will be covered later.

At the root of a Taems structure there will be a special kind of a task, called a *task group*. Task groups may be thought of as an overall goal associated with a structure, where all the elements suspended beneath it are there to describe how that goal may be achieved.

Methods

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
agent	Agent Symbol	no
supertasks	Symbol List	no
outcomes	Special	no
arrival_time	Integer	yes
earliest_start_time	Integer	yes
deadline	Integer	yes
start_time	Integer	yes
finish_time	Integer	yes
accrued_time	Integer	yes
nonlocal	None	yes

Figure 3.1.1.1: Specification of spec_method.

Methods are in some sense the most primitive, yet also the most important, elements in a Taems task structure. They represent the things your agent can actually do, and all the other elements within the structure are there to organize the methods and annotate their effects in a manner consistent with some view of the world.

Methods are generally represented graphically as a rectangle. The method specification contains information pertinent to the execution of the method, primarily the different possible outcomes of the method and slots for the anticipated or observed characteristics of an actual execution. Here is an example method, as encoded in TTaems:

```
(spec_method
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Method_1)
   (agent Agent_A)

   (supertasks Task_1)
   (outcomes
      (Outcome_1
         (density 8.0)
         (quality_distribution 7.0 1.0)
         (duration_distribution 5.0 0.3 6.0 0.7)
         (cost_distribution 6.0 1.0)
```

```
      )
      (Outcome_2
         (density 2.0)
         (quality_distribution 9.0 1.0)
         (duration_distribution 15.0 0.3 10.0 0.7)
         (cost_distribution 6.0 1.0)
      )
   )

   (arrival_time 0)
   (earliest_start_time 0)
   (deadline 0)

   (start_time 10)
   (finish_time)
   (accrued_time 3)

   (nonlocal)
)
```

Figure 3.1.1.2: An example method in TTaems.

All methods should have associated with them a list of one or more potential `outcomes`. This outcome list describes what the quality, duration and cost distributions will be for each possible result the method might achieve when it is executed. The duration associated with an outcome simply specifies how long the method will execute before it completes. Cost is a unit-less property that defines the "amount" of some tangible or intangible thing that is used when the method is executed. Conventionally, this might represent a financial cost to the agent, but it is not restricted to this view. Instead, cost could represent processing power, or some other property that the agent would try to minimize when selecting a method for execution. Quality is similar to cost, in that it is also a unit-less property. It represents an abstract characteristic of method execution that the agent would want to maximize - intuitively, a high quality method is better. The actual quality value associated with a method is meaningless by itself - it is only relevant and meaningful when compared with the quality of other outcomes in the environment (both of that method, and of other methods). Thus, a quality outcome of 100 would mean nothing to the agent until it is able to compare that number to some predefined scale or the quality of another method.

Field Overview

`spec_attribute`
: See Elements. The attributes field is sometimes used in a method to store such things as the maximum quality attainable by the method, or indicate its place in the current schedule.

`label`
: See Elements.

`agent`
: See Elements.

`supertasks`
: This field lists the supertasks of the method, e.g. those tasks which have the method as one of their subtasks. The fact that this field can contain a list of such supertasks implies that in general the task-subtask organization of Taems structures are graphs (specifically DAGs), not trees. To a certain degree, this information is redundant with the subtasks field in a task specification.

`outcomes`

This field describes the possible outcomes associated with the execution of a method. One or more outcomes may be listed here, each with a label unique to the outcome set. Some implementations will require that an outcomes field be present for the method object to be considered valid. The specification for the outcome field is as follows:

Field Name	Data Type	Optional
`density`	Float <= 1.0	no
`quality_distribution`	Distribution	no
`duration_distribution`	Distribution	no
`cost_distribution`	Distribution	no

Figure 3.1.1.3: Specification of individual outcomes.

`density`

The probability that this outcome will be the one that is exhibited when the method is executed. The sum of the densities of the different outcomes in a single method should sum to 1.0.

`quality_distribution`

This distribution describes the expected quality associated with the outcome.

`duration_distribution`

This distribution describes the expected duration associated with the outcome.

`cost_distribution`

This distribution describes the expected cost associated with the outcome.

The outcomes for a method that has been executed may be updated to reflect the actual results of the execution. In this case, the method would have a single outcome (density 1.0), where the Q/D/C distributions each have a single value that matches the observed results.

`arrival_time`

Specifies the arrival time of the method. This is used to keep track of when tasks arrived at the local agent (if it had been sent by another agent as part of a negotiation process, for instance).

`earliest_start_time`

Indicates to the agent the earliest possible time at which the method can be executed. This is typically defined by a remote agent as part of a coordination/ negotiation commitment, although it may be used to indicate analogous local conditions as well.

In scheduling terms, `earliest_start_time` is a hard constraint. Schedulers respecting this field will not schedule a method to begin its execution before this time.

`deadline`

Indicates to the agent the latest possible time at which the method should be completed. This is typically defined by a remote agent as part of a coordination/ negotiation commitment, although it may be used to indicate analogous local conditions as well.

In scheduling terms, `deadline` is a hard constraint.

`start_time`

Specifies the absolute time at which the method is expected to start. This is typically filled in by a scheduler, and may be used to detect aberrant behavior when available.

`finish_time`
: Specifies the absolute time at which the method is expected to finish. A method with a non-negative `finish_time` is assumed to have been executed, with respect to supertasks and interrelationships.

`accrued_time`
: Specifies the accrued time of the method. It is expected that this field will be updated as new information becomes available while the method is executing.

`nonlocal`
: A flag which, if present, indicates that the method is not local to the agent. In general, this specifies that this is an activity that the agent is aware of (and presumably has a quantitative description of), but it can not perform itself. This is useful when modeling methods that one agent can request another to do for it (e.g. contract net) - the method is present and described so it may be easily scheduled over, but the flag allows the agent to recognize that further coordination may be necessary for the action to actually take place.
Note: Some implementations may expect the deprecated tag `method_is_non_local` for this field.

Tasks

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`supertasks`	Symbol List	no
`subtasks`	Symbol List	no
`qaf`	Predefined Symbol	no
`arrival_time`	Integer	yes
`earliest_start_time`	Integer	yes
`deadline`	Integer	yes

Figure 3.1.2.1: Specification of `spec_task`.

Where methods represent the activities an agent may perform in actuality, tasks represent activities an agent may perform in the abstract. They represent the idea that a perticular action may be performed, or goal achieved, by encapsulating the specific mechanisms used to implement those activities. A task's primary purpose is to serve as a focal point for its subtasks, by both giving them order and a notion of how they may be combined to achieve the task. This "combination" is represented by a quality accumulation function, or QAF, which defines how the quality of the subtasks is used to compute the quality of the task.

Tasks are generally represented graphically as an oval, or rectangle with rounded corners.. Below is an example task in TTaems form:

```
(spec_task
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Task_1)
   (agent Agent_A)

   (supertasks Root)
   (subtasks Task_2 Method_1 Task_3)
   (qaf q_sum)

   (arrival_time 0)
   (earliest_start_time 0)
   (deadline 20)
)
```

Figure 3.1.2.2: An example task in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`supertasks`
: This field lists the supertasks of the task, e.g. those tasks which have this task as one of their subtasks. The fact that this field can contain a list of such supertasks implies that in general the task-subtask organization of Taems structures are graphs (specifically DAGs), not trees. To a certain degree, this information is redundant with the subtasks field.

`subtasks`
: This field lists the subtasks of the task, e.g. those tasks or methods which, when completed or achieved in some manner, may lead to the completion of the parent task. Given that some QAFs have sequencing effects, the order of the subtasks in the list is pertinant and should be respected by parsers and Taems implementations.

`qaf`
: The QAF, or quality accumulation function, associated with a task defines how the quality, or lack thereof, of subtasks is combined to determine the quality of the parent task. The particulars of how this is done, and the semantics of the different symbols will be covered in the QAF section. Accepted symbols: See the QAF section for a complete list.

`arrival_time`
: Specifies the arrival time of the task.

`earliest_start_time`
: Indicates to the agent the earliest possible time at which the task can be executed. This is typically defined by a remote agent as part of a coordination/negotiation commitment, although it may be used to indicate analagous local conditions as well. The semantics of this are that any method arising as a subtask of the task (or as a subtask of any other task below the method) should not be started before this time. So conceptually you can view all methods below this task as having an earliest start time of the given time or later.

Tasks Groups

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`subtasks`	Symbol List	no
`qaf`	Predefined Symbol	no
`arrival_time`	Integer	yes
`earliest_start_time`	Integer	yes
`deadline`	Integer	yes

Figure 3.1.3.1: Specification of `spec_task_group`.

A task group is essentially identical to a task, serving the same general function and having the same form with one exception - it has no supertasks. Conceptually, a task group is the highest level goal that the agent is aware of. A given structure may have multiple task groups, and arising subgraphs may be interdependant in various ways, but to the agent the various task groups should represent wholly separate goals which can potentially be addressed.

Multiple task group objects may be present in any given Taems structure, although some implementations may conceptually view these as all being grouped under another (implicit) task group object.

Below is an example task group in TTaems form:

```
(spec_task_group
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Task_1)
   (agent Agent_A)

   (subtasks Task_1 Method_1 Task_2)
   (qaf q_min)

   (arrival_time 0)
   (earliest_start_time 0)
   (deadline 10)
)
```

Figure 3.1.3.2: An example task group in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`subtasks`
: See Tasks.

`qaf`
: See Tasks.

`arrival_time`
: See Tasks.

`earliest_start_time`
: See Tasks.

QAFs

Quality Accumulation Functions, or QAFs as we will refer to them from here on, define how the quality of a task's subtasks can be used to calculate the quality of the task itself. Imagine, for instance, a task which has three methods, A, B and C, as its descendants. There are many possible execution combinations and orderings using these methods, some of which may have correct semantics and some which don't.

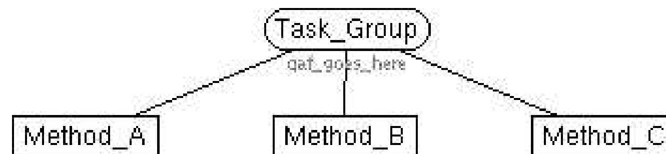


Figure 3.2.1: A three-method task. [src]

Suppose for instance, that {A, B, C} are {clean-bedroom, clean-kitchen, clean-livingroom}. In this case (barring external influences), it wouldn't really matter what order they are done in, and the more that gets done, the better. The total quality of this process would in some sense be the total quality accrued from all three tasks - the sum of the individual qualities.

In another situation we may have entirely different needs. Think of a situation where you have a meal you have to cook, and your three ways of doing it are {use-stove, use-microwave, use-bunsen-burner}. Now, we clearly don't want to be cooking things using more than one technique (assuming each task runs for some fixed, scaled period of time equal to the "total cooking" duration for the food on that device). So we don't want to execute multiple methods, and therefore aren't concerned about ordering. The final quality therefore would be that of whichever method executes first, and if any others execute the quality will drop to zero (as we've then burnt our dinner) - we want exactly one method to run.

Now consider a situation where we're getting ready for a big night out. We've got to {buy-clothes, get-hair-cut, and take-shower}, again assuming independence between methods. In this case we still have three things to do, and they can be done in any order, but our evening companion is fairly picky and will needle us on whatever is done the worst. If we get clothes that don't match that's a problem, and if we don't take a shower then that's a big problem. So the final quality here will be determined by whatever task doesn't go so well - it will be the minimum of the qualities of the respective methods.

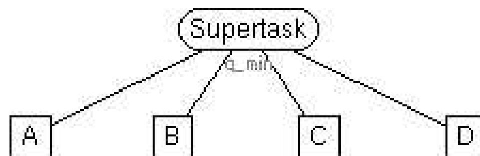
So, as you can see, these are three situations where the task structure is essentially the same, but we require different semantics associated with how that overall task's quality is determined. This is where QAFs come in. A QAF is a tag associated with a task or task group that does exactly this. In the first example, a `q_sum` QAF would be applied. In the second, a `q_exactly_one` would be used, and in the third a `q_min` would best describe our predicament. The full range of defined QAFs will be discussed in this section.

Graphically, the QAF typically appears underneath the task it is attached to, as in Figure 3.2.1. The QAF is also not a first-class object in the same sense of a task or method, it appears only as a field in a task itself (see Figure 3.1.2.2 for an example). As such, it does not have attributes or an agent, and the label is implicitly the type of QAF.

Min

A min QAF is functionally equivalent to an AND operator. It says that the quality of the supertask is equal to the minimum quality of its subtasks. Since methods which have not been performed are assumed to have zero quality, this means that for the supertask to obtain a non-zero quality, all the subtasks must be executed.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



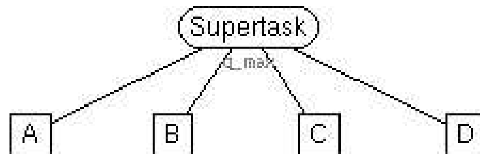
1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	2
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	2
A (5)	B (2)	D (0)	C (7)	0

Figure 3.2.1.1: Final quality examples for `q_min`. [src]

Max

A max QAF is functionally equivalent to an OR operator. It says that the quality of the supertask is equal to the maximum quality of any one of its subtasks. Thus, a efficiency-oriented scheduling process might choose to perform only the task with the highest potential quality, while a survivability-oriented one might select several to be performed, knowing that if one fails there will be others which might take its place.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



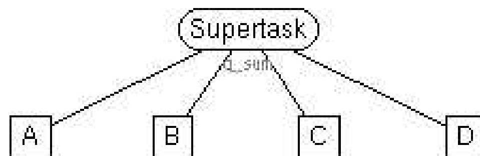
1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	7
A (5)	B (2)	C (7)	-	7
B (2)	A (5)	D (3)	C (7)	7
A (5)	B (2)	D (0)	C (7)	7

Figure 3.2.2.1: Final quality examples for `q_max`. [src]

Sum

The sum QAF says that the quality of the supertask is equal to the sum of the qualities of its subtasks, regardless of order or which methods are actually invoked. The essentially models a process where each additional method the agent chooses to perform will increase the final quality of the task - the accumulated quality increases monotonically.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	17
A (5)	B (2)	C (7)	-	14
B (2)	A (5)	D (3)	C (7)	17
A (5)	B (2)	D (0)	C (7)	14

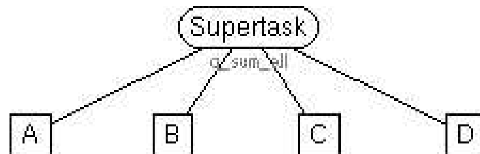
Figure 3.2.3.1: Final quality examples for `q_sum`. [src]

All (Sum All)

Sum all works much the way sum does, with the exception that all the subtasks must have completed for the supertask to have quality. Thus, the quality of the supertask will not incrementally increase as sum does, but will instead make one big leap up to its final value once all the subtasks have completed.

Note that the subtasks do not necessarily have to complete *successfully*, they just need to have been attempted, so that a failed method (which has a final quality of 0) will not add any quality to the supertask, but neither will it prevent it from accruing any quality.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	17
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	17
A (5)	B (2)	D (0)	C (7)	14

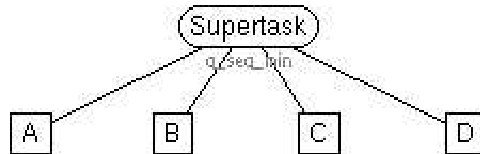
Figure 3.2.4.1: Final quality examples for `q_sum_all`. [src]

Seq Min (Sequenced Min)

Seq_min works similar to a normal min, except that that all the subtasks must have been performed in order for the supertask to have any quality. When all the subtasks have completed in order, the supertask will obtain the same quality as the supertask with the minimal quality.

Sequence is defined by the order of the subtasks below the task, i.e. the order that is given in the task's `subtasks` field. "In order" in this case says that the successor to a method cannot begin before the predecessor has completed (strict sequential execution). If a task is included under a sequence QAF (as opposed to a simple method), then the start time of the task is the earliest start time of any of its subtasks, and the finish time is the latest finish time of any of its subtasks.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	2
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	0
A (5)	B (2)	D (0)	C (7)	0
A (5)	B (2)	C (7)	D (0)	0

Figure 3.2.5.1: Final quality examples for `q_seq_min`. [src]

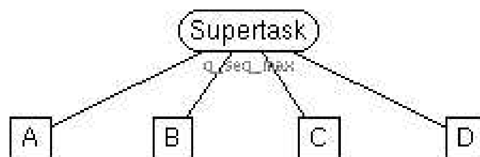
Seq Max (Sequenced Max)

Seq_max works similar to a normal max, except that that all the subtasks must have been performed in order for the supertask to have any quality. When all the subtasks have completed in order, the supertask will obtain the same quality as the supertask with the maximum quality.

Sequence is defined by the order of the subtasks below the task, i.e. the order that is given in the task's `subtasks` field. "In order" in this case says that the successor to a method cannot begin before the predecessor has completed (strict sequential execution). If a task is included under a sequence QAF (as opposed to a simple method), then the start time of the task is the earliest start time of any of its subtasks, and the finish time is the latest finish time of any of its subtasks.

Note that the subtasks do not necessarily have to complete *successfully*, they just need to have been attempted, so that a failed method (which has a final quality of 0) will not add any quality to the supertask, but neither will it prevent it from accruing any quality.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	7
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	0
A (5)	B (2)	D (0)	C (7)	0
A (5)	B (2)	C (7)	D (0)	7

Figure 3.2.6.1: Final quality examples for `q_seq_max`. [src]

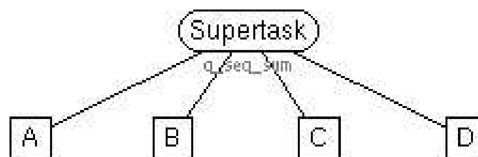
Seq Sum (Sequenced Sum)

Seq_sum works similar to a normal sum, except that that all the subtasks must have been performed in order for the supertask to have any quality. When all the subtasks have completed in order, the supertask will has the combined quality of all its subtasks as its quality.

Sequence is defined by the order of the subtasks below the task, i.e. the order that is given in the task's `subtasks` field. "In order" in this case says that the successor to a method cannot begin before the predecessor has completed (strict sequential execution). If a task is included under a sequence QAF (as opposed to a simple method), then the start time of the task is the earliest start time of any of its subtasks, and the finish time is the latest finish time of any of its subtasks.

Note that the subtasks do not necessarily have to complete *successfully*, they just need to have been attempted, so that a failed method (which has a final quality of 0) will not add any quality to the supertask, but neither will it prevent it from accruing any quality.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	17
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	0
A (5)	B (2)	D (0)	C (7)	0
A (5)	B (2)	C (7)	D (0)	14

Figure 3.2.7.1: Final quality examples for `q_seq_sum`. [src]

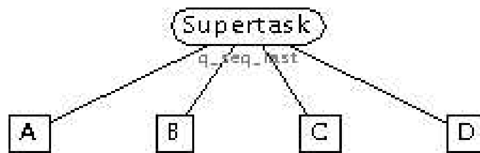
Seq Last (Sequenced Last)

The semantics of Seq_last state that the quality of the task will be equal to the quality of the last subtask to be executed. In addition, all the subtasks must have been performed in order for the supertask to have any quality.

Sequence is defined by the order of the subtasks below the task, i.e. the order that is given in the task's `subtasks` field. "In order" in this case says that the successor to a method cannot begin before the predecessor has completed (strict sequential execution). If a task is included under a sequence QAF (as opposed to a simple method), then the start time of the task is the earliest start time of any of its subtasks, and the finish time is the latest finish time of any of its subtasks.

Note that the subtasks do not necessarily have to complete *successfully*, they just need to have been attempted, so that a failed method (which has a final quality of 0) will not add any quality to the supertask, but neither will it prevent it from accruing any quality.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



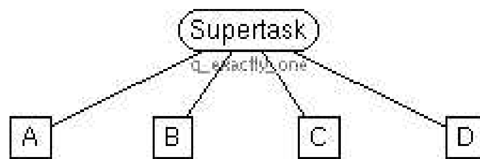
1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	3
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	0
A (5)	B (2)	D (0)	C (7)	0
A (5)	B (2)	C (7)	D (0)	0

Figure 3.2.8.1: Final quality examples for `q_seq_last`. [src]

Exactly One

An exactly one QAF is functionally equivalent to an XOR operator. The quality of the supertask is the quality of any of its subtasks, provided that only one subtask has quality. If more than one has been successfully performed, the quality of the task drops back to zero.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



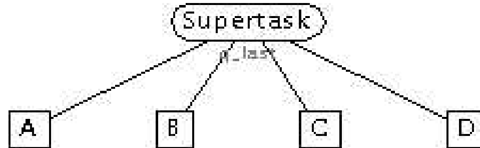
1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	0
A (5)	B (2)	C (7)	-	0
B (2)	-	-	-	2
B (0)	A (2)	-	-	2
B (0)	A (2)	C (7)	-	0

Figure 3.2.9.1: Final quality examples for `q_exactly_one`. [src]

Last

The last QAF says that the quality of the supertask is equal to the quality of the one subtask beneath it that was last finished.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	3
A (5)	B (2)	C (7)	-	7
B (2)	A (5)	D (3)	C (7)	7
A (5)	B (2)	D (3)	C (0)	0

Figure 3.2.10.1: Final quality examples for `q_last`. [src]

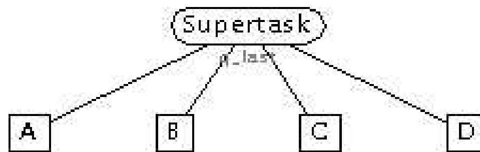
Sigmoid

Sigmoid is weird. I'll cover it in time. Note that right now no one really supports this, so don't try to use it.

```
(spec_task_group
   (label Supertask)
   (agent Agent_A)
   (subtasks A B C D)
   (qaf q_sigmoid
      (sigmoid_distribution 1.0 .34 0.9 .66 .8 .99)
   )
)
```

Figure 3.2.10.1: An example task using `q_sigmoid`.

The figure and table below show how the QAF works in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.



1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	?
A (5)	B (2)	C (7)	-	?
B (2)	A (5)	D (3)	C (7)	?
A (5)	B (2)	D (0)	C (7)	?

Figure 3.2.10.2: Final quality examples for `q_sigmoid`. [src]

Resources

We've seen thus far how to model actions and goals to this point, but what about other things in the environment? There are many times when we might wish to model the fact that a method produces something more tangible than "quality", or that it requires something be available to it other than what is specified by it's generic "cost". This is where resources come into play.

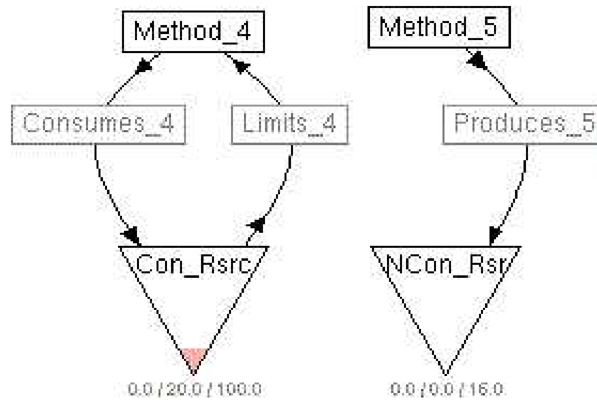


Figure 3.3.1: Resources being modeled. [src]

In Taems, a resource is a somewhat abstract concept. It has a level and bounds, and a defined behavior of either consumable or non-consumable. The resource itself is unit-less, the only thing of import to the model is weather or not the current level falls within the given bounds. The level itself is affected by the produces and consumes interrelationships which target the resource, and the effects the resource has on methods and tasks are defined by the limits interrelationships arising from it.

The following sections describe the characteristics of the different types of resources modeled by Taems, although the complete view of resources in Taems is not possible without a thorough discussion of the interrelationships that affect them. The graphical representation of a Taems resource is typically an inverted triangle, optionally with the bounds and current known state enumerated below it.

Consumable Resources

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	yes
`state`	Float	no
`depleted_at`	Float	no
`overloaded_at`	Float	no

Figure 3.3.1.1: Specification of `spec_consumable_resource`.

Consumable resources are very straightforward - if you use them, their state drops and it doesn't go back up until you produce some more. Common examples of consumable resources are printer paper, food in your fridge, gasoline in your car and lead in a pencil. The `depleted_at` field is usually set at zero, although it does not have to be (the pencil, for instance, will probably be unusable before the lead is completely used up).

In the above examples, we showed resources where you typically think as the lower bound as the important one - the act of consuming is the more pertinent to the model. It is also reasonable to think of the upper bound as being the more important, however. Consider a landfill, for instance - in this case we are more worried about producing too much trash. The same could be said of pressure inside of a boiler, or boxes in a warehouse.

```
(spec_consumable_resource
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Con_Resource_1)
   (agent Agent_A)

   (state 20.0)
   (depleted_at 10.0)
   (overloaded_at 100.0)
)
```

Figure 3.3.1.2: An example consumable resource in TTaems.

Field Overview

`spec_attribute`
    See Elements.

`label`

See Elements.

`agent`

See Elements. By convention, if the agent field is filled in on a resource, that agent is assumed to control that resource, and all coordination needs associated with the resource should be sent to that agent. If the agent field is not present, it is assumed to be a "community" resource, where some decentralized form of coordination is necessary.

Alternate View: If the field is not present, the resource is assumed to be owned by the agent reading the file.

`state`

The current known state of the resource. This field is unit-less, or perhaps a better term is that the Taems representation does not care what units are associated with the resource.

`depleted_at`

The lower bound on the resource's state. We assume that a resource may not cross this bound, and if a method attempts to do so, any limits relationships arising from the resource will be enabled. On some implementations the state may be permitted to cross this boundary, but the effects (regarding limits) will be the same.

`overloaded_at`

The upper bound on the resource's state. We assume that a resource may not cross this bound, and if a method attempts to do so, any limits relationships arising from the resource will be enabled. On some implementations the state may be permitted to cross this boundary, but the effects (regarding limits) will be the same.

Non-Consumable Resources

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
agent	Agent Symbol	yes
state	Float	no
depleted_at	Float	no
overloaded_at	Float	no

Figure 3.3.1.1: Specification of spec_non_consumable_resource.

A non-consumable resource is one where the level of the resource is instantly reset once the effecting action has stopped. You don't permanently consume a non-consumable resource (hence the name), you only temporarily alter it, and the amount changed - weather it was produced or consumed - is automatically removed or replaced once the producing method finishes. Examples of non-consumable resource include activity on a network, electrical load, brightness of a light bulb, and noise in a room.

To clarify, let's look at two cases. In the first we will model noise in a room, which has a lower bound of 0 and an upper bound of 90. Ambient noise puts the current state at around 20. When Use-Vacuum starts up, it pushes the noise level up to 80. When the vacuum is turned off, the noise level instantly drops back to 20. Another instance is bandwidth on a network. Lets say our bandwidth has a starting level of 10 and goes down to 0 as more data is transferred. One FTP session is opened, which consumes the available bandwidth down to 8. Another user fires up another session, which further reduces what's available to 5. While the second FTP session is running, the first completed, freeing up 2 units of bandwidth, which puts the actual level at 7. Finally, when the second completes the available bandwidth goes back up to 10.

```
(spec_non_consumable_resource
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Non_Con_Resource_1)
   (agent Agent_A)

   (state 20.0)
   (depleted_at 0.0)
   (overloaded_at 250.0)
)
```

Figure 3.3.1.2: An example consumable resource in TTaems.

Field Overview

`spec_attribute`
: See Elements. The attributes field is sometimes used in a resource to store such things as the point at which the resource is overloaded - the converse of `depleted_at`.

`label`
: See Elements.

`agent`
: See Elements. By convention, if the agent field is filled in on a resource, that agent is assumed to control that resource, and all coordination needs associated with the resource should be sent to that agent. If the agent field is not present, it is assumed to be a "community" resource, where some decentralized form of coordination is necessary.
: *Alternate View*: If the field is not present, the resource is assumed to be owned by the agent reading the file.

`state`
: The current known state of the resource. This field is unit-less, or perhaps a better term is that the Taems representation does not care what units are associated with the resource.

`depleted_at`
: This currently holds the upper bound on the resource's state, but but in later versions will hold the lower bound as expected. We assume that a resource may not cross this bound, and if a method attempts to do so, any limits relationships arising from the resource will be enabled. On some implementations the state may be permitted to cross this boundary, but the effects (regarding limits) will be the same.

`overloaded_at`
: The upper bound on the resource's state. We assume that a resource may not cross this bound, and if a method attempts to do so, any limits relationships arising from the resource will be enabled. On some implementations the state may be permitted to cross this boundary, but the effects (regarding limits) will be the same.

Interrelationships

An important quality that sets Taems apart from other hierarchical modeling systems is its explicit, quantitative representation of the interactions that actions have on other elements in the agent's environment. In Taems, these interactions are called interrelationships, or sometimes non-local effects (NLEs) to distinguish the action's "local" results (quality, cost, duration) from the impact it has on other methods or resources.

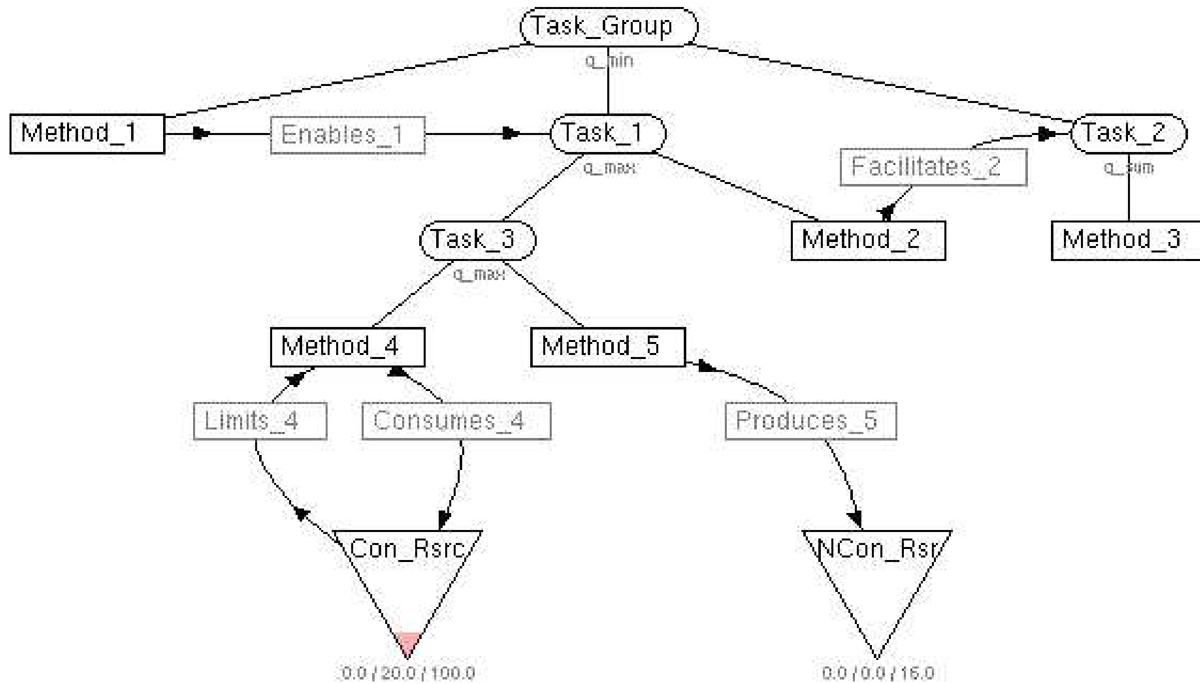


Figure 3.4.1: A Taems task structure with several interrelationships. [src]

Consider for instance the case where we have two methods Get-Into-Car and Drive-Away. Clearly these two have some sort of relationship, since one cannot be done before the other. In some sense, a successful Get-Into-Car action is a prerequisite for Drive-Away. This is an example of a "hard" interrelationship between two activities.

Now consider the case where we have another pair of methods, Warm-Leftovers and Eat-Leftovers. As most people can attest, cold leftovers are edible, and can provide some satisfaction, but if one were to warm them first the experience can be much more rewarding. So here we have a different relationship, since we don't have to warm our food, but if we do the actual act of eating will somehow have greater quality. This is an example of a "soft" type of interrelationship.

There are other kinds of relationships that don't affect activities, but instead have some sort of impact on the surrounding environment. Going back to our car example, we can see that Drive-Away will affect the Gasoline resource, specifically by consuming it. Analogously, the Gasoline resource will affect Drive-Away, in that if the resource is spent, Drive-Away won't accrue much quality (unless it's all downhill, of course). So given that we have models for both resources and methods in our Taems

structures, the idea of interrelationships between them comes quite naturally.

In the first example, we say that Get-Into-Car `enables` Drive-Away. In the second, Warm-Leftovers `facilitated` Eat-Leftovers, and in the third we were `consuming` Gasoline or `limiting` Drive-Away if none was available. As insinuated above, interrelationships also have a state, either active or not, which determines if their defined effects are manifested. This state is controlled by the state of the source, or from node, which will be described briefly below and in more detail in the respective interrelationship sections.

Interrelationships give the designer a powerful way of representing complex interactions, and the quantitative values associated with those effects can form the foundation for very sophisticated behaviors. Graphically, interrelationships are typically shown as rectangles above or overlapping a line, with arrowheads pointing from the source node to the target.

Interrelationships are very similar from one type to the next. They all have names, sources and targets which have the same semantics. Below, we cover the fields that (most of) the interrelationships have in common.

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`from_outcome`	Symbol	yes
`to`	Symbol	no
`delay`	Distribution	yes

Figure 3.4.2: Fields shared among different interrelationship types.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: The node from which the interrelationship arises. The active state of the interrelationship (that is, weather its effects will be manifested or not) is determined by the `from` node (see also `from_outcome` and `delay` below). If the from node is either a task or a method, then the interrelationship is active if that node has nonzero quality. If the from node is a resource, then the interrelationship (necessarily a limits) is active if the resource has violated its stated bound(s).

`from_outcome`

If the `from_outcome` field is filled in, then the interrelationship is interpreted as arising from a particular outcome from that method. In this case, the active state of the interrelationship is determined not only by the presence of nonzero quality on that method, but that that quality must also have been obtained from the particular outcome the interrelationship is attached to.
Note: Limits interrelationships do not have this field, since they arise from resources.

`to`

The `to` field indicates the target of the interrelationship. If the interrelationship is active, then its effects will be manifested in some manner in the target node. So, if an active facilitates points to Method_B, then Method_B's execution characteristics will be favorably adjusted. Analogously, if an active consumes points to Resource_1, then Resource_1's state will decrease by the specified amount.
(This matter is still under discussion) Note that when an interrelationship targets a task node (as opposed to a leaf method), it is the task itself which is affected, not its subtasks. For example, in Figure 3.4.1 above we see that Method_1 enables Task_1, meaning Method_1 must be successfully completed before Task_1 can succeed. In this case, Task_3 and Method_2 (and Method_4 and Method_5), *can* successfully complete without Method_1. It is just the quality accrued by Task_1 itself which would be inhibited in this case. So if Method_1 failed, Task_3 could still have quality, while Task_1 could not.

`delay`

This indicates the amount of time which will pass, after other activation requirements have been met, before the interrelationship will actually become active. Consider a generic enables interrelationship IR:

```
(spec_enables
   (label IR)
   (agent Agent_A)

   (from Method_1)
   (to Method_2)

   (delay 5 1.0)
)
```

Further assume that Method_1 completes with quality 10 at time 25. Under these conditions, IR will become active at time 30, due to its delay of 5. If Method_1 had finished with zero quality, IR would never become active.
Note: Produces, consumes and limits interrelationships do not have this field. Some older implementations may have `delay` implemented as a simple Integer, rather than a distribution.

Enables

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`from_outcome`	Symbol	yes
`to`	Symbol	no
`delay`	Integer	yes

Figure 3.4.1.1: Specification of `spec_enables`.

An enables interrelationship is of the "hard" variety, meaning that it essentially acts as a binary switch. In this case, the target method or task cannot accrue quality until the enabling interrelationship is active. Enablements have proven to be very common in our modeling history, as they essentially represent the idea that the relevant tasks and methods have an ordering constraint, e.g. if some X enables Y then X cannot be started until Y completes.

Note also that if you have a series of methods or tasks, all falling under the same task and also all requiring ordering constraints, that it might be more consise to use one of the `seq` QAF variants under the grouping task rather than explicitly link them together with a series of enablements. So, provided the enables come from the methods (not particular outcomes), and they have no delay, the following two structures are semantically the same:

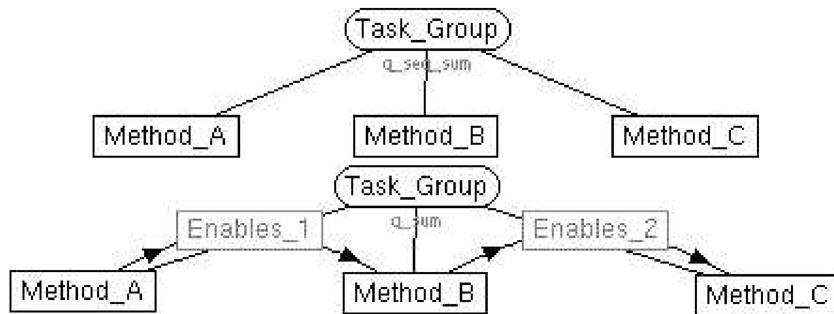


Figure 3.4.1.2a & b: Comparison of QAF sequenced versus explicitly enabled ordering constraint models. [src a] [src b]

It is worth mentioning again that if your enables are complex (arise from a specific outcome or have a delay), or if not all methods or tasks under the task are to be ordered, then a `seq` QAF will not give you the correct semantics.

```
(spec_enables
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Enables_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Method_2)

   (delay 0 1.0)
)
```

Figure 3.4.1.3: An example enables interrelationship in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`delay`
: See Interrelationships.

Disables

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
agent	Agent Symbol	no
from	Symbol	no
from_outcome	Symbol	yes
to	Symbol	no
delay	Integer	yes

Figure 3.4.2.1: Specification of spec_disables.

Where enables indicates that one method cannot run before another completes, disables indicates the exact converse. If X disables Y, then Y cannot be run sucessfully (i.e. accrue quality) after the disablement is active. A good example of this is Seal-Envelope and Put-Letter-In-Envelope, where we say that a disablement exists arising from SE and terminating at PLIE. If Seal-Envelope has been successfully executed, then Put-Letter-In-Envelope cannot, because the envelope is now presumably sealed. Like enables, this is also a hard interrelationship, and indicates an ordering constraint between the relevant nodes (you may also want to review the seq comments noted in enables).

```
(spec_disables
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Disables_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Method_2)

   (delay 0 1.0)
)
```

Figure 3.4.2.2: An example disables interrelationship in TTaems.

Field Overview

spec_attribute
: See Elements.

label
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`delay`
: See Interrelationships.

Facilitates

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`from_outcome`	Symbol	yes
`to`	Symbol	no
`quality_power`	Distribution	no
`duration_power`	Distribution	no
`cost_power`	Distribution	no
`delay`	Integer	yes

Figure 3.4.3.1: Specification of `spec_facilitates`.

Facilitates is an example of a "soft" interrelationships. Instead of inducing a binary, on/off type behavior in the target, soft interrelationships can alter the target's characteristics in a more continuous manner. Consider the following figure:

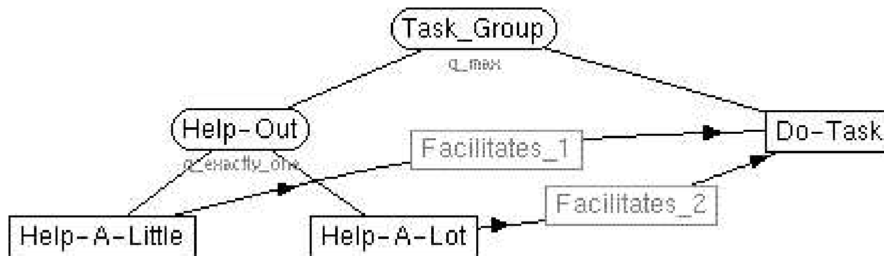


Figure 3.4.3.2: A method with the potential of being facilitated. [src]

In this case, the agent has a choice of doing one of two "helper" actions, each of which has a different level of facilitation effects on the Do-Task method. The agent making use of this structure would first have to determine if there were resources available (and the rewards high enough!) to take advantage of either of the facilitations. If this were true, then the agent would need to decide which of the helper methods provides the most benefits for an acceptable cost.

Attached to the facilitates are three distributions which describe how the interrelationship affects the quality, duration and cost of the target. Like all distributions, each one contains a list of value/ density pairs, where each value has a probability equal to its density of occurring. The values stored in the distribution represent the quantitative adjustments that will occur in the target method when the interrelationship is active. These adjustments are given as percentages of the whole, so if the quality

power of a facilitates is 0.6, the the target quality will be increased by 60% when the facilitates is active. Similarly, if a duration or cost power of 0.4 is given, the target duration or cost will be reduced by 40%.

These values are further weighted based on the performance of the source node. Conceptually, the values given in the distribution represent the effects when the source has achieved maximal quality. If the source does got attain maximal quality, the weights are scaled accordingly, modeling the idea that if your method does perform very well it's nonlocal effects will reflect that. So, if our source node above achieves 75% of its potential quality, then the new quality power of the facilitaties will be 0.45 (0.6 * 0.75 = 0.45). Similarly, the cost or duration power would be 0.3 (0.4 * 0.75 = 0.3). This modifier is called the *power factor* generated from the source node's quality. The general equation is:

```
q = q [+,-] q * (p * (sq / smq))
```

where `q` is the target's quality (or cost or duration), `p` is the quality (or cost or duration) power of the facilitates, `sq` is the actual quality of the source and `smq` is the maximum possible quality of the source. The term `(sq / smq)` defines the source node's power factor. The `[+,-]` represents the idea that the change may be added or subtracted depending on the trait in question (qualities are added, costs and durations are subtracted).

So let's say you have a facilitates with the following specifications. For simplicity, we'll just look at power distributions with a single outcome, although if you have more than one in a given power the mechanism is similar (you just pick a value from the distribution and proceed the same way).

```
(quality_power 1.0 1.0)
(duration_power 0.5 1.0)
(cost_power 0.0 1.0)
```

The source node of the facilitates looks like this:

```
(quality_distribution 10.0 0.4 25 0.6)
(duration_distribution 10.0 1.0)
(cost_distribution 3.0 1.0)
```

The source node's actual quality was 10, so it only achieved 40% of it's potential maximum quality of 25. The source's power factor is therefore `(10 / 25) = 0.4`. The source node's actual duration and cost are irrelevant in this context. The target method's outcome looked like this:

```
(quality_distribution 9.0 1.0)
(duration_distribution 15.0 0.3 10.0 0.7)
(cost_distribution 6.0 1.0)
```

When the facilitates becomes active, the outcome would change to look like this:

```
(quality_distribution 12.6 1.0)
  ; 9  + 9  * (1.0 * 0.4) = 12.6
(duration_distribution 12.0 0.3 8.0 0.7)
  ; 15 - 15 * (0.5 * 0.4) = 12.0
  ; 10 - 10 * (0.5 * 0.4) = 8.0
(cost_distribution 6.0 1.0)
  ; 6  - 6  * (0.0 * 0.4) = 6.0
```

In the resulting outcome, you can see that the quality was increased (by 40%), the duration was decreased (by 20%), and the cost remained the same.

```
(spec_facilitates
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Facilitates_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Method_2)

   (quality_power 0.7 0.5 0.3 0.5)
   (duration_power 0.5 1.0)
   (cost_power 0.5 0.2 0.4 0.8)

   (delay 0 1.0)
)
```

Figure 3.4.3.2: An example facilitates interrelationship in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`quality_power`
: The distribution contains a list of percentages, which indicate how much the quality of the target is increased if this interrelationship is active. The final quality of the target will be `original_quality` + `original_quality` * `quality_power` * `source_power_factor`, for each value in the target quality distribution with a selected facilitation quality power.

`duration_power`
: The distribution contains a list of percentages, which indicate how much the duration of the target is decreased if this interrelationship is active. The final duration of the target will be `original_duration` - `original_duration` * `duration_power`, * `source_power_factor` for each value in the target duration distribution with a selected facilitation duration power.

`cost_power`
: The distribution contains a list of percentages, which indicate how much the cost of the target is decreased if this interrelationship is active. The final cost of the target will be `original_cost` - `original_cost` * `cost_power` * `source_power_factor`, for each value in the target cost distribution with a selected facilitation cost power.

`delay`
: See Interrelationships.

Hinders

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
agent	Agent Symbol	no
from	Symbol	no
from_outcome	Symbol	yes
to	Symbol	no
quality_power	Distribution	no
duration_power	Distribution	no
cost_power	Distribution	no
delay	Integer	yes

Figure 3.4.4.1: Specification of `spec_hinders`.

Hinders acts in the same manner as facilitates, except that the powers affecting the target have their meanings reversed. When a hinders interrelationship is active, the target's quality is reduced by the `quality_power`, while the duration and cost are increased by the `duration_power` and `cost_power`, respectively. See the facilitates discussion for more details on how the powers are applied.

```
(spec_hinders
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Hinders_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Method_2)

   (quality_power 10.0 0.5 7.0 0.5)
   (duration_power 5 1.0)
   (cost_power 0.5 0.2 0.4 0.8)

   (delay 0 1.0)
)
```

Figure 3.4.4.2: An example hinders interrelationship in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`quality_power`
: The distribution contains a list of percentages, which indicate how much the quality of the target is decreased if this interrelationship is active. The final quality of the target will be `original_quality` - `original_quality` * `quality_power` * `source_power_factor`, for each value in the target quality distribution with a selected facilitation quality power.

`duration_power`
: The distribution contains a list of percentages, which indicate how much the duration of the target is increased if this interrelationship is active. The final duration of the target will be `original_duration` + `original_duration` * `duration_power` * `source_power_factor`, for each value in the target duration distribution with a selected facilitation duration power.

`cost_power`
: The distribution contains a list of percentages, which indicate how much the cost of the target is increased if this interrelationship is active. The final cost of the target will be `original_cost` + `original_cost` * `cost_power` * `source_power_factor`, for each value in the target cost distribution with a selected facilitation cost power.

`delay`
: See Interrelationships.

Produces

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`from_outcome`	Symbol	yes
`to`	Symbol	no
`model`	Predefined Symbol	yes
`produces`	Distribution	no

Figure 3.4.5.1: Specification of `spec_produces`.

Produces interrelationships always start at a method and terminate at a resource. They indicate that when the source node of the interrelationship has successfully executed (or is successfully executing), that some amount of the target resource will be produced, assuming no bounds violations are encountered. So if you have a a produces relationship arising from Make-Widget that goes to a Widgets resource, then it will do exactly that - increment the Widgets resource by some quantity when Make-Widget is successfully performed.

The activation criteria for this interrelationship are discussed in the `model` description field.

```
(spec_produces
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Produces_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Resource_1)

   (model per_time_unit)

   (produces 15 0.5 10 0.5)
)
```

Figure 3.4.5.2: An example produces interrelationship in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`model`
: Two types of resource production models are supported in Taems, a `per_time_unit` model, and a `duration_independent` one. In the per_time_unit model, when the interrelationship is active, it will add the amount specified in the `produces` field to the target resource at each time pulse. So if your method runs for 5 time units and has a produces of 10, then you will produce a total of 50 units of the resource. When using this model, the interrelationship is considered to be active during the entire time the source method is running (with one exception). So, if in this example the method ran during time units 7, 8, 9, 10 and 11, the 10 units of resource would be added during times 7, 8, 9, 10, and 11. Further note that if your method's quality accumulation is halted somehow (say by the negative effects of a limits or consumes interrelationship), that the produces will not be active. So, if the method above did not gain quality during time 8 and 9, then the total amount of resources produces would be 30, assuming the same total duration.

  In the duration_independent model, if the interrelationship is active, it will add the amount specified in the `produces` field to the target resource only once, when the method has completed. So if your method runs for 5 time units and has a produces of 10, then when the method has completed, 10 units of the resource will be added. When using this model, the interrelationship is considered to be active during the final unit of time the method is active. So, if in this example the method ran during time units 7, 8, 9, 10 and 11, the 10 units of resource would be added during time 11. The duration independent model does not care about quality accumulation as per_time_unit does - as long as the method has quality at its completion the interrelationship will activate.

  So, here's a table that might clarify things. We'll look at the four combinations of resource type and usage model, where each resource starts out with a level of 0. The same method execution pattern, with a produces of 10, will be used. The level of the resource at the end of each time unit is shown in the cells.

Time	Resource Type and Usage Model			
	Consumable + per_time_unit	Non-Consumable + per_time_unit	Consumable + duration_independent	Non-Consumable + duration_independent
7	10	10	0	0
8	20	10	0	0
9	30	10	0	0
10	40	10	0	0
11	50	10	10	10

Figure 3.4.5.3: Clarification of resource model and usage type effects.

See also Tom Wagner's notes on this subject.
Accepted symbols: `per_time_unit`, `duration_independent`.

`produces`

This distribution indicates how much of the resource will be produced when the interrelationship is active. The exact semantics of this are described in the `model` field.

Consumes

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`from_outcome`	Symbol	yes
`to`	Symbol	no
`model`	Predefined Symbol	yes
`consumes`	Distribution	no

Figure 3.4.6.1: Specification of `spec_consumes`.

Consumes interrelationships always start at a method and terminate at a resource. This relationship is used when you want to indicate that a particular method uses up some amount of resource during its execution. So if we image a consumes interrelationship between Use-Widget and Widgets, then Use-Widget will use up some number of widgets if it successfully executes.

The activation criteria for this interrelationship are discussed in the `model` description field.

Note: This element replaces the depricated `spec_uses` interrelationship.

```
(spec_consumes
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Consumes_1)
   (agent Agent_A)

   (from Method_1)
   (from_outcome Outcome_1)
   (to Resource_1)

   (model per_time_unit)

   (consumes 15 0.5 10 0.5)
)
```

Figure 3.4.6.2: An example consumes interrelationship in TTaems.

Field Overview

`spec_attribute`
: See Elements.

`label`
: See Elements.

`agent`
: See Elements.

`from`
: See Interrelationships.

`from_outcome`
: See Interrelationships.

`to`
: See Interrelationships.

`model`
: As with produces, two types of resource production models are supported by consumes, a `per_time_unit` model, and a `duration_independent` one. In the per_time_unit model, when the interrelationship is active, it will remove the amount specified in the `consumes` field from the target resource at each time pulse. So if your method runs for 5 time units and has a consumes of 10, then you will consume a total of 50 units of the resource. When using this model, the interrelationship is considered to be active during the entire time the source method is running (with one exception). So, if in this example the method ran during time units 7, 8, 9, 10 and 11, the 10 units of resource would be removed during times 7, 8, 9, 10, and 11. Further note that if your method's quality accumulation is halted somehow (say by the negative effects of a limits or consumes interrelationship), that the consumes will not be active. So, if the method above did not gain quality during time 8 and 9, then the total amount of resources consumed would be 30, assuming the same total duration.

  In the duration_independent model, if the interrelationship is active, it will remove the amount specified in the `consumes` field from the target resource only once, when the method starts. So if your method runs for 5 time units and has a consumes of 10, then when the method has starts, 10 units of the resource will be removed. When using this model, the interrelationship is considered to be active during the first unit of time the method is active. So, if in this example the method ran during time units 7, 8, 9, 10 and 11, the 10 units of resource would be removed during time 7.

  The issue with quality accumulation, consumes and duration independence is a tricky one. We say that if a method does not get quality, then the consumes will not activate and no resources will be used. This is true to a point, in that it will only be enforced if quality accumulation is prevented by something like an inactive enables, or an active disables, or some other failed precondition. If, however, the method is aborted, or quality accumulation is disrupted midstream, then the consumes will have already fired, and barring time travel, that's how it will have to be. In these cases, the semantics state that you've already consumed your resource, so that they cannot be replaced.

  This table should help describe resource consumption. We'll look at the four combinations of resource type and usage model, where each resource starts out with a level of 50. The same method execution pattern, with a consumes of 10, will be used. The level of the resource at the end of each time unit is shown in the cells.

	Resource Type and Usage Model			
Time	Consumable + per_time_unit	Non-Consumable + per_time_unit	Consumable + duration_independent	Non-Consumable + duration_independent
7	40	40	40	40
8	30	40	40	50
9	20	40	40	50
10	10	40	40	50
11	0	40	40	50

Figure 3.4.6.3: Clarification of resource model and usage type effects.

See also Tom Wagner's notes on this subject.

Accepted symbols: `per_time_unit`, `duration_independent`.

`consumes`

This distribution indicates how much of the resource will be consumed when the interrelationship is active. The exact semantics of this are described in the `model` field.

Limits

Field Name	Data Type	Optional
`spec_attributes`	Special	yes
`label`	Symbol	no
`agent`	Agent Symbol	no
`from`	Symbol	no
`to`	Symbol	no
`model`	Predefined Symbol	yes
`quality_power`	Distribution	no
`duration_power`	Distribution	no
`cost_power`	Distribution	no

Figure 3.4.7.1: Specification of `spec_limits`.

Note: The limits interrelationship is not all that well defined at this time, so the implementation and semantics are subject to change.

Limits interrelationships always start at a resource and terminate at a method. They are used to describe the idea that when you are out of a resource (depleted), or have more of a resource than can be held (over-produced), something bad should happen. For instance, Use-Widget has a consumes interrelationship that targets Widgets. There is also a limits interrelationship that goes from Widgets to Use-Widget. In this situation, when the capacity of Widgets is zero (its `depleted_at` state), the limits interrelationship will activate if Use-Widget is executed, which will affect the execution of Use-Widget in some negative way. A similar situation would exist if we had a produces/ limits pair running between Make-Widget and Widgets. While it is not necessary, you will probably find that whenever you use a consumes or produces interrelationship it is also reasonable to have a limits associated with it.

In the current Taems model, limits are fired whenever its source resource's state has grown or shrunk to a point beyond its bounds at the same time the target method produces or consumes that resource. Given that the state of a resource can in some implementations not go past its bounds, we imply by this statement that limits are fired whenever an action changes the state of a resource in such a way that it *attempts* to cross a bound. So in the example above, the Widget resource's state would never actually drop below zero, but the Use-Widget method would in some sense be trying to reduce its state to -1. This -1 state is never seen, but its effects will be, in the form of the activated limits. In general then, a limits interrelationship will activate if the attempted capacity `c` satisfies this boolean expression:

```
((c < depleted_at) || (c > overloaded_at)
```

Note that this does mean you cannot have a limits without a produces or consumes, although you can fake it by using one of the two with a production or consumption level of zero. Also note that at this time the limits does not differentiate weather the resource has exceeded its upper or lower bound, which can lead to

some strange behavior under rare circumstances.

```
(spec_limits
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Limits_1)
   (agent Agent_A)

   (from Resource_1)
   (to Method_1)

   (model per_time_unit)

   (quality_power 10.0 0.5 7.0 0.5)
   (duration_power 5 1.0)
   (cost_power 0.5 0.2 0.4 0.8)
)
```

Figure 3.4.7.2: An example limits interrelationship in TTaems.

Field Overview

`spec_attribute`
  See Elements.

`label`
  See Elements.

`agent`
  See Elements.

`from`
  See Interrelationships.

`to`
  See Interrelationships.

`model`
  The model discussion for limits is similar to those for consumes and produces. In a `per_time_click` model, if the limits is activated it will affect only the quality, duration or cost accumulated during the time unit it is active. In the `duration_independent` model, if the interrelationship is activated it will affect the final quality, duration, or cost accordingly. Note that this is independent of what produces or consumes interrelationship actually caused the activation. A per_time_unit produces, which activates a duration_independent limits, will affect the final Q/D/C values, not just those of that particular time unit. It is also independent of which resource type is being used.

  So, let's say your limits has an effect of reducing quality by half. Further assume that the method in question, Use-Widgets, normally runs for 5 time units, consumes Widgets during each time unit (a per time unit model) and accumulates 10 quality under optimal conditions. If the limits is duration independent, the final quality will be 5. If the limits is per time unit, and is active throughout the method's execution, it will also have a final quality of 5 (quality per unit: 1, 1, 1, 1, 1). Now lets say the limits is only active during time units 1 and 2. The duration independent limits will still give a final quality of 5, since once it has fired it only effects the final quality. The per time unit model,

however, would give us a final quality of 8 (quality per unit: 1, 1, 2, 2, 2).

See also Tom Wagner's notes on this subject.

Accepted symbols: `per_time_unit`, `duration_independent`.

`quality_power`
: This distribution describes how much the quality will be reduced if the limits is activated. The values are percentages, as in hinders (with a power factor of 1), but the effect will change the current or final quality as described in the `model` field.

`duration_power`
: This distribution describes how much the duration will be extended if the limits is activated. The values are percentages, as in hinders (with a power factor of 1), but the effect will change the current or final quality as described in the `model` field.

`cost_power`
: This distribution describes how much the cost will be increased if the limits is activated. The values are percentages, as in hinders (with a power factor of 1), but the effect will change the current or final quality as described in the `model` field.

Effects of Multiple Interrelationships

Describing the effects of several interrelationships (active or not) targeting a single node is a very complex task. As stated earlier, interrelationships are classified in three types, *hard*, *soft* and *resource based*. If several interrelationships point to the same method, then, generally speaking, the different effects are ANDed together and sorted by time (from the earliest active time to the most recent one).

It gets trickier, however, if you have a heterogenous combination of interrelationship types - such as a mixture of enables and disables. In these cases, interrelationships follow a precedence ordering: hard interrelations are most important, then soft and then resource based. In this classification, the strongest interrelationship is disables, which means that if even one disables is activated for a method, that method will never accrue quality, regardless of what other interrelationships affect it. Following disables is enables, which is similarly strict - if any enables affecting a task is not activated the task will never accrue quality. Looking at it in another way, if you have several enables affecting the same method, all of them must be active in order to enable the method.

The combination of the soft interrelationships hinders and facilitates are cumulative, and applied in the order they are activated. This means that if a method has two interrelationships affecting it, one hinders and one facilitates, in this order, the final duration distribution for the method would be Facilitates(Hinders(Duration)).

The weakest class of interrelationships are resource based. The combined effects of these can be quite complex, owing to the two models (`per_time_click` and `duration_independent`). We'll start with an example of cumulative limits.

Assume that the Volvo company expects to ship a new car to a customer once every 10 weeks. The normal cost is 500$ and it takes 10 weeks to be shipped (so it then takes 20 weeks for the customer to actually get the car - 10 to build and 10 to ship). There are a few limiting relations here, but we'll focus on the act of shipping. The first, and most obvious limiting resource is the car itself. If the car isn't available, that has an impact on shipping it. There must also be a truck available to ship the car (we'll assume the driver comes with the truck). Finally, there must still be a buyer at the dealer's end, if for some reason the original buyer cancels the order the act of shipping the car might become useless and the car needs to be shipped back to the Volvo company. Of course, this model is not realistic but the point is to illustrate the effects of the limits on the shipping cost and duration. The quality of the shipping may also be affected, but for clarity this dimension is omitted.

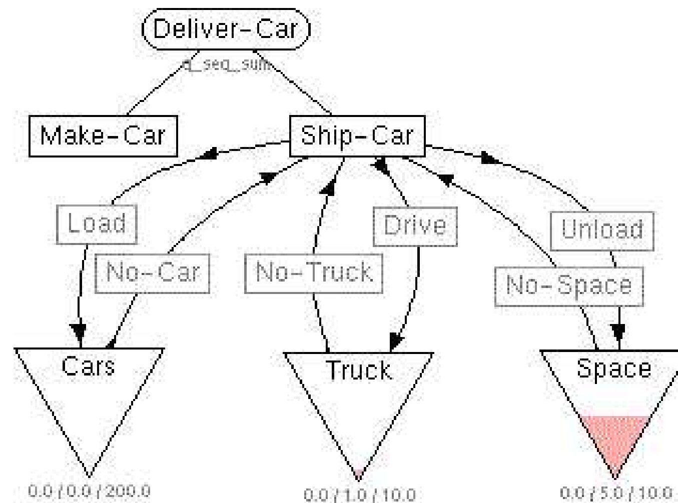


Figure 3.4.8.1: A Taems task structure modeling the car supply scenario. [src]

The figure above shows the task structure describing the above scenario. We'll now walk through a more detailed view of several examples involving the effects of these multiple limits interrelationships. It is important to note that, from the agent's perspective, these calculations are hidden - but as an agent designer you should understand them to use limits correctly in similar situations.

In the figures below, the dotted line represents the original cost accumulation over time. Even if the interrelationship is duration independant, there is a accumulation function working behind the scenes, even though the agent only sees the end result (upon completion of the method's execution). The red or blue line represents the cost accumulation function after the limits has been fired. The red or blue box on the time axis represents the pulse when the method has been overloaded.

Example 1 - If you try to ship a car before any cars are built, the limits will double the time of shipping because you have to wait 10 weeks to build a car. Of course, during the same time, the cost of shipping will increase by 50% because your truck is waiting for you.

```
(spec_limits
   (label No-Car)
   (agent Volvo)
   (from Cars)
   (to Ship-Car)
   (model duration_independent)
   (quality_power 0.0 1.0 )
   (duration_power 1.0 1.0 )
   (cost_power 0.5 1.0)
)
```

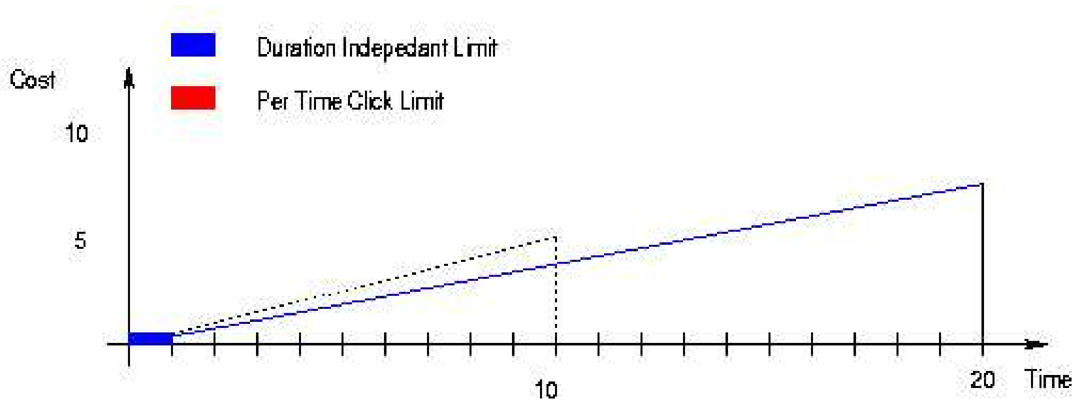


Figure 3.4.8.2: Effect of a duration independent limits on consumes.

Example 2 - If you try to unload a car without sufficient space being available in the dealer's lot, the limits will double the time of shipping because you have to take the car back and attempt to deliver it again in 10 weeks. During the same time, the cost of shipping will increase by 50%. (again note that values given here are just for example, they are not supposed to model reality carefully).

Another point to bear in mind: if when you redeliver the car 10 weeks later (at time 20), there is still no space, the limits will be fired again. Again, the duration will be increased by a factir of two (based on method duration of **20**) that will give a total duration of **40** weeks for this method (note the actual amount of increase here doubled). You will see the same effects on cost, which increases by 50% (based on a total cost of $750), producing a total cost of $1125. Of course, if space never becomes available this method will *never* finish unless the agent aborts it.

```
(spec_limits
   (label No-Space)
   (agent Volvo)
   (from Space)
   (to Ship-Car)
   (model duration_independent)
   (quality_power 0.0 1.0 )
   (duration_power 1.0 1.0 )
   (cost_power 0.5 1.0)
)
```

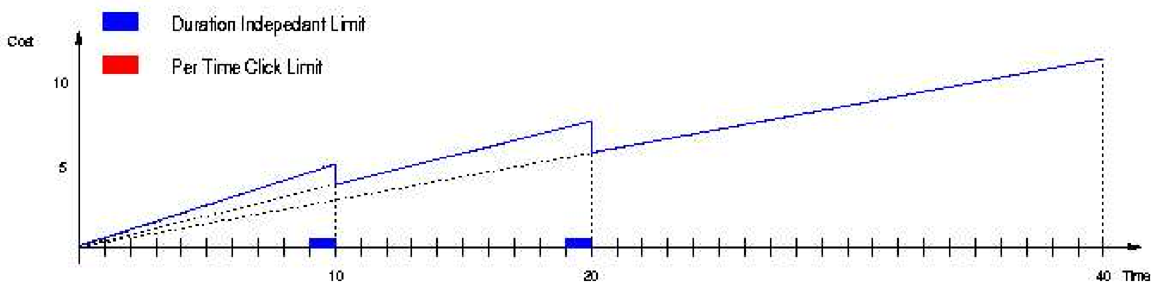


Figure 3.4.8.3: Effect of a duration independent limits on produces.

Example 3 - In order to ship the car, you need a truck and a driver (the driver comes with the truck). If a truck unavailable for any reason, you will have to wait until one is available; this will delay your shipping time but will not increase the cost. Because the consumption of trucks is based on a per_time_unit model, the limits is also based on it. So the limits will only affect the current time pulse (in this case, the current week). In a related note, you should be careful when modeling this type of limits, if you instead wrote the limits like this:

```
(spec_limits
   (label No-Truck)
   (agent Volvo)
   (from Truck)
   (to Ship-Car)
   (model per_time_unit)
   (quality_power 0.0 1.0 )
   (duration_power 1.0 1.0 )
   (cost_power 0.0 1.0)
)
```

You will produces the upper curve in Figure 3.4.8.4. If the method is overloaded during only one time pulse, the duration for this time pulse will be increased by 100%, while the cost will be the same. Essentially what happens is that by leaving the cost accumulation unaffected on a per time unit consumes, you will wind up accumulating cost for a longer period of time than predicted (because it accumulates per time unit) due to the duration being increased, resulting in a higher overall cost. This will produce a final cost of $550. Instead, you may change your limits this way:

```
(spec_limits
   (label No-Truck)
   (agent Volvo)
   (from Truck)
   (to Ship-Car)
   (model per_time_unit)
   (quality_power 0.0 1.0 )
   (duration_power 1.0 1.0 )
   (cost_power -1.0 1.0)
)
```

...which produces the lower curve by elimintating the cost accumulation created by the consumes during the time it is active. This limit will just delay the method by 100% (specifically, delay by one pulse, because the limits is per time unit), without increasing the overall cost.

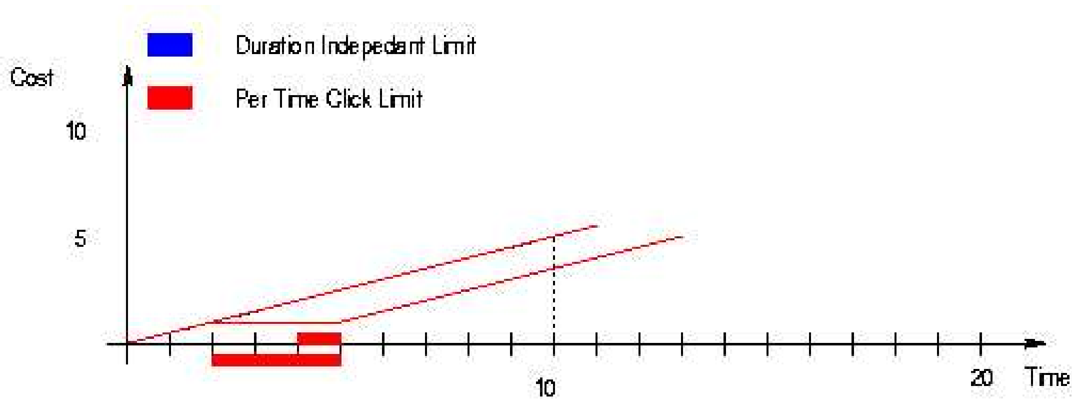


Figure 3.4.8.4: Effect of a per time unit limits on consumes.

Example 4 - Figure 3.4.8.5 shows the cumulative effects of not having a car available at the beginning, followed by a truck problem. The next shipping date increases to 23, and if at time 23 there is also no space at the dealer, the method will finish at 46 with only 25% of the original quality and a very unhappy customer.

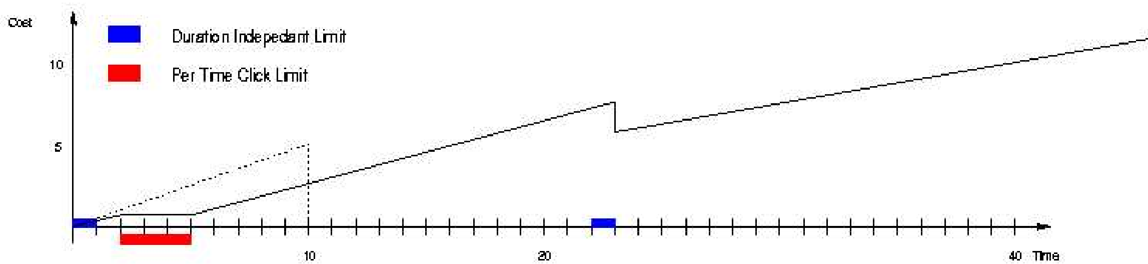


Figure 3.4.8.5: The effects of all three limits on a single method.

Agents

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Agent Symbol	no

Figure 3.5.1: Specification of spec_agent.

The agent objects in a Taems structure allows the modeler to make available the list of agents represented at a quick glance, and associate attributes with each agent in a central location. Multiple spec_agent objects may appear in your Taems structure, although by convention the first to appear is considered to be the "owner" of the structure (i.e. the one likely to use it for local scheduling purposes, if any). You should have a spec_agent object for each of the agents that appear in the structure (i.e. in the agent field of nodes), and it is required that at least the owner agent is given.

Agent objects have no graphical representation in Taems.

```
(spec_agent
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label Agent_A)
)
```

Figure 3.5.2: An example agent in TTaems.

Field Overview

spec_attribute
: See Elements.

label
: See Elements.

Examples

In this section we will look at several examples that should give you an idea how Taems can be used in practice. This serves two purposes then: to give you several concrete examples of Taems structures and the reasoning behind them, and also to show how some of the more complex nuances of the language can be used.

Example 1: Getting Dinner

In this example we'll be looking at how one might go about getting dinner for one's self, assuming everyone has done that on at least one occasion. Being the first example, we'll just look at the node structure in this one, and leave interrelationships and resources for another day. Just as a disclaimer I'll point out that this isn't going to be the Julia Child version of getting dinner, but a contrived example to show task-method relationships.

We'll start out with just a task group, aptly named "Get-Dinner". This will represent the fact that we have a highest level goal of getting dinner for ourselves, and it isn't (in our current view of the world) related to any other thing we happen to be doing. In this example, there will be three different independent ways of getting dinner - order in, cook it yourself, or eat out at a restaurant. With those decisions made, the top of our task structure will look like this:

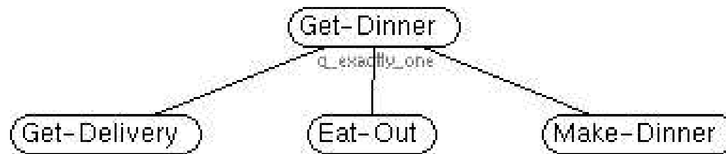


Figure 4.1.1: The top of the dinner tree.

We've used the QAF `q_exactly_one` in our task structure, to indicate the fact that only one dinner-getting technique should be used. One could argue that a `q_max` could also work here, but I suspect the semantics we're after are that we only eat one dinner this evening.

Moving on, we can now tackle the actions behind each of these subtasks. Delivery food is fairly straightforward - you just call the restaurant up and some time later the food appears. Eating out is somewhat more complicated, as you must get to the restaurant before you can place your order and it will cost more, but in return you get somewhat better food. Making your own dinner on the other hand is the cheapest alternative, and you get reasonable food, but it may also take a long time to prepare. These options give us the following structure:

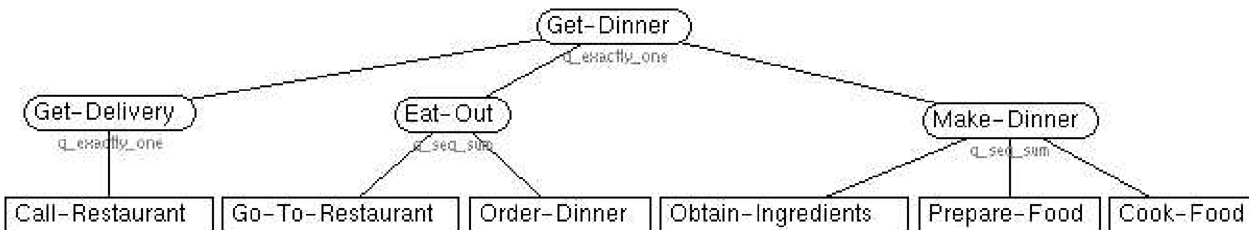


Figure 4.1.2: The completed the dinner tree. [src]

Here we have chosen `q_exactly_one` as the QAF for the delivery task, as we only need to call one restaurant. The other two tasks, however, require that several actions be performed in a certain sequence, therefore the `q_seq_sum` QAF was used. If you look at the source for the figure you can also see how our task characterizations were broken down into quality, cost and duration quantifications in the methods.

Example 2: Alternatives

One of the most interesting and useful aspects of the Taems modeling language is its ability to model alternatives. For instance, you may wish to describe the fact that one of several actions may be performed (as was seen in Example 1). You can also model the fact that a given method may have several possible outcomes, or that an agent has the option to perform a facilitating task before the actual one. The breadth of possibilities is exactly what makes Taems useful in intelligent systems - a quantitative representative of the agent's *choices* lets it select that which is most appropriate in the given context. This in turn allows the system to exhibit the actions that hopefully seem intelligently directed.

In these examples we will be looking at the "soft" parts of Taems - those which give the agent the option to perform them or not. "Hard" objects in Taems are quite useful as well (sequenced QAFs, enables, etc.), but in some sense they are not as interesting because they either **must** be satisfied to achieve quality, or their selection is driven by the presence of a soft object elsewhere in the structure.

The simplest form of alternatives involves tasks and their quality accumulation functions. Both `q_max` and `q_sum` give the model a wide range of flexibility, while `q_min` and `q_exactly_one` do as well to a lesser degree. Here's an example describing the process of cleaning a house:

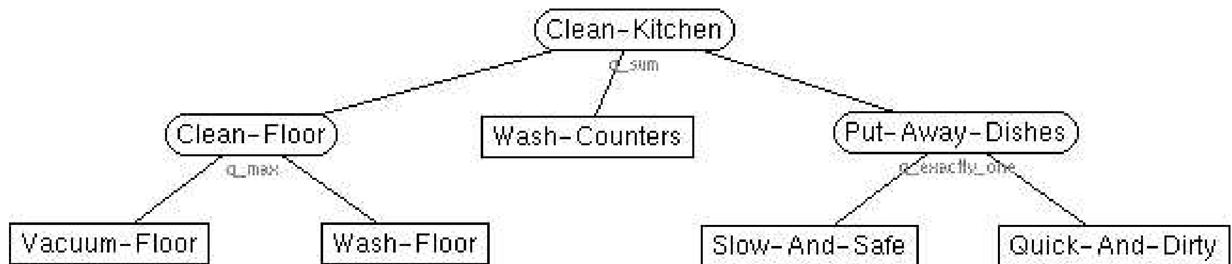


Figure 4.2.1: Cleaning the kitchen with QAFs. [src]

We use three different accumulation functions in the example above. In the Clean-Kitchen taskgroup, `q_sum` is used, which tells us that the more cleaning we are able to accomplish, the better. This is true because all the subtasks are distinct. The tasks under Clean-Floors, however, are not distinct, so a `q_max` QAF is used to model the situation where each task is capable of raising the total amount of quality, if it did a better job than all its predecessor tasks. There is a perceived maximum quality that can be obtained from cleaning the floors (they can clearly only get so clean), so that if we've already washed all the dirt off it won't help to vacuum it afterwards, for isntance.

Putting away the dishes is yet another situation. Again, these tasks are conceptually linked somehow, because once they've been put away, that's it - if you've broken some then you are out of luck, and you wouldn't really take them all out again to give it another go. Thus, a `q_exactly_one` is used to model the situation that only one subtask may be chosen for execution. Note that we use `q_exactly_one` here to tell the agent what we want it to do, not necessarily to exactly model the quality semantics of the task. There is no physical reason why performing both actions would make the supertask have a quality of zero, but because we don't want the agent to consider doing this, we associate a penalty with that situation anyway.

There are potentially other things in this example that can be modeled as well. If we vacuum the floor before washing it, for instance, the washing activity may have a higher quality. In addition, after we have washed the floor, it will be wet, which can interfere with other tasks to be performed in that room. These effects may be modeled with interrelationships in the following way:

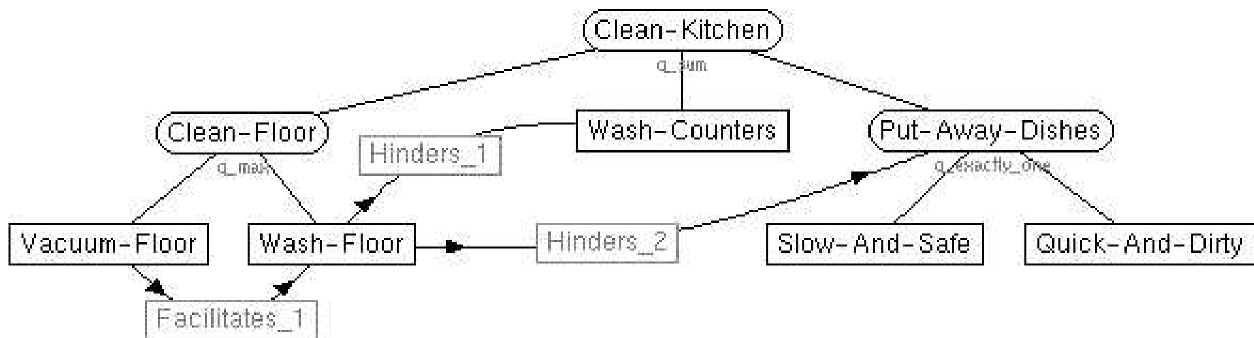


Figure 4.2.2: Cleaning the kitchen with interrelationships. [src]

The component which is generating the agent's schedule of activity now has more information available to it concerning the interactions between the methods. If it has a lot of time available to it, it can choose to vacuum before washing the floors, which will give a higher overall quality (at the expense of a longer duration because it must perform two actions). Analogously, it can also realize the negative effects that can occur if the floor is washed before cleaning the counters or putting away the dishes.

The last alternative type we'll look at are outcomes. The way it was modeled in the previous sections, the Quick-And-Dirty method will always produce less quality than Slow-And-Safe. This isn't really what we want to model however - instead we want to specify that there is some *probability* that putting away the dishes could go horribly wrong some percentage of the time if we play it fast and loose. There should still be a chance that it will all work out, however, otherwise our cleaning friend above would always be breaking plates. The answer to this is to give the Quick-And-Dirty method two (or more) outcomes:

```
(spec_method
   (label Quick-And-Dirty)
   (agent Me)
   (supertasks Put-Away-Dishes)
   (outcomes
      (Outcome_1
         (density 0.9)
         (quality_distribution 8.0 1.0)
         (duration_distribution 5 1.0)
         (cost_distribution 0 1.0)
      )
      (Outcome_2
         (density 0.1)
         (quality_distribution 0.0 1.0)
         (duration_distribution 5 1.0)
         (cost_distribution 0 1.0)
      )
   )
)
```

Figure 4.2.3: Using outcomes to better describe Quick-And-Dirty.

In this new object we have two possible outcomes. The first, `Outcome_1` gets the same quality as the Slow-And-Safe method, but it takes only 5 units of time (as opposed to 15). This outcome will occur 90% of the time. There is a 10% probability, however, that we'll smash up the dishes, resulting in a quality of 0. So, if we have lots of time, the agent might choose to do things Slow-And-Safe; if there is an extra set of dishes lying around it might make more sense go down the more adventurous Quick-And-Dirty path. Using outcomes, this model now more accurately describes the situation at hand, which gives the agent the information it needs to make the correct action selection given its current goals and environment.

Example 3: Non-Local Methods

Sooner or later, you will want to represent a method in taems that your agent knows about, but cannot perform itself. This is useful when you need to represent an activity that affects your local plans in one way or another, but your agent can't execute it itself and doesn't have direct control over who does it. Consider the following task structure:

Figure 4.3.1: A task structure with a non-local method. [src]

Agent_A (the owner of the structure) is making widgets, and can do everything but polish them. In the above structure, Polish-Widget is a non-local method. It looks just like a regular task structure, right? That's the point - the agent can reason about the structure as it normally does, and the semantics are the same, but tags on Polish-Widget indicate that it is actually non-local. Thus, if that method is selected for execution, the agent will know that it must coordinate in some manner (e.g. contract net) with whatever agent(s) can perform the action (Agent_B, in this case) to get that action done for it.

The outcome (quality, cost, and duration) estimates for the nonlocal method represent what Agent_A believes Agent_B's execution characteristics will be. In this example, they do not include any potential overhead, such as increasaed duration or cost, that might be incurred from the act of coordination. You may include this information in your model, although this information is very coordination and instance dependant, so it may be incorrect or misleading at runtime.

Here's what Agent_B's task structure might look like:

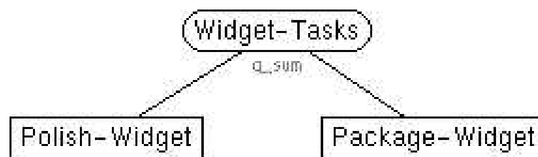


Figure 4.3.2: A task structure with the local version of the non-local method discussed above. [src]

Looking at the source, it it interesting to note that the real version of Polish-Widget done by Agent_B takes longer and produces an output with less quality than what Agent_A believes. Sounds like a little false advertising to me.

Figure 4.3.3: A objective view of the situation described above. [src]

Shown above is an objective model of the above situation would look like. In this global world view, the nonlocal method object does not exist. Instead, Polish-Widget, owned by Agent_B, is put directly under Fabricate-Widget, with the "true" characteristics shown in Figure 4.3.2.

Example 4: Non-Local Tasks

This example is much like the previous example, with one significant difference: here we want to have one agent represent as a nonlocal method what another agent represents as a task. This is a useful technique, as it allows an agent to encapsulate what might be a very complicated process as a single "black-box" method that it can reason about. For instance, you might know that your friend can Fix-Broken-Table, but have no idea what that process entails. To you, Fix-Broken-Table is a nonlocal method to which you can assign some rough description of what the execution characteristics are. To your friend, however, Fix-Broken-Table is a task (or even a task group), which can have numerous subtasks and interrelationships.

To clarify the difference between this process and the previous example, we'll look at a very similar problem. In this case, Agent_A still has the same task structure, with a nonlocal Polish-Widget method:

Figure 4.4.1: A task structure with a non-local method. [src]

On Agent_B's side, however, we have something that looks somewhat different:

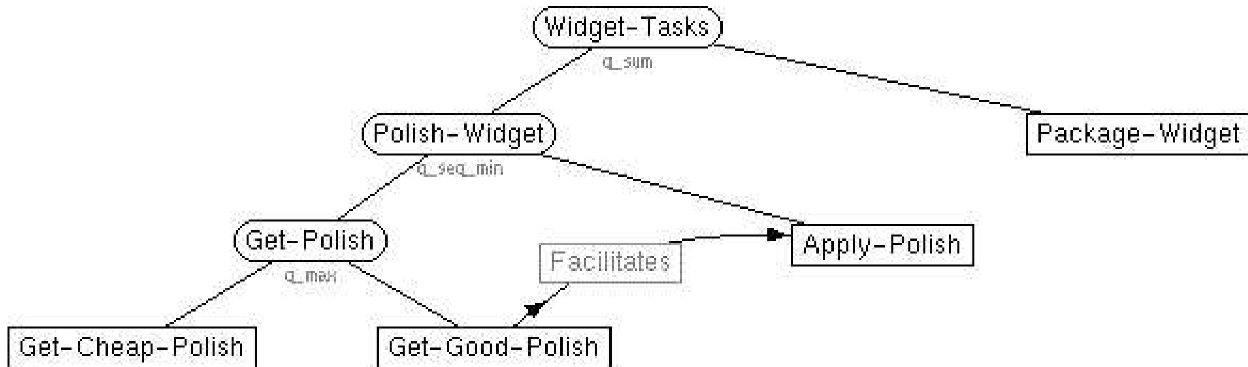


Figure 4.4.2: A more detailed version of Agent_B's activities. [src]

You can see that what Agent_A perceives to be a single method, Agent_B actually has represented as a task containing other tasks, methods and an interrelationship. Presumably, Agent_A has a rough approximation of how Polish-Tasks might be performed by Agent_B in the `outcomes` field of its nonlocal method, although it cannot capture all the semantics of what actually might take place. This is a useful abstraction, however, as it allows Agent_A to coordinate over the Polish-Widget activity without it having to know all those gory details. Of course, it is also quite possible to represent all the gory details locally, and perhaps the agent might coordinate slightly better because of it, but there will still be cases where simplicity wins out over optimality from the agents perspective.

Here's what the objective view might look like:

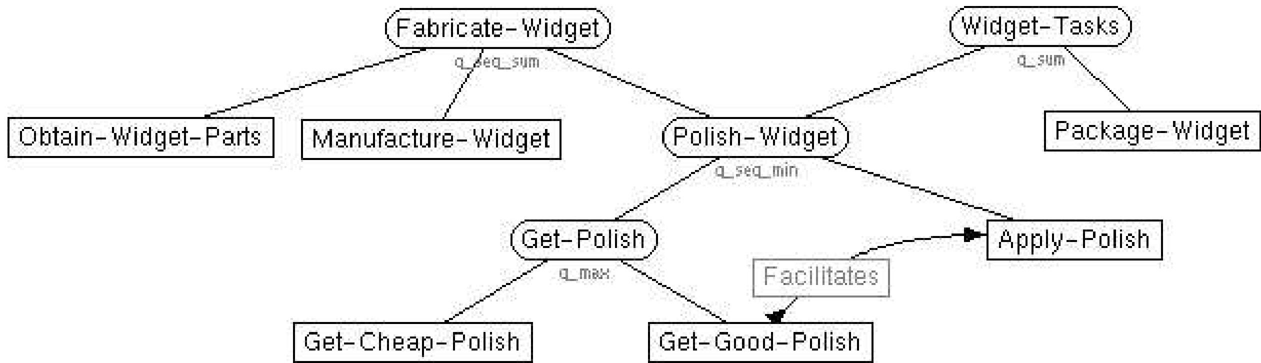


Figure 4.4.3: An objective world view of the above situation. [src]

Example 5: Inter-Agent Interrelationships

Given that Taems was designed to be used in a multi-agent system, you may frequently run into cases where you want to indicate that an interrelationship exists which spans two distinct task structures. Say for instance we have a younger sibling with the following task structure:

Figure 4.5.1: The sibling task structure. [src]

Looking at the source, you can see that the expected quality for the speaker method is higher than that of the headphone method, so all things being equal, the sibling would prefer to use speakers.

On the other hand, this is my task structure:

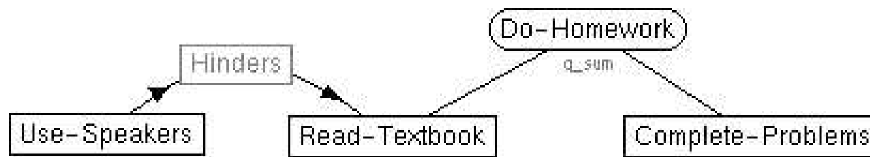


Figure 4.5.2: My task structure.

I've got some homework to do, some reading and some exercise problems to complete. Since music disturbs many people when they are trying to read, we have modeled the interaction between the sibling's speaker use and my textbook reading as a hinders. The Use-Speakers method is encoded as a nonlocal method, meaning in general that the method does not belong to the agent owning the task structure (me, in this case), and that incomplete or incorrect information about the method is relatively common. Here all we know is that Use-Speakers is owner by Sibling and that it hinders Read-Textbook. As far as agents are concerned, this information is generally sufficient to take some sort of action. In this case, I can negotiate with my sibling with knowledge of this interaction in hand - perhaps the sibling would decide to perform Use-Headphones instead, or we can coordinate our activities so Use-Speakers does not overlap with Read-Textbook (taking advantage of the fact that Complete-Problems is apparently not affected by speaker usage). The presence of this interrelationship, and the availability of information describing the situation, allows the agents to exhibit a wide range of behaviors to solve the problem.

In the source, you can see that Sibling owns the hinders interrelationship. We recommend that the agent owning the source node of the interrelationship be set as the owner of the interrelationship, although this isn't technically necessary. Note that some Taems implementations may force you to place all objects under some sort of task group node, in which case you should place Use-Speakers under a dummy task group (or bundle Use-Speakers and Do-Homework under a common task group).

It is also quite possible to model things differently, by storing the hinders in the sibling's view. In the first case, the semantics were that the sibling was operating without knowledge of their effects on my homework. If the interrelationship were present in the sibling's structure, however, the sibling would be cognizant of the effects Use-Speakers might have on my actions, and could choose to negotiate in a similar way, or alter their local actions in a more independent manner. It is also possible that both agents possess knowledge of the interrelationship, and in a real system one might expect that such information would be shared in such a way to produce a more complete local view for each agent.

Just to complete the picture, the objective view (actual state of the world) would be represented like this, regardless of the location of the hinders in the separate models above:

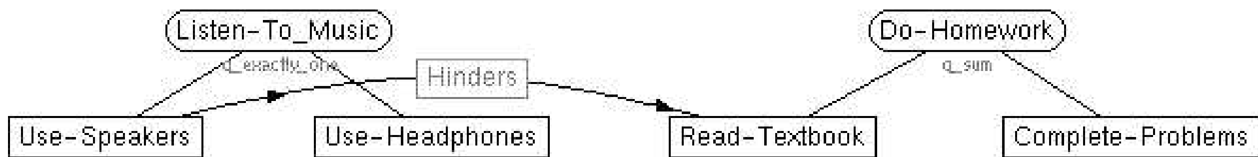


Figure 4.5.3: The objective, world view task structure. [src]

This is the type of view that is implied by the two separate structures above, and would be constructed if some sort of centralized model were needed (as is done by the MASS simulator, for instance).

Example 6: Making Coffee

In this example we'll look at a more sophisticate task structure, desribing how to make coffee. This task structure was adapted from one used by the CoffeeMaker agent from the Intelligent Home project. We'll briefly go over the some of the modeling decisions made in the figure.

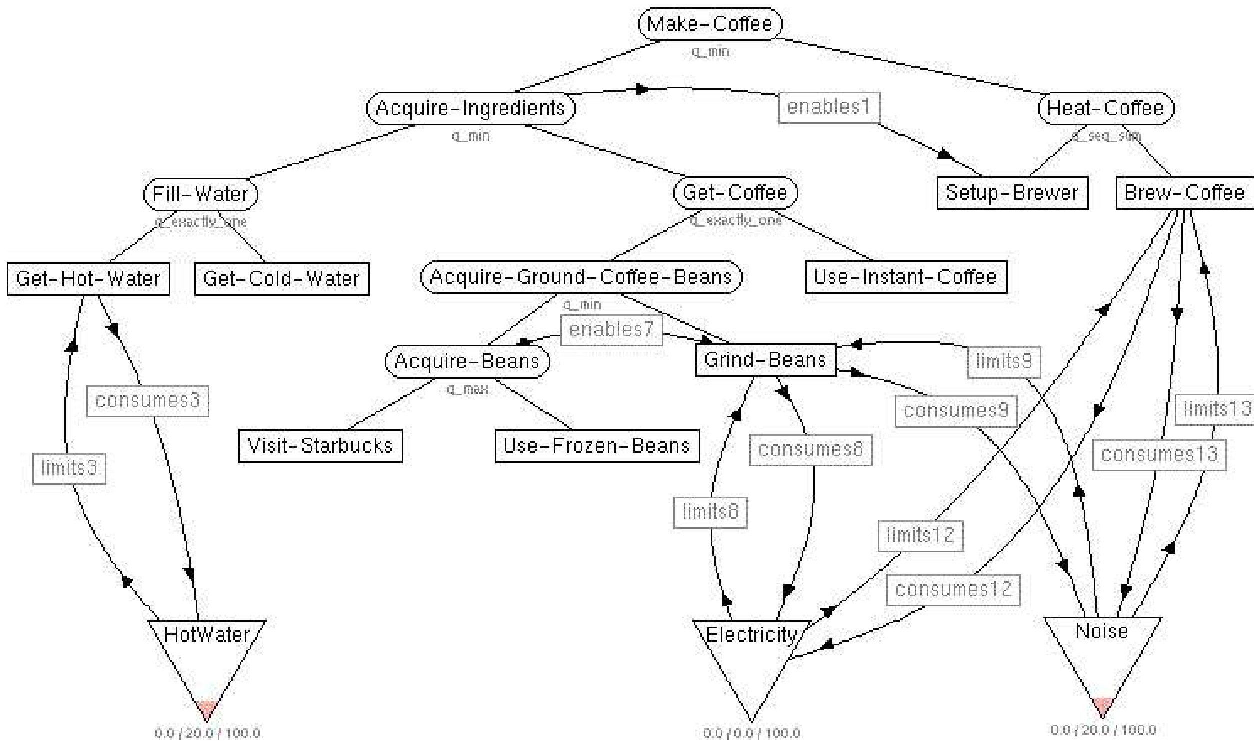


Figure 4.6.1: How to make coffee. [src]

Starting from the top, you can see that Make-Coffee has a `q_min` QAF, as does Acquire-Ingredients. This was done to specify that all the subtasks must be performed to accrue any quality (AND semantics). A `q_seq_min` could have been chosen for the task group's QAF, but we elected to go with an enables interrelationship (enables1) to indicate this constraint. Similarly, `q_seq_min` was used under Heat-Coffee, while a simple `q_min` is under Acquire-Ground-Coffee-Beans - the semantics on these actions are the same, so you should see how a sequenced QAF and hard interrelationship are frequently interchangable.

Below the main task structure are three resources, which are being consumed by various methods. Note the limits which exist to describe the negative effects of insufficient quantities of the resources.

Modeling

This section contains information about various modeling issues with Taems, including a troubleshooting/ rules-of-thumb page, and a discussion about Taems views.

Rules of Thumb

Often you will model something to your satisfaction in Taems, but when you go to use that in an implemented sense (e.g. scheduling, agent control interpretation, objective simulation, etc.), you get unexpected results. Many times these are caused by common errors or misunderstandings about the semantics of the different Taems objects. This page is meant to serve as a list, or FAQ, describing these situations, and how you can fix them.

1. Look up - See which QAFs are being used by your tasks, and what exact semantics are associated with them. Are the QAFs too hard (sequenced, min, exactly one) or too soft (sum, max)?
2. Look sideways - What interrelationships are attached to your tasks or methods? Is an enables not being satisfied, is a hinders being unexpectedly activated? Does your model describe an unsatisfiable situation because of unintended loops or mutually exclusive requirements? Are you taking into account the source's power factor when looking at facilitates or hinders?
3. Look down - Have you associated limits interrelationships with your consumes and produces? Did you update the actual state of a resource before giving it to your Taems processing tool? Is your resource type (consumable, non-consumable) what you want it to be? Do your consumes/ produces/ limits have the correct `model`?
4. Look elsewhere - If you are dealing with nonlocal effects, check the task structure belonging to the remote agent in question. Does it have the method you are concerned about? Is it annotating its task structure correctly (deadlines, commitments, etc.) when it is producing schedules?
5. Note that interrelationship effects for things like facilitates, hinders and limits are percentage based, so the greater the orginal value, the larger the actual change value will be. Other interrelationships, like consumes and produces, are absolute values so their effects will be constant regardless of the target's state (assuming no bounds violations). Enables and disables also act like this, they are essentially on-off switches that control their target's quality accumulation regardless of its state.
6. If a method isn't being scheduled or considered, look at the `earliest_start_time` and `deadline` fields to see if it is over-constrained in the current environment. Does the method have the potential accrue enough quality for the scheduler to pay attention to it? Are there hard commitments (or lack thereof) that might be interfering with the method being chosen?
7. Check to make sure the `agent` field on all your objects are what they should be. Do those on local methods match your agent's real name? Do those on nonlocal methods match the real name of the intended target?
8. Are you attempting (intentionally or not) to re-execute methods in the same task structure? This isn't generally supported. If you want to re-execute a method (or make it available for consideration by something like a scheduler), make sure its `finish_time` is set to -1. If you want the process to view the methods as completed, give it a positive `finish_time` and update the outcomes to be consistent with the observed results.

Views

An important and useful concept when working with Taems task structures is the notion of a *view*. A Taems view is a generated or derived task structure which is appropriate given the current needs, context or control frame. For instance, an agent may store its entire world description in one view, while it may use a trimmed down version when actually generating schedules. The view that is chosen to use is based on what best meets its needs at the time. If the agent needs to select a high level goal to achieve it may get this information from a high level task structure, but if the agent needs to generate a schedule of activities it may use a finely grained one that focuses on a specific task. An agent can have any number of views available to it, at any level of granularity that the designer feels is appropriate. There are three views however: *objective*, *subjective*, and *conditioned*, which are most common. These three will be discussed is this section.

When someone says that a particular description or comment is objective, they mean that it is devoid of bias; it is the observable truth. The objective Taems view, then, is the "true" description of the agent’s environment, both in terms of global structure and quantitative values. In a real agent system no single agent would have the objective view - under these circumstances the operating environment would be an instance of a theoretical, intangible objective view. In a simulated agent system, however, the objective view would be the tangible object used by the simulation controller to represent the environment. The objective view should be considered to accurately represent the real world, any view the agent possess is necessarily an approximation of it.

A subjective comment is one based on an individual’s opinion - it may be skewed in some way based on their particular interests or observations. A subjective view then, is one that is specific to an agent, and may (or may not) deviate from the objective view in important ways. The subjective view should contain what the agent believes to be true its capabilities and the world. In this light, we can see that the primary goal of learning systems like TLS is to adapt the agent’s subjective view so that it is as close to the true objective view as possible.

A conditioned view is generally an engineered variation of the subjective. In this view the agent represents those elements that it has determined are important and relevant to the current goal is it working towards. For instance, if the agent has determined that a particular method from the subjective view has consistently given low quality, it may choose to remove it from the objective view so the scheduler will not consider it. Similarly, if it has determined that it is not worthwhile to coordinate over a particular interrelationship, then that interrelationship may be removed to prevent the coordination component from acting upon it. The conditioned view itself may also be just a subsection of a much larger structure, perhaps a single tree or subtree from a larger graph containing many task groups.

To make this more concrete, we’ll look at an example, based in part the task structure explored in the Alternatives example. We’ll look first at an objective view:

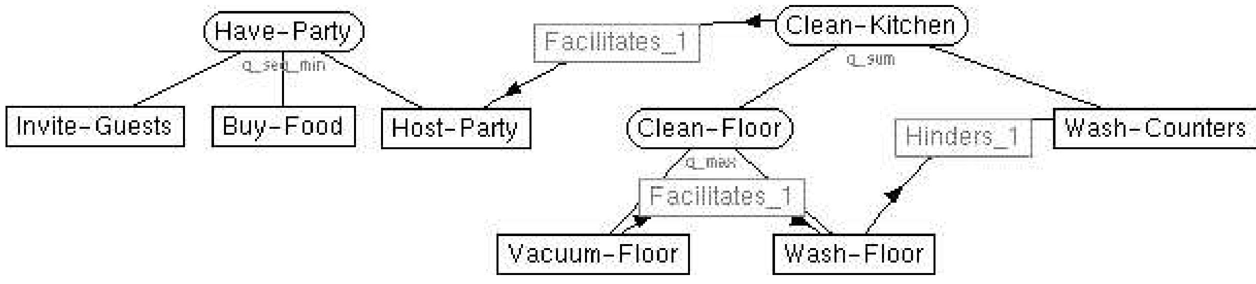


Figure 5.2.1: An objective view of a household. [src]

We are looking at the same task of washing the kitchen, but what was not shown in the example, however, is that there is a scheming sibling in the house who is planning on having a party when we leave for the evening. In our objective view we show that there is an interrelationship describing the fact that if the kitchen gets washed, it will help out with hosting the party. Clearly this is a good thing for the sibling, but there is no guarantee that we (as kitchen cleaners) would know about this fact.

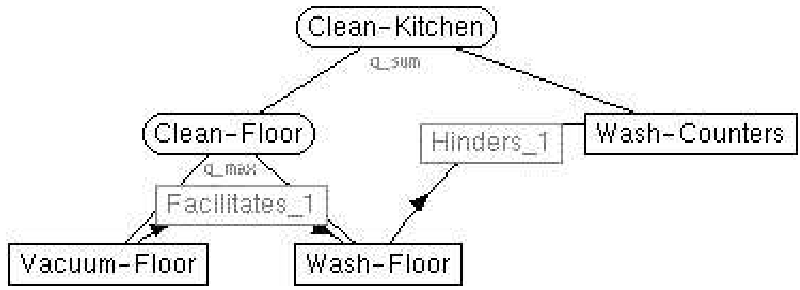


Figure 5.2.2: A subjective view of the kitchen cleaning task. [src]

In our subjective view, you can see that we don't know anything about our sibling's plans, including the facilitates interrelationship. So, in cleaning the kitchen, we will unwittingly be assisting the sibling's plans (unless they tell us, which is unlikely). If you compare the sources to the two task structures, you may also note that the Facilitates_1 interrelationship is not as useful as we think, and that Wash-Counters is going to take longer than expected. Such differences can exist between the objective and subjective as shown, and can also occur between two subjective views on different agents. These both provide for a more realistic working environment, and make for more interesting scenarios where issues of adaptability and survivability can come into play.

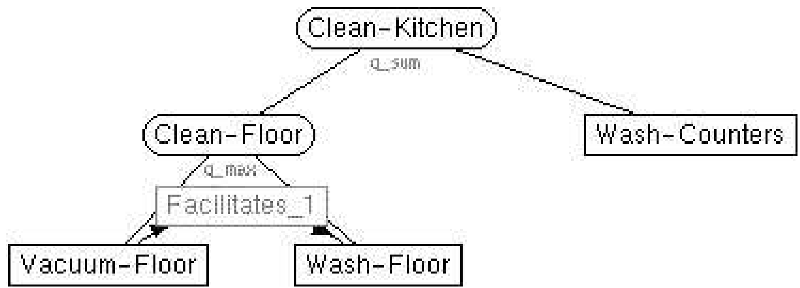


Figure 5.2.3: A conditioned view of the kitchen cleaning task. [src]

In the conditioned view we have removed another structural element - the Hinders_1 interrelationship. The semantics behind this say in a general sense that we are not interested in respecting this relationship - some control unit has determined that the effort required to work with the constraint is not worth the predicted benefits. Within the scope of this particular example, it might mean that we don't care if the floor is wet when we wash the counters.

Modeling Active Tasks

A Taems structure can be logically broken down into two major parts. The "upper" part, consisting of a hierarchy of tasks and task groups, represents the organization of the structure. These nodes form the organizing backbone of the "lower" part of the structure, which consists of a set of methods, which represent actual activities which may be performed in the environment. Together, the upper and lower aspects of the Taems structure form the tree or graph structure characteristic of any task decomposition representation. The remainder of the structure, including interrelationships, resources, QAFs and the like, provide ways to further describe and define the task tree.

In Taems, task groups and tasks serve a purely organizational role - they represent the concept of a goal or subgoal, which can be ultimately be broken down into simpler executable methods. There may be some instances, however, where one might wish to associate some sort of action with the task itself. Such a task will in some sense have an executable component, in addition to its primary organizational role - presumably it will work with or act upon the results of its subtasks. We will refer to these objects as *active tasks*. While this notion cannot be directly represented with a conventional Taems task structures, it is possible to emulate the semantics of such a construct using Taems in a relatively straightforward manner. The key insight is that an active task is essentially a hybrid node possessing the qualities of both a task and method, which one can model in Taems by separating out those qualities into a new pair of nodes.

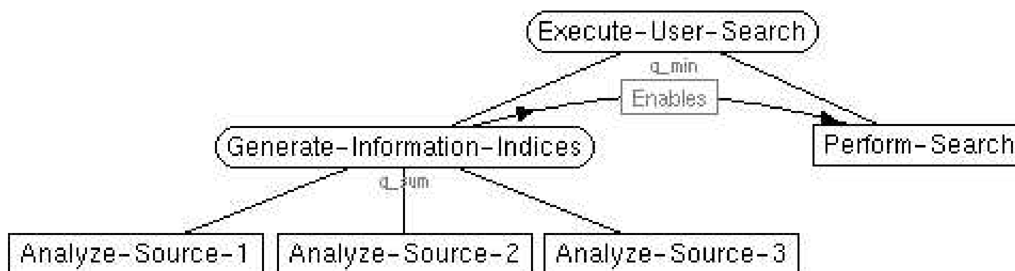


Figure 5.3.1: A prototypical task decomposition with an active task. This is not a Taems structure.

Consider the Figure 5.3.1, which contains the active task "Generate-Information-Indices" (thus, this is not a Taems task structure - it just looks like one). This represents the subgoal of generating a set of indices for an information retrieval task. The subtasks would first be performed, in this case analyzing three data sources. Following this, the task itself would be responsible for using the results of those analyses to generate the indices. Thus, one could envision that this task would be characterized with quality, duration and cost distributions just like a normal method.

To capture these same semantics in a Taems structure, one needs to separate the notion of task organization and method execution. This can be simply done by literally creating two new nodes to represent these needs. One node will be a pure task, organizing the three analyses methods. The other would be a method, representing the index generation activity.

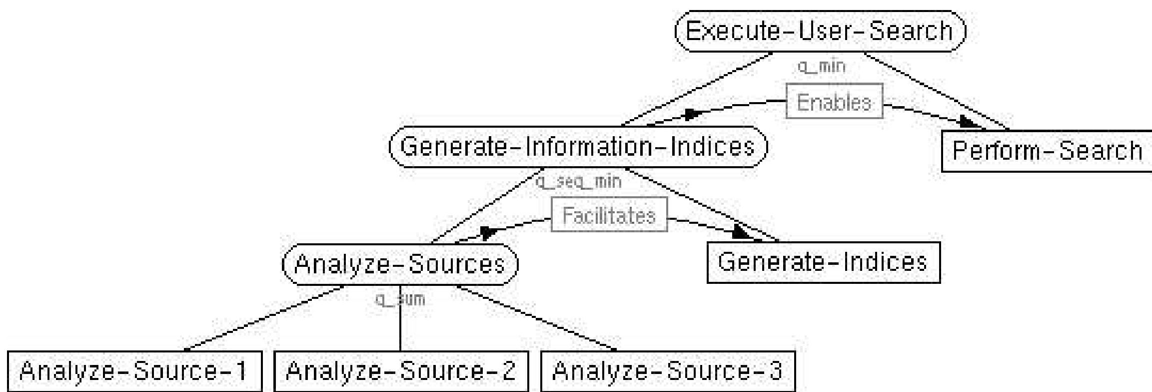


Figure 5.3.2: The structure from 5.3.1 as a Taems structure. [src]

In Figure 5.3.2 you can see how the original structure has been modified. Below Generate-Information-Indices are the two new nodes, "Analyze-Sources" and "Generate-Indices", which fill the roles (organizational and executional, respectively) outlined for the original active task above. The two nodes are joined under a `q_seq_min`, indicating that they must both be performed in order, which matches the original semantics. In addition to this constraint, a facilitates interrelationship has also been added, to reflect the notion that the quality of the generation process is still dependent on that of the analysis process.

Clearly it is plausible to have other characteristics associated with so-called active tasks - their actions might execute before or during their subtasks, their quality, cost and duration may be independent of their subtasks' results, and so on. Each of these traits will affect the details of how you choose to model the situation in Taems, but the core process of decomposing the task into its organizational and executional roles will remain the same.

Identifying Points of Coordination

Taems offers different ways to represent the need for coordination or negotiation between agents. We will explore different techniques here, in an effort to help direct developers towards the most appropriate solution. We will first cover how remote activities are represented, and then show how these serve as a basis for identifying instances where coordination or negotiation is needed.

Heterogenous `agent` Fields - All elements in a Taems task structure possess an `agent` field, used to identify which agent is, for instance, capable of performing the specified action, or responsible for a particular resource. They indicate that the specified agent should be contacted or coordinated with if one needs to interact in some way with the element in question. Suppose for example, that Agent_A is scheduling a particular task structure. By looking at the `agent` field of each scheduled method, it can determine if the action can be performed locally, or if it should be done by some other available agent. In the latter case, Agent_A would need to coordinate with the remote agent to ensure that the remote method is completed successfully in a manner consistent with Agent_A's needs.

In another situation, the `agent` field can be used to specify that a particular resource is *managed*. In this case, an agent requiring that resource at some time would need to coordinate with that managing agent over its use. Tasks with remote agent names could indicate that the remote agent is capable of performing the specified action in several ways. Subtasks could be used to locally differentiate the various possibilities, and direct coordination appropriately. Methods work similarly, except the only offer a single alternative, although one could use these as "abstractions", allowing one to encapsulate a more complex procedure with a single method name. In general, a certain amount of flexibility is allowed in how these are used in practice.

Coordination in this case is identified in how the nodes with remote agent fields are connected to the main structure. A QAF or interrelationship will typically cause interactions with remote nodes during scheduling, when they are selected to help accomplish local goals. Thus, the need for coordination can be detected by examining methods, tasks and resources used in or affected by the schedule which have different agent names.

Nonlocal Methods - These are quite similar to heterogenous agent fields, except they perhaps more explicitly represent the idea that the method should be performed by a remote agent. More so than a remote-agent node, a nonlocal node also represents some notion of abstraction. The local agent will know that some remote activity can be performed, but it won't necessarily know how. The QCD descriptors might describe an entire subtree at the remote host, although the task itself will be represented as a single method locally.

Nonlocal methods are also used to specify a potential point of expansion within the taems structure. The agent might know that some action should be done, but not necessarily know how. As it obtains more information, this nonlocal method could be expanded, for instance by splitting it into several separate methods, each identifying a different remote agent or mechanism which can perform that activity. Examples of non-local methods can be seen here.

Points of coordination detected in the same was as with heterogenous agent fields. The schedule is scanned for methods having the `nonlocal` flag, which will direct the agent that some form of coordination is needed to ensure the desired activity is performed.

Explicit Representation - Rather than representing a method or task which should be performed remotely, this type of structure explicitly specifies that some form of coordination should be employed. A method, such as "Coordinate-Over-Resource-Use", will appear in the task structure, and be scheduled just like any other. Just as the agent might locally know how to perform "Make-Beef-Stew", it will recognize methods of this form, and be able to perform the desired coordination activity. This has the important advantage of clarity, and also allows the designer to directly express the expected characteristics of the coordination - to directly reason about the potential results of coordination and monitor its progress over time.

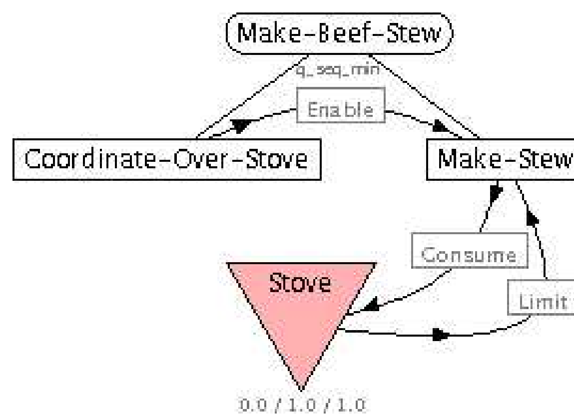


Figure 5.4.1: Explicitly representing coordination [src]

Figure 5.4.1 shows what an explicit representation would look like. In this case, the agent would locally perform Coordinate-Over-Stove to ensure that the stove was free. A component local to the agent would recognize (perhaps by the title), that it is responsible for this type of action. It would perform the action, and then update the task structure with the results of the coordination. If this process succeeds (indicating that the stove was reserved), Make-Stew can proceed as normal.

In practice, a hybrid of these schemes is used to indicate and perform coordination functions. Not all tools correctly interpret the agent and nonlocal fields, so a somewhat convoluted approach is taken to achieve our goals. Nonlocal methods are first used in the task structure to specify actions which are to be performed by other agents. When these are passed to the scheduling component, the nonlocal flag is stripped out, and these methods treated like any other. In the resulting schedule, the original task structure is used to identify which, if any, scheduled methods are nonlocal. If any are found, the problem solving component is responsible for generating an explicit coordination structure, which is scheduled and executed. This new structure can be generated either with domain knowledge, or by embedding appropriate hints in the attributes field of the original method. In either case, the new structure will define the type of coordination or negotiation needed, the remote agents to interact with, and the expected quality, cost and duration of the interaction. In this way, the coordination can be monitored just like any other method, and appropriate recovery mechanisms used in case of failure. Similar methods can be used to handle remote activities specified with heterogenous agent fields.

Translations

In this section we will show how the Taems modeling language can be translated into other formats. This is done both as an exercise in connecting our research with that of other groups and communities, and also to show how other problem domains can be viewed as Taems task structures (and vice versa).

MDP

In a perfect world scenario, an optimal meta-control policy would determine the best possible method to execute at each instant so as to achieve the desired high-level goal while optimizing the criteria and resource specifications. The drawback with such an approach is that for most reasonable size task structures the computational overhead of constructing this policy online is infeasible. However, we would like to see how well the DTC as well as contingency analysis approach perform in relation to optimal, assuming the policy is computed off-line.

Our approach for constructing an optimal policy is to define the problem as a finite-horizon Markov Decision Process(MDP) which tries to maximize its expected accumulated reward i.e. The MDP provides an optimal policy for achieving the high level goal given the criteria (quality, cost, duration) specification. It is a finite-horizon MDP because a primitive action can be executed only once in a particular execution path and hence there are no loops. MDPs are widely used in artificial intelligence as a framework for decision-theoretic planning and reinforcement learning.

As mentioned earlier, the Design-to-Criteria scheduling problem is framed in terms of a Taems task network, which imposes structure on the primitive actions and models the task interactions. The execution characteristics of primitive actions are modeled in terms of quality, cost and duration distributions. The following are some of the functional differences between the Taems framework and the MDP framework.

1. Taems does not represent the actual effects of the individual alternative paths. In other words, it does not carry through the implications of choices. The MDP framework, on the other hand, explicitly describes the primitive actions and their precise execution characteristics.
2. Taems specifies constraints on an ordering rather than explicitly representing the implications of the ordering. Consider, for instance, the primitive methods ***UserBenchMarks***, ***FindUserReviews***, ***ApplyNLP***and ***SearchAdobeURL*** in Figure 6.1.1. The only constraint on ordering specified by the Taems task structure is that execution of method ***FindUserReviews*** should precede that of ***ApplyNLP***. There is no requirement of immediate precedence and no constraint on immediate succession either. An MDP representation, on the other hand, would lay out exact precedence and succession orderings of methods within a path in the MDP tree.
3. Taems can be thought of as a compact representation of a class of MDP problems. It implicitly describes the enumerated search space which is explicitly described by the MDP.

The translation process of a Taems task structure to a MDP involves following a procedure which lays out each possible execution path for achieving the high level goal. We now describe the algorithm for expanding the compact Taems representation to an elaborate MDP representation is described.

The Translation

The MDP translation is a procedure which allows for the transformation of a Taems ask structure **T** to the corresponding MDP **M**. The state in the MDP representation is a vector which represents the methods that have been executed in order to reach that state along with their execution characteristics. The MDP action is the execution of a particular method. MDP actions have outcomes and each outcome is characterized by a 3-tuple consisting of discrete quality, cost and duration values obtained from the expected performance distribution of the MDP action. The rewards are computed only for the terminal states, i.e. the intermediate states have null rewards. The reward is computed by applying a complex criteria evaluation function of the quality, cost and duration values obtained by the terminal state. Value iteration is the dynamic programming algorithm used to compute the optimal policy. In theory, value iteration requires an infinite number of iterations to converge to the optimal policy. In practice, however we stop once the value function changes by only a small amount in a sweep. The following is the algorithm for the translation process.

Let TG be the top level goal in **T** and let METHODS be the set of primitive actions in **T**

1. Initialize MDP with state ***s***
2. Translate(s)
   1. Identify the set of actions(subset of METHODS) which are possible from ***s.***
   2. Iterate over each action
   3. If action is not TERMINATE
      - Expand each outcome(characterized by discrete quality, cost, duration values).
      - Determine if outcome can lead to a new state while adhering to the criteria constraints.
      - if new state s' is reached, Translate(s')
   4. Else if action is TERMINATE,
      - set reward of terminating state to be a function of the quality, cost and duration values of the state.
3. ValueIteration(StateSet);

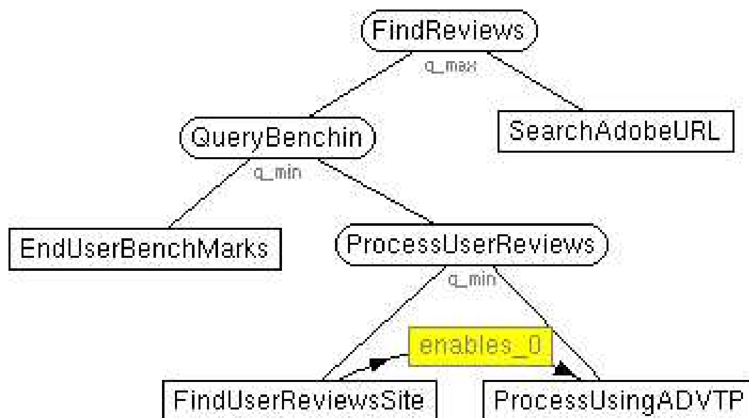


Figure 6.1.1: Taems task structure for review gathering example [src]

To make this discussion on the translation process more concrete, we will apply a few iterations of the algorithm on the example discussed in this paper. Upon translation, the corresponding MDP has 49 states. The input to the algorithm is the task structure in Figure 6.1.1 described in textual format. Figure 6.1.2 describes the translation process in progress. The start state ***SO*** is initialized. The PossibleActionSet for state ***SO*** is ***SearchAdobeURL***, ***UserBenchMarks***, ***FindUserReviews***. ***ApplyNLP*** is not a valid action since it has an inactive incoming enables from ***FindUserReviews.*** For each state in PossibleActionSet, we consider the outcomes of each of the action starting with ***SearchAdobeURL*** which has 4 outcomes resulting in the states ***S1***, ***S7***, ***S13***and ***S21*** respectively. The outcome resulting in state ***S1*** has a quality of 0.5, cost of 8.0 and duration of 3.0 and occurs with a frequency of 16\%. We now determine the PossibleActionSet for state ***S1***. The set is ***UserBenchMarks***, ***FindUserReviews***, ***Terminate***. ***UserBenchMarks*** has 3 possible outcomes resulting in the states ***S2***, ***S3*** and ***S4***. The PossibleActionSet for state ***S2*** is ***Terminate*** and ***FindUserReviews***. We exit from the current loop when a ***Terminate*** action is encountered and also exit from current loop if a deadline is crossed as is the case for both outcomes of ***FindUserReviews***

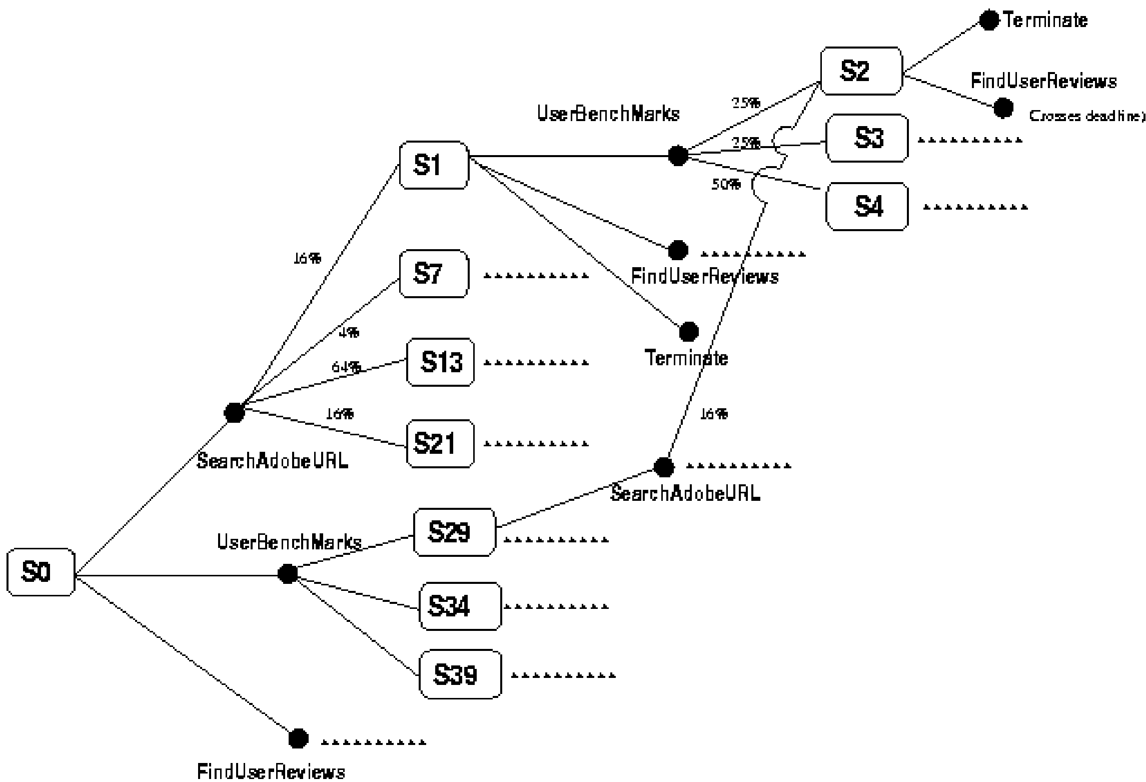


Figure 6.1.2: Partial translation to MDP of review gathering task structure

The optimal policy for the above problem is shown in Figure 6.1.3 As we can see the policy suggests the method sequence **{*FindUserReviews*, *UserBenchMarks*, ApplyNLP}** as the best schedule when method ***FindUserReviews***achieves non-zero quality and **{*FindUserReviews*, *SearchAdobeURL*}**would be an alternate schedule in the event of ***FindUserReviews***’s failure to achieve quality.

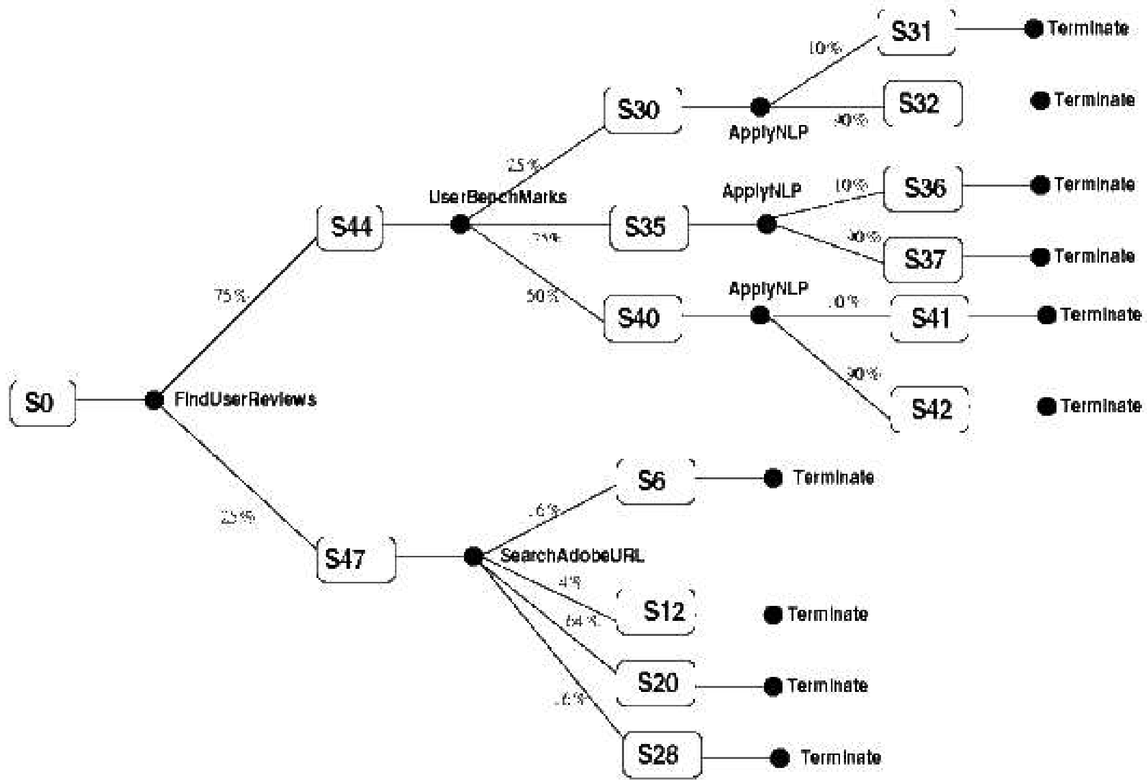


Figure 6.1.3: Optimal Policy review gathering task structure

Little-JIL

Little-JIL is a process-programming language that is used to describe software development process and other processes. Little-JIL represents processes as compositions of steps, which may be divided into substeps. The specification of a step is defined in terms of a number of elements. Each element defines a specific aspect of step semantics, such as data, control, resource usage, or consistency requirements. Briefly, the elements of a step specification are as follows:

1. **Resources** Specifications of resources needed by the step, including agent, software hardware.
2. **Steps** Identification of the substeps of a step (which are themselves steps).
3. **Step execution constraints** Restrictions on relative execution order of substeps.
4. **Handlers** Identification of handlers for local exceptions. Handlers can invoke substeps, thus exception handlers can use the full power of JIL to recover from errors.

Little-JIL language provides a rich framework for describing a process, including the control flow, data flow, exception handlers. The execution of the process is controlled by the dataflow, state variables, precondition and postcondition of each step. Now what we propose to do is to obtain a global view of this process and make a DTC schedule to accommodate specified objective functions. Since the process is complex and difficult to predict every detail before it is actually executed, it is impossible and unnecessary to make an exhaustive schedule about every detail at the beginning. We need to schedule at some certain level of detail but still leave opportunism for the JIL interpretor to react to the environment state. This means we do not perform an complete translation from the Little-JIL process program to the TAEMS task structure, but extract enough information from the Little-JIL process program to construct a TAEMS task structure.

Trip Planner Example

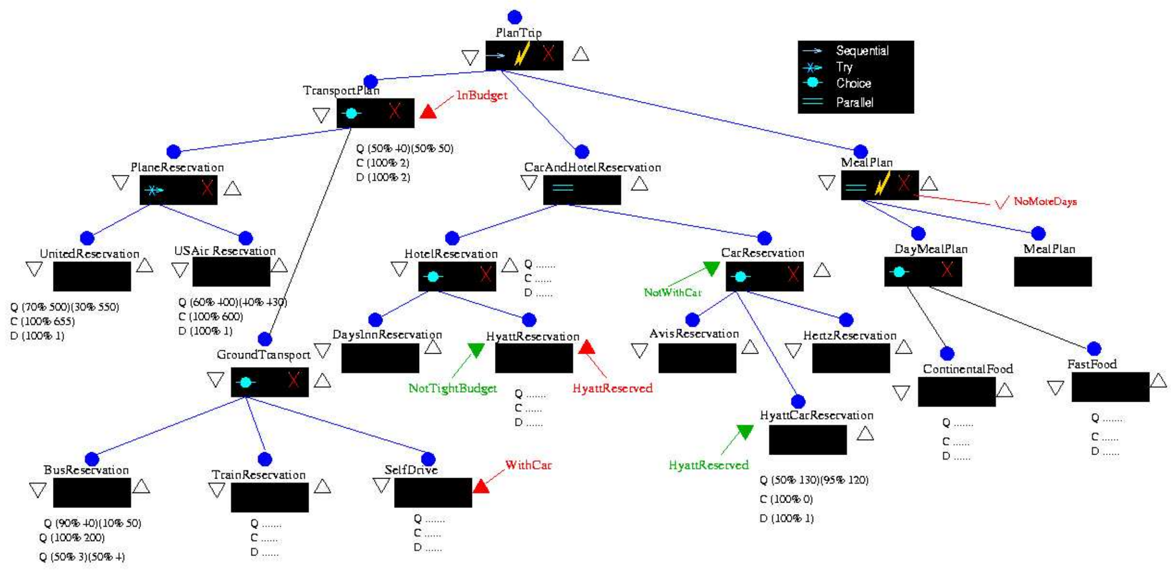


Figure 1 . PlanTrip Control Flow

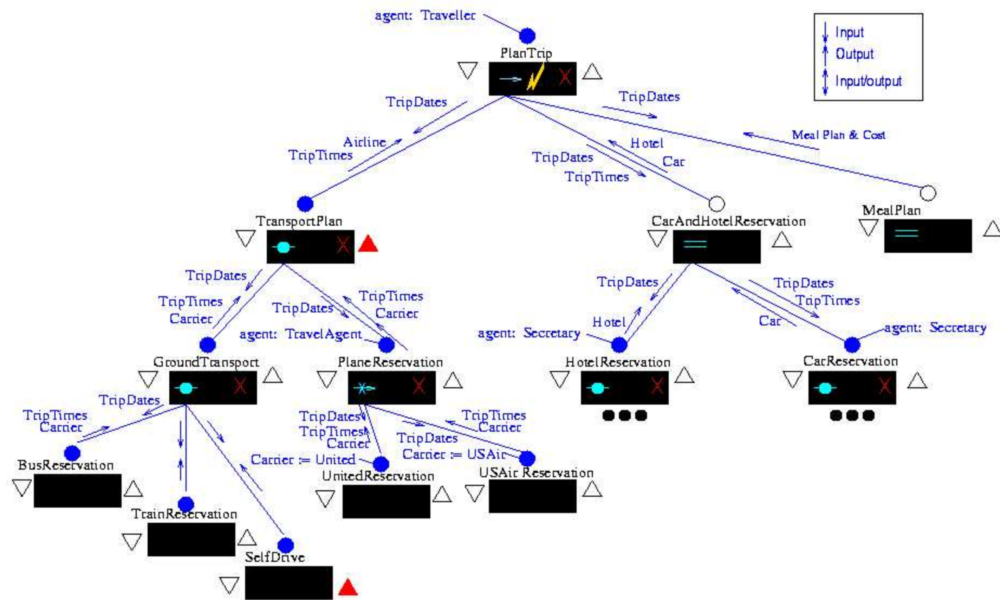


Figure 2. PlanTrip - Data Flow

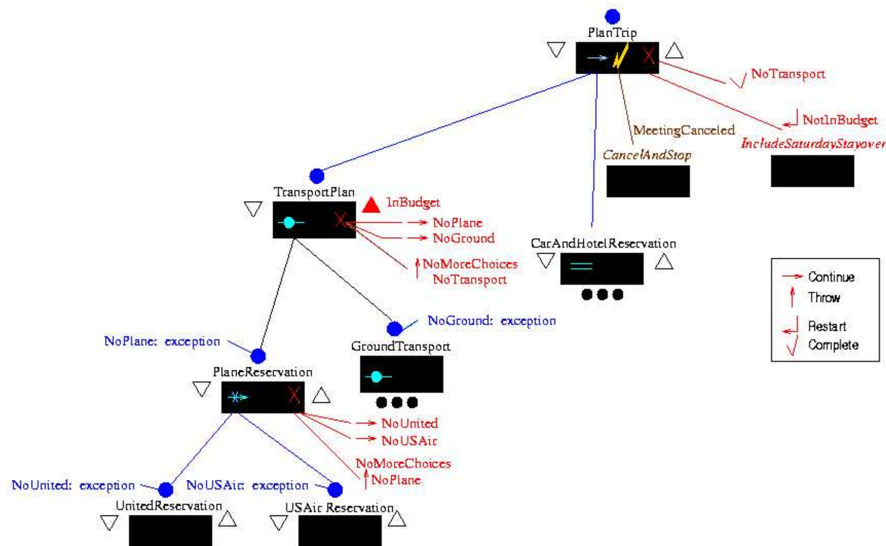


Figure 3. PlanTrip - Exception Handlers

Figure 1 shows the control flow of the ‘‘PlanTrip’’ step and Figure 2 indicates the dataflow of this step. Figure 3 shows the exception handlers for this process.

Unwinding Phrase

The agent receives the ‘‘PlanTrip’’ step from the Little-JIL editor. First, the task assessor unwinds this step and finds there are three substeps in this step, which are the ‘‘TransportPlan’’, ‘‘CarAndHotelReservation’’ and ‘‘MealPlan’’ steps. The task assessor unwinds these steps one by one. Unwinding process performs opening a JIL step, collecting necessary information and preparing for the translation phase. Major
function of the unwinding process includes:

1. Discovery of Non-Local relationships between steps caused by precondition, postcondition or dataflows. For example,there is no need to reserve a car if the traveler choose to drive his own car, so, there is a ‘‘disable’’ relationship from the ‘‘SelfDrive’’ step to the ‘‘CarReservation’’ step. Figure 5 shows two examples of discovering NLEs.
2. Identify reference steps and obtain abstraction expression by partial evaluation. Figure 6 shows an example of opening a reference step.
3. Identify non-local tasks and obtain virtual task expressions for them; Obtain an initial resource binding for non-local steps.
4. Analyze exception handler and decide if it should be opened; if so, discover and record all related relationships for this handler; Figure 7 shows some examples of opening exception thrower and handlers.
5. Control the depth of TMS task structure tree.

Unwind(JILStep Input_Step) :

1. Check precondition ‘‘A’’: Search the array of postconditions, if find step ‘‘Another_Step’’ has the postcondition ‘‘A’’, then record the ‘‘enables’’ relationship from ‘‘Another_Step’’ to ‘‘Input_Step’’; if find step ‘‘Another_Step’’ has the postcondition ‘‘Not_A’’, then record the ‘‘disables’’ relationship from ‘‘Another_Step’’ to ‘‘Input_Step’’.

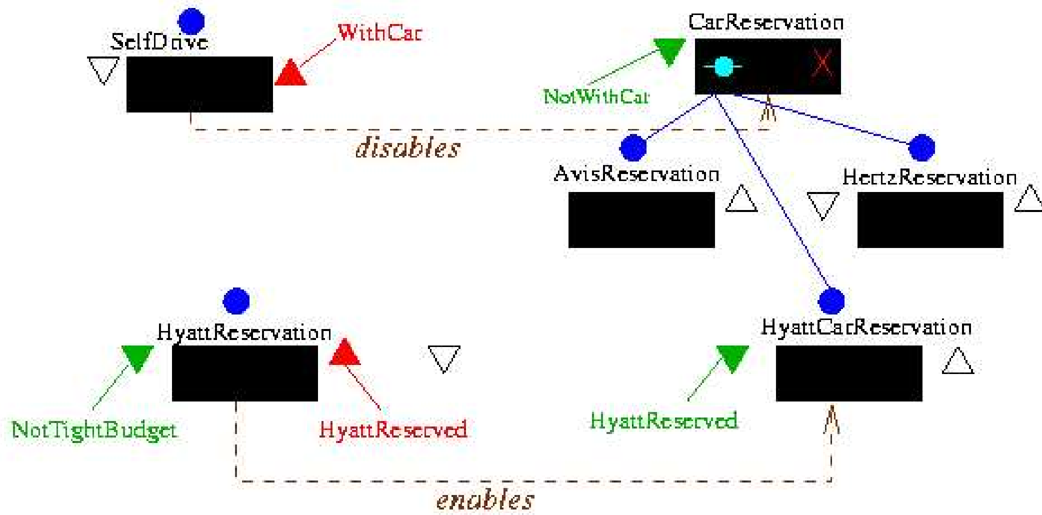


Figure 4. unwind NLE relationship

2. Check postcondition ‘‘B’’: Search the array of preconditions, if find step ‘‘Another_Step’’ has the precondition ‘‘B’’, then record the ‘‘enables’’ relationship from ‘‘Input_Step’’ to ‘‘Another_Step’’; if find step ‘‘Another_Step’’ has the precondition ‘‘Not_B’’, then record the ‘‘disables’’ relationship from ‘‘Input_Step’’ to ‘‘Another_Step’’.
3. Record precondition ‘‘A’’ in the array of preconditions.
4. Record postcondition ‘‘B’’ in the array of postconditions.
   Check the substeps of ‘‘Input_Step’’, if one of its substeps is a reference of ‘‘Input_Step’’, put a ‘‘loop’’ flag on this step, open this step using available data and send this step to the ‘‘partial evaluation’’ component.

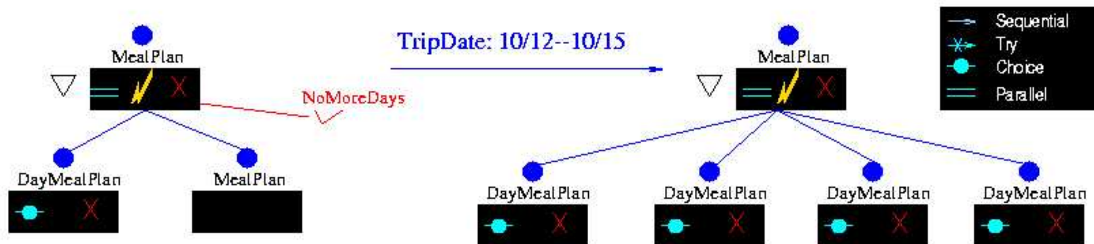


Figure 5. open inference step

5. Analyze every exception handlers: if the possibility of this exception happens is large enough, mark this exception handler ‘‘open’’. Discover all non-local relationships related to this exception handler. For an exception hander that is to be opened, the corresponding exception thrower should be opened

also. If the thrower is a leaf node, the throwing of the exception is represented as one outcome of the thrower, otherwise it is represented as a substep of the thrower.

1. Continuation handers: If the continuation handler is on a try node, an enable edge is created from the thrower to the next try step. If there is no remaining try step, this enable edge goes to the ''NoMoreChoices'' exception. On choice steps, a continue handler should add the possibility of throwing a NoMoreChoices exception.
2. Completion handlers: There are disable edges from the thrower of the exception to the other children and siblings of the handler. For try and choice steps, no disable edges are needed.
3. Restart handlers: There are disable edges from the thrower of the exception to the other children and siblings of the handler.
4. Rethrow handlers: There are disable edges from the thrower of the exception to the other children and siblings of the handler.

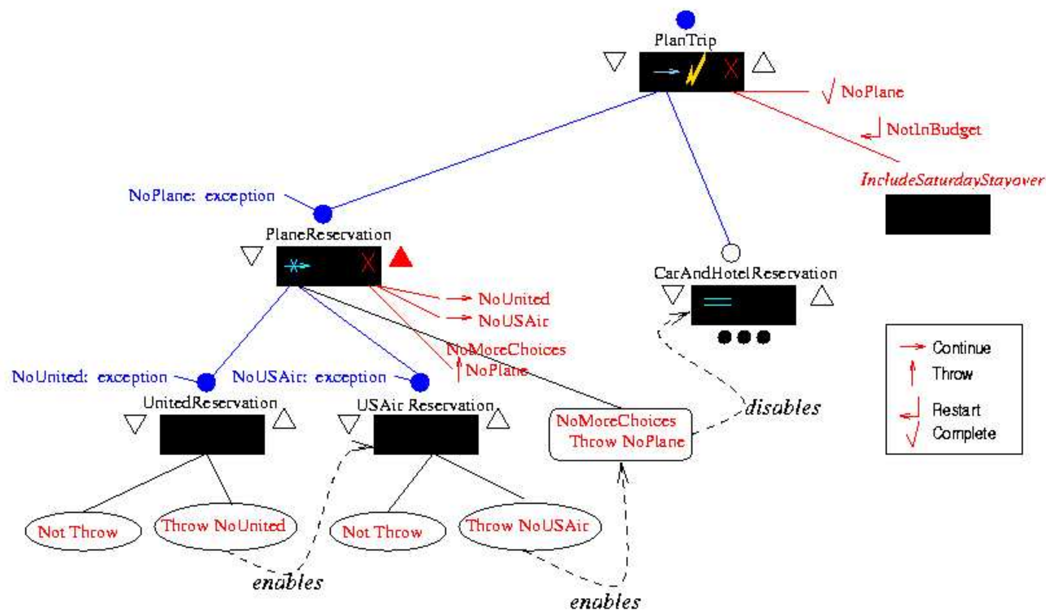


Figure 6. Exception Handler

6. Check the agent resource of this step, if the agent is not ''Local_Agent'', mark this step as ''non-open''. Also, send request to resource manager to reserve agent resource for this step.
7. Unwind every substep.
8. Unwind every exception handler marked as ''open''.
9. Return.

It sends this Little-JIL program with special flags to the translation components.

Translation Algorithm

For an input JIL step ‘‘InputStep’’, the translation component translates it into a TMS task structure. Figure 8 shows the basic idea of translation.

<u>**Translate(JILStep Input_Step, T MS Node Output_Node)**</u> :

1. If this step marked as ‘‘non-open’’, build a virtual task node using the same name of the step and return. More details about this virtual task will be provided by the abstraction component.
2. If this step is a leaf step, build a method node for this step, extract the quality, duration, and cost information from the step. Check the NLE record, if there is any step that has NLE relationship with this step. If so, build the NLE relationship between the corresponding task/method pair. Return.
3. If this step is marked as ‘‘loop’’, use the result from ‘‘partial evaluation’’ to replace this translation of this step. Return.
4. Build a task node ‘‘Output_Node’’ for this step, check the NLE record and build corresponding NLE relationships for this task ‘‘Output_Node’’.The quality accumulation function (qaf) for ‘‘Output_Node’’ is ‘‘sum’’.

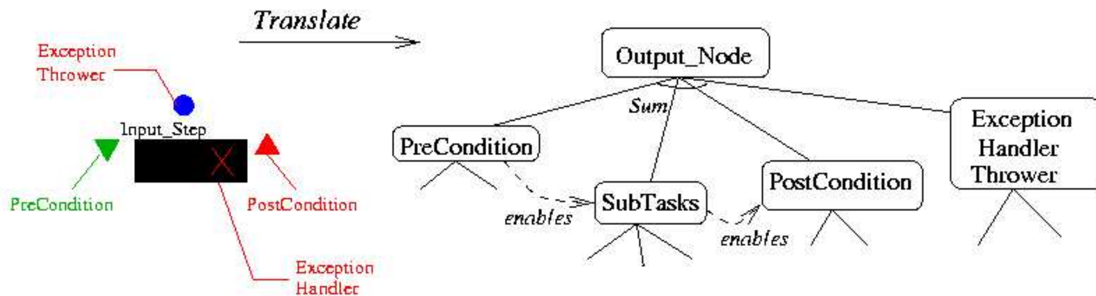


Figure 7. Translate Action

5. If precondition step exists, translate the precondition step into a task node ‘‘PreCondition’’ as one subtask of ‘‘Output_Node’’.Add an ‘‘enables’’ relationship from ‘‘PreCondition’’ to ‘‘SubTasks’’.
6. Build a task node ‘‘SubTasks’’ as the parent task of all substeps, each substep is translated into one subtask of ‘‘SubTasks’’.
   1. ‘‘Input_Step’’ as ‘‘sequential’’ step, set the qaf of ‘‘SubTasks’’ as ‘‘seq_sum’’.
   2. ‘‘Input_Step’’ as ‘‘parallel’’ step, set the qaf of ‘‘SubTasks’’ as ‘‘sum_all’’.
   3. ‘‘Input_Step’’ as ‘‘choice’’ step, set the qaf of ‘‘SubTasks’’ as ‘‘max’’.
   4. ‘‘Input_Step’’ as ‘‘try’’ step, set the qaf of ‘‘SubTasks’’ as ‘‘seq’’.Build outcome nodes ‘‘Success’’ and ‘‘Failure’’ for the first substep, add an ‘‘enables’’ relationship from the ‘‘Failure’’ outcome node of the first substep to the second substep.
7. If postcondition step exists, translate the postcondition step into a task node ‘‘PostCondition’’ as one subtask of ‘‘Output_Node’’. Add an ‘‘enables’’ relationship from ‘‘SubTasks’’ to ‘‘PostCondition’’.
8. For every exception handler marked with ‘‘open’’, translate this handler step into a task node ‘‘Exception_Handler’’, which is a subtask of ‘‘Output_Node’’. Check the NLE record and build necessary NLE relationships for ‘‘Exception_Handler’’.

At last get a T MS task structure described in Figure 9.

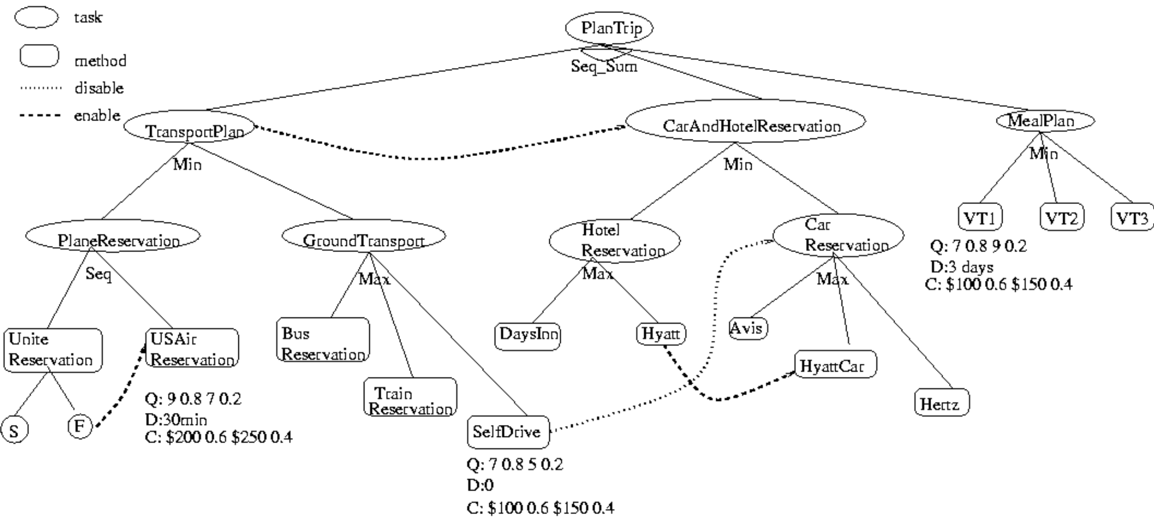


Figure 9. triptask.epsTrip Plan Task Structure

Pre-Taems

Pre-Taems (or PTaems) can be thought of as a meta-organization for textual Taems descriptions. It is designed to allow the developer to generate new Taems structures at runtime, which can vary both probabalistically and based on local or environmental characteristics. A Pre-Taems description is essentially marked-up textual Taems. These mark-up tags allow the developer to describe how a Taems object should be constructed, given a certain set of input parameters. Different blocks of TTaems data will be surrounded by, and contain, PTaems tags, which define both when a particular chunk of TTaems is to be used and where it should be inserted. A loose analogy can be made with the preprocessor phase of C/C++ compilers, where PTaems defines which bits of data should be used (and where), and the resulting unified block of data will be parsed by another utility - in this case a standard Taems parser.

In general, you will use your PTaems file like an interactive Taems file. Instead of just reading from it, you'll pass information into the parser that processes it, which will then be interpreted and (hopefully) acted upon in some way by the script itself to generate an appropriate Taems structure. In the following sections we will cover the syntax used in PTaems, and walk through some examples of how it can be used in practice.

Pre-Taems Syntax

A Pre-Taems, or PTaems, description has essentially the same syntax as a regular textual Taems description, along with a set of markup tags. These markup tags essentially serve three functions - to store data, control the construction of a Taems object, and to perform simple data generation.

In the first section, we will look at how PTaems can be used to store data internally, which provides the ability to use local variables to store statically or dynamically generated values that can be used elsewhere in the code. Following this, we will see how if/then type control structures may be used to conditionally control which Taems data is actually generated. In the third section we will cover the set of built-in parameterized functions that can be used to perform common generative and modification operations dynamically during PTaems processing.

Variables

Type	Syntax
In-line single variable assignment	`#define VAR = value`
In-line multiple variable assignment	`#define VAR = "value value ..."`

Multi-line variable assignment	`#define VAR`
	`value value ...`
	`value value ...`
	`value value ...`
	`#enddefine`

Static variable assignment	`#definenow` in above.

Figure 7.1.1.1: Usage of variable definition commands.

Lets look at the first internal capability of PTaems - data storage. This gives us essentially the same functionality as variable assignment is a normal high-level language. Within a PTaems description, the designer can assign an arbitrary chunk of data to a symbol, and then use that symbol in places within the description where that data would originally have gone. Assignment is done with the `#define` command, as in this example:

```
#define FOOBAR = 10.0
```

The above statement simply assigns the value 10.0 to the symbol `FOOBAR`. We can then use the symbol `$FOOBAR` later to use the original value, as in this method description:

```
(Some_Outcome
   (density 8.0)
      (quality_distribution $FOOBAR 1.0)
      (duration_distribution 5.0 0.3 6.0 0.7)
      (cost_distribution $FOOBAR 1.0)
   )
)
```

Later, when the PTaems interpreter runs through this code, the above statement would be expanded to produce this final output:

```
(Some_Outcome
   (density 8.0)
      (quality_distribution 10.0 1.0)
      (duration_distribution 5.0 0.3 6.0 0.7)
      (cost_distribution 10.0 1.0)
   )
)
```

You can store arbitrary data in variables, but multiple tokens (anything separated by whitespace) must be enclosed by quotes. So if we wanted to keep the entire `10.0 1.0` tuple in FOOBAR, it would work as follows, and still produce the same output.

```
#define FOOBAR = "10.0 1.0"

(Some_Outcome
   (density 8.0)
      (quality_distribution $FOOBAR)
      (duration_distribution 5.0 0.3 6.0 0.7)
      (cost_distribution $FOOBAR)
   )
)
```

The above example sketched out how to use *in-line* variables - those that are used to store data that can appear within a single line of output. In order to store more complex data that appears on multiple lines, you must use the *multi-line* `#define` syntax. Note that multi-line variables do not need to be enclosed by quotes; instead they use a trailing `#enddefine` to mark the termination of the block. Here's an example:

```
#define DEFAULT_DQ
   (density 0.5)
   (quality_distribution 10.0 1.0)
#enddefine

(Some_Outcome_1
   $DEFAULT_DQ
   (duration_distribution 5.0 1.0)
   (cost_distribution 2 1.0)
)
(Some_Outcome_2
   $DEFAULT_DQ
   (duration_distribution 10.0 1.0)
   (cost_distribution 6 1.0)
)
```

In this example, we've stored our default density and quality data in the `DEFAULT_DQ` variable, since we know both our outcomes are equally probable and return the same final quality. This saves time writing up the individual outcomes, reduces the possibility of typos in the code, and make it quite clear that it is the duration and cost that differentiate the two. If you were interested more in the structure than the performance of your task structure, you could even code up a single outcome definition with the above technique and just use the same one throughout your code.

You can reference other variables in your variable assignments as well, so the `DEFAULT_DQ` variable definition could contain a reference to the `FOOBAR` variable we were using earlier. Forward references are not permitted.

A useful capability of current PTaems parsers is the ability to pass variables into the process, much as one would pass a parameter into a function. These parameters are treated exactly as if they had been defined locally at the beginning of the script - they can be used and overridden in the file just like the variables described above. Within an agent, this means that such things as the agent's name, the current time, desired goals or the names of organizationally related agents can all be passed into and used by the PTaems script. The exact list of items which are passed in is architecture or agent-dependent, but for the

sake of the examples in this section we will assume the agent's name and the current time are passed in in the AGENT and TIME variables, respectively.

Armed with this overall description, we'll now code up our final method specification:

```
#define AGENT = Gomer  ; Implicit, passed in as parameter

#define FOOBAR = "10.0 1.0"

#define DIST = distribution

#define DEFAULT_DQ
   (density 0.5)
   (quality_$DIST $FOOBAR)
#enddefine

(spec_method
   (label Some_Method)
   (agent $AGENT)
   (supertasks Some_Task)
   (outcomes
      (Some_Outcome_1
         $DEFAULT_DQ
         (duration_$DIST 5.0 1.0)
         (cost_$DIST 2 1.0)
      )
      (Some_Outcome_2
         $DEFAULT_DQ
         (duration_$DIST $FOOBAR)
         (cost_$DIST 6 1.0)
      )
   )
)
```

When expanded, this should produce the following Taems object:

```
(spec_method
   (label Some_Method)
   (agent Gomer)
   (supertasks Some_Task)
   (outcomes
      (Some_Outcome_1
         (density 0.5)
         (quality_distribution 10.0 1.0)
         (duration_distribution 5.0 1.0)
         (cost_distribution 2 1.0)
      )
      (Some_Outcome_2
         (density 0.5)
         (quality_distribution 10.0 1.0)
         (duration_distribution 10.0 1.0)
         (cost_distribution 6 1.0)
      )
   )
)
```

An interesting characteristic of the above uses of `define` is that they are all dynamically bound. This means that if you set a variable X to equal variable Y, and then Y changes at some later point, X's value will also be updated. Here's a little example:

```
#define DUM = "Billy"
#define AGENT = $DUM
#define DUM = "Joe"

(spec_agent
  (label $AGENT)
)
```

When processed, the final name in the agent object would actually be `Joe`, which may perhaps be counterintuitive. In many cases, this is the behavior you may desire - however there are valid points where this would not be desired. The alternate behavior is to make static variable assignments - in these cases the value specified by the assignment statement is calculated and stored immediately. Thus, when the variable is referenced later, it will contain the current value when it was first assigned. Static assignment is done with the `#definenow` command, as shown below:

```
#define DUM = "Billy"
#definenow AGENT = $DUM
#define DUM = "Joe"

(spec_agent
  (label $AGENT)
)
```

The output from this data would be this:

```
(spec_agent
  (label Billy)
)
```

Control Structures

Type	Syntax

If-then structure	`#if (EXPRESSION)`
	`   value value ...`
	`   value value ...`
	`#endif`

If-then-else-if-then-else structure	`#if (EXPRESSION)`
	`   value value ...`
	`   value value ...`
	`#elseif (EXPRESSION)`
	`   value value ...`
	`   value value ...`
	`#else`
	`   value value ...`
	`   value value ...`
	`#endif`

While structure	`#while (EXPRESSION)`
	`   value value ...`
	`   value value ...`
	`#endwhile`

Expressions	`OPERAND > OPERAND`
	`OPERAND < OPERAND`
	`OPERAND == OPERAND`
	`OPERAND != OPERAND`
	`(EXPRESSION) && (EXPRESSION)`
	`(EXPRESSION) || (EXPRESSION)`

Operands	Number StringWithNoWhitespace "String With Whitespace" Variable `EXPRESSION`

Figure 7.1.2.1: Usage of control structures.

PTaems enables the designer to make use of a rudimentary control structures to better condition the generation process. These structures use local state, along with comparison functions, to control which taems structures should be expressed in the final output.

The simplest control structure is the if...then structure found in most high-level languages. This allows a particular block to be expressed if the condition statement evaluates to a true value. For example:

```
#if (10 > 5)
(Some_Outcome
   (density 8.0)
      (quality_distribution 10 1.0)
      (duration_distribution 5.0 0.3 6.0 0.7)
      (cost_distribution 10 1.0)
   )
)
#endif
```

Since the expression above is trivially true (we assume that 10 is greater than 5 in all cases), the outcome enclosed in the block will always be expressed. The conditional expression is the interesting feature here. Expressions must be fully parenthesized, and can contain variable references and nested expressions, so they can become quite complex looking. See the following example:

```
#if (($FOOBAR == 10) && (($BARFOO < 2) || ($BARFOO > 5)))
(Some_Outcome
   (density 8.0)
      (quality_distribution 10 1.0)
      (duration_distribution 5.0 0.3 6.0 0.7)
      (cost_distribution 10 1.0)
   )
)
#endif
```

Here is an example of a simple while structure, which deccrements a variable while setting a particular attribute value for each step.

```
(spec_task_group
   (spec_attributes
#while ($A != 0)
      (Variable_#trim($A) Float $A)
      #definenow A = #diff($A, 1)
#endwhile
   )
   (label blah)
   (qaf q_sum)
)
```

Expressions can contain static references to numbers or strings. Strings containing multiple tokens (i.e. if it contains any whitespace characters) must be enclosed by quotes. Expression functions will by default use the numeric value of the operands, but if either operand is a string then the lexical value of each of the operands will be used instead. Boolean operators (e.g. && and ||) must have operands that evaluate to boolean values, so `(1 || Jack)`, for instance, would not be valid.

The if-then structure also has a more complex cousin, if the form of the if-then-else-if-then-else structure. This is essentially an if-then structure, followed by zero or more conditional fall-throughs and a catch-all fall-through. Here’s an example:

```
#if ($TIME < 5)
   (quality_distribution 10 1.0)
#elseif ($TIME < 10)
   (quality_distribution 8 1.0)
#elseif ($TIME < 15)
   (quality_distribution 5 1.0)
#else
   (quality_distribution 1 1.0)
#endif
```

Note that the `#elseif` clause is really just a convenience function, the above statement is functionally equivalent to

```
#if ($TIME < 5)
   (quality_distribution 10 1.0)
#else #if ($TIME < 10)
   (quality_distribution 8 1.0)
#else #if ($TIME < 15)
   (quality_distribution 5 1.0)
#else
   (quality_distribution 1 1.0)
#endif
```

This does more or less as one would expect given the indentation - the condition will express a different string based on the `$TIME` variable. We'll conclude with a more complex example, hopefully demonstrating several of the capabilities outlined above.

```
#define DEFAULT_DQ
   (density 0.5)
   (quality_distribution $FOOBAR)
#enddefine

#define EMPTY = " "
#define OUTCOME1 = $EMPTY
#define OUTCOME2 = $EMPTY
#if ($AGENT == Gomer)
   #define OUTCOME1
      (Some_Outcome_1
         $DEFAULT_DQ
         (duration_distribution 5.0 1.0)
         (cost_distribution 2 1.0)
      )
   #enddefine
   #define OUTCOME2
      (Some_Outcome_2
         $DEFAULT_DQ
         (duration_distribution 5.0 1.0)
         (cost_distribution 2 1.0)
      )
   #enddefine

#elseif (($TIME < 25) && ($AGENT == Pyle))
   #define OUTCOME1
      (Some_Outcome_1
         $DEFAULT_DQ
         (duration_distribution 5.0 1.0)
```

```
         (cost_distribution 2 1.0)
      )
   #enddefine
#endif

#if (($OUTCOME1 != $EMPTY) || ($OUTCOME2 != $EMPTY))
(spec_method
   (label Some_Method)
   (agent $AGENT)
   (supertasks Some_Task)
   (outcomes
      $OUTCOME1
      $OUTCOME2
   )
)
; #else
; <Nothing Generated>
#endif
```

Note that the `#else` clause at the end there is actually a comment to clarify what is going on, and will thus have no impact on either the PTaems parse or the final Taems object. Here are several examples of what would be output by the above PTaems expression under various conditions:

```
#define AGENT = Gomer  ; Implicit, passed in as parameter
#define TIME = 100     ; Implicit, passed in as parameter

(spec_method
   (label Some_Method)
   (agent Gomer)
   (outcomes
      (Some_Outcome_1
         (density 1.0)
         (duration_distribution 5.0 1.0)
         (cost_distribution 2.0 1.0)
      )
      (Some_Outcome_2
         (density 1.0)
         (duration_distribution 5.0 1.0)
         (cost_distribution 2.0 1.0)
      )
   )
)

#define AGENT = Pyle  ; Implicit, passed in as parameter
#define TIME = 100    ; Implicit, passed in as parameter

<Nothing Generated>

#define AGENT = Pyle  ; Implicit, passed in as parameter
#define TIME = 10     ; Implicit, passed in as parameter

(spec_method
   (label Some_Method)
   (agent Pyle)
   (outcomes
      (Some_Outcome_1
```

```
            (density 1.0)
            (duration_distribution 5.0 1.0)
            (cost_distribution 2.0 1.0)
        )
    )
)
```

Built-In Functions

Type	Syntax
Sum	`#SUM(operand, operand, ...)`
Product	`#PROD(multiplicand, multiplier, ...)`
Difference	`#DIFF(operand, operand)`
Quotient	`#QUOT(dividend, divisor)`
Random Integer	`#RANDOM_INT(low, high)`
Random Float	`#RANDOM_FLOAT(low, high)`
Random List	`#RANDOM_LIST(item1, "item 2", ...)`
String Concatenate	`#CONCAT(item1, "item 2", ...)`
String Trim	`#TRIM(item1, "item 2", ...)`
Space	`#SPACE()`
Defined-ness check	`#DEF($FOO, $BAR, ...)`
Non defined-ness	`#NDEF($FOO, $BAR, ...)`

Figure 7.1.3.1: Pre-Taems's Built-in functions.

PTaems offers several built-in functions, which can be used more or less interchangeably with static values and variable references. This adds to PTaems the powerful notion of being able to generate completely new Taems data on the fly, where in the previous sections functionality was limited to selecting and rearranging existing structures or characteristics. This is particularly useful when coupled with the ability to pass variables into the parsing process, as a sufficiently sophisticated PTaems script could use this information to dynamically generate new task structures in response to changing goals or environmental conditions.

Below, we'll briefly describe the functions supported in PTaems, and give a short example of how each can be used.

Function Overview

`#SUM`

This function allows you to sum an arbitrary list of numbers.
Ex. A = B + C + 10 + D
=> `#define A = #SUM($B, $C, 10, $D)`

`#PROD`

This function allows you to multiply an arbitrary list of numbers.
Ex. A = B * C * 10 * D
=> `#define A = #PROD($B, $C, 10, $D)`

#DIFF

This function allows you to find the difference between a pair of numbers.

Ex. A = B - 10

=> `#define A = #DIFF($B, 10)`

#QUOT

This function allows you to find the quotient of (divide) a pair of numbers.

Ex. A = 10 / B

=> `#define A = #QUOT(10, $B)`

#RANDOM_INT

This function allows you to generate a random integer value at runtime. Returned numbers will be between the low (inclusive) and high (not inclusive) bounds.

Ex. A = random float between (B and 20)

=> `#define A = #RANDOM_INT($B, 20)`

#RANDOM_FLOAT

This function allows you to generate a random float value at runtime. Returned numbers will be between the low (inclusive) and high (not inclusive) bounds.

Ex. A = random float between (-1.5 and B)

=> `#define A = #RANDOM_FLOAT(-1.5, $B)`

#RANDOM_LIST

This function allows select an arbitrary string from a predefined list of strings.

Ex. A = one of (jack, jill, $HILL, pail of water)

=> `#define A = #RANDOM_LIST(jack, jill, $HILL, "pail of water")`

#CONCAT

This function concatenates several strings (or variables, or function results) into one string. Note that internal whitespace may actually cause the result to be interpreted as multiple tokens by the Taems parser, see the trim function to work around this, or the space function to cause it.

Ex. A = concatenation of (jack, jill, $HILL)

=> `#define A = #CONCAT(jack, jill, $HILL)`

#TRIM

This function trims any whitespace from the beginning or end of the parameters, and returns a concatenation of them. Note that whitespace *inside* individual strings may still force the result to be parsed as mulitple tokens.

Ex. A = single, trimmed token of (jack, jill, $HILL)

=> `#define A = #TRIM(jack, jill, $HILL)`

#SPACE

This function outputs a space, which is useful when generating list Taems elements.

Ex. A = "jack and jill"

=> `#define A = #CONCAT(jack, #SPACE(), and, #SPACE(), jill)`

#DEF

This function returns true if the variable has been previously defined, either directly in the .ptaems file, or through some passed in variable. If multiple parameters are passed in, the function will return true if all are defined.

Ex. Do something if FOOBAR is defined

=> `#if (#def($FOOBAR) == true)...`

#NDEF

This function returns true if the variable has not been previously defined. If multiple parameters are

passed in, the function will return true if all are not defined.
Ex. Do something if FOOBAR is not defined
=> `#if (#ndef($FOOBAR) == true)...`

Note that the `RANDOM` functions may be seeded with the variable `RandomSeed`. Typically, when used by an agent at runtime, a seed will be passed by the agent into the parser, allowing for deterministic generation of structures which make use of these functions. This seed can in turn be overridden with a normal `#define RandomSeed` within the PTaems file itself. Finally, if neither option is used to pass a seed to the parser, the current time (or some other sufficiently random number) is used, which will make generation non-deterministic.

Most of the functions are fairly simple and self explanatory. To give you a better notion about where they might be used, here are a few out of context examples using the functions described above.

```
#define QUALITY = #RANDOM_INT(5, 10)
#define DURATION = #PROD($QUALITY, 2)
#define COST = #QUOT($QUALITY, #RANDOM_INT(1, 4))
```

The above example shows how one might randomly generate the parameters for an outcome. The quality is a random number between 5 and 10, the duration is twice the quality, and to make things interesting the cost can be anywhere from 100% to 25% of the quality.

```
#define GOAL = #RANDOM_LIST(Do_Chores, Watch_TV)
#if ($GOAL == Do_Chores)
   #define COORD_PARTNER = #RANDOM_LIST(Sister, Brother, $AGENT)
#else
   #define COORD_PARTNER = #RANDOM_LIST(Tom, Dick, Harry)
#endif
#define GOAL_DEADLINE = #SUM($TIME, #RANDOM_INT(20,40))
#define COORD_TIMEOUT = #DIFF($GOAL_DEADLINE, 10)
```

Here we are generating meta-level constraints for our agent. First we select a high-level goal, which presumably will also (elsewhere in the PTaems file) dictate which task structure gets instantiated. Given a goal, we now face the decision of who to coordinate with concerning the goal. If we are to `Do_Chores` then we might enlist a sibling, or possibly just do it by ourselves. If we decide to slack off and `Watch_TV`, however, we'll definitely get in touch with Tom, Dick or Harry to come join the fun. In either case, we have some sort of goal deadline that must be generated, and given that, some notion of a timeout on the coordination act itself - because it won't do us any good to continue coordination if the partner wouldn't be able to perform the activity in time.

Notice in this example we've generated some seemingly non-Taems related data - this can be used either to control things elsewhere in the PTaems file (as with `GOAL`), or stored within other Taems objects (`COORD_TIMEOUT` could be placed in the attributes field of a Commitment object, for instance) where the agent problem solver could make use of them.

You'll also see that even with these simple functions, the PTaems processor can begin to simulate the activities a problem solver might perform. We believe it possible that a sophisticated script combining generated values with prespecified task structure pieces could mimic the reasoning activities of a domain-specific problem solver, potentially reducing the effort needed to construct agents.

```
#define DEADLINE = 40;
#define TIME_REMAINING = #DIFF($DEADLINE,$TIME);
#define TASKNAME = #CONCAT(Task_for_time_,#TRIM($TIME))

(spec_task_group
   (label $TASKNAME)
   (agent $AGENT)
#if ($TIME_REMAINING < 10)
   (subtasks #RANDOM_LIST(Short1A, Short2A)
             #RANDOM_LIST(Short1B, Short2B))
#else
   (subtasks #RANDOM_LIST(Normal1A, Normal2A)
             #RANDOM_LIST(Normal1B, Normal2B, Normal3B))
#endif
   (qaf #RANDOM_LIST(q_min, q_max, q_sum))
)
```

In this example, we start out by calculating the amount of time remaining and then generate the task's name. Because variable derefernces by default prepend spaces to their result, we must `TRIM($TIME)` to append the bare number to the name prefix. Later in the definintion, we can see that a random QAF is selected with the `RANDOM_LIST` function. Finally, we vary the possible subtask lists by the remaining time, and select random subtasks within each list. This is what the output would look like for agent Gomer at time 40.

```
(spec_task_group
   (label Task_for_time_20)
   (agent Gomer)
   (subtasks Normal1A Normal2B)
   (qaf q_sum)
)
```

You may frequently come across a situation where you wish to build up a Taems list entry - a subtasks list for instance. The typical instinct is to generate this list with `define` commands, like this:

```
#define SUBTASKS = ""
#define SUBTASKS = #CONCAT($SUBTASKS, "Task_1")
#define SUBTASKS = #CONCAT($SUBTASKS, #SPACE(), "Task_2")
```

The problem here is that, as discussed in the variables section, `define` is a dynamic command. When the above `$SUBTASKS` variable is actually referenced, only the last assignment will be available in memory. The reference to `$SUBTASKS` within that last assignment will then be dynamically bound to that same `define` statement. The end result (if you are still following me) is that an endless recursion will be produced, which will neatly cause a stack overflow (that's bad). To get around this problem, you must use static assignments, so each sucessive define builds on the current results from the last. Here is a correct example:

```
#define SUBTASKS = ""
#definenow SUBTASKS = #CONCAT($SUBTASKS, "Task_1")
#definenow SUBTASKS = #CONCAT($SUBTASKS, #SPACE(), "Task_2")

(spec_task_group
  (label Some_Task)
  (agent Some_Agent)
  (subtasks $SUBTASKS)
  (qaf q_sum)
)
```

And it's output... (Note we also had to use the `#SPACE` function to correctly separate the entries.)

```
(spec_task_group
  (label Some_Task)
  (agent Some_Agent)
  (subtasks Task_1 Task_2)
  (qaf q_sum)
)
```

Examples

In this section we'll cover some examples of how PTaems can be used.

Example 6: Producer/Consumer/Transporter

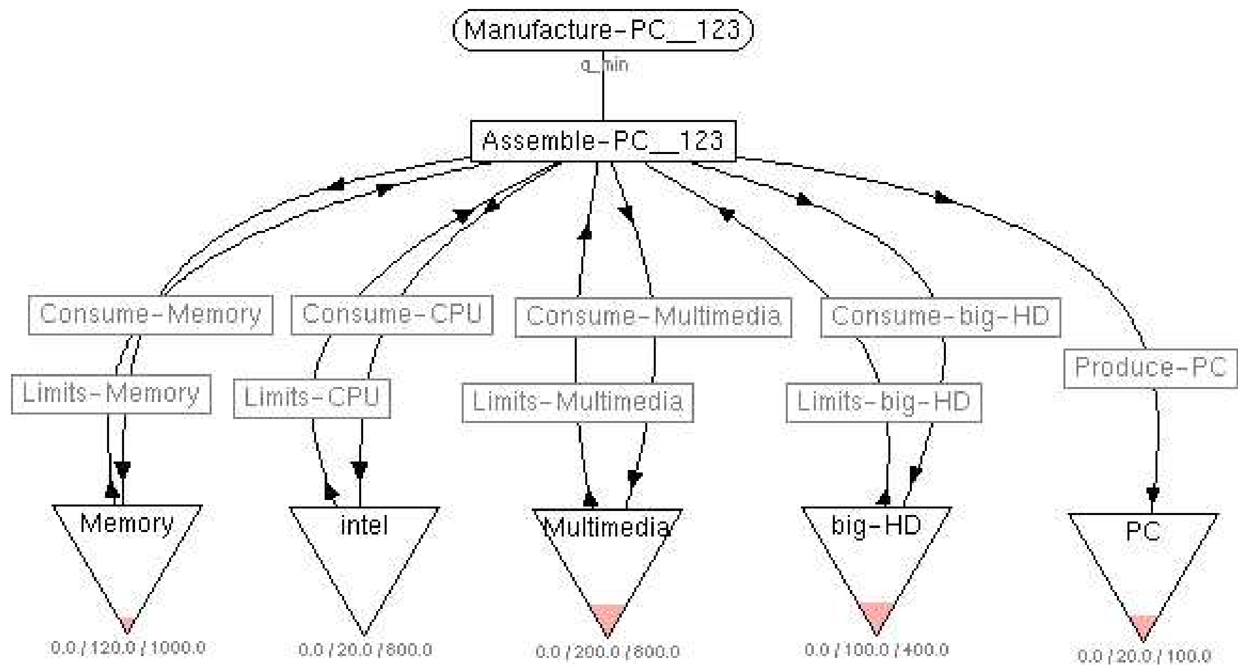


Figure 4.6.1: PCT Agent [psrc] [tsrc]

Proposed Extensions

In this section we will cover potential additions to the Taems modeling language. In general, we will document the proposed extensions, why they are needed, and what impact they might have on existing tools. As the features are evaluated and approved, they will be incorporated into the general Taems specification given in earlier sections.

Weights

This extensions allows the modeler to associate a weighting value with the quality produced by a particular task or method. For instance, if you have a task A under a task group G, you may specify that only a certain percentage of A's actual quality will be used to calculate the quality of G. An unweighted task in Taems therefore has a weight of 100% - all of it's quality can be accumulated by its supertasks. Using weights, however, you can use just a fraction of the quality (e.g. 20%), or much more than would normally be available (e.g. 400%).

See Tom Wagner's Technical Memo for more information on this subject.

Iteration

Lacking in Taems is any notion of a repetative, or iterative activity. One can think of many instances where it would be useful and appropriate to model actions as being expicitly iterative, dependant on some other variable in the model or environment. The following reference is a memo covering our thoughts on the implementation of iterative actions in Taems.

Thomas Wagner, Victor Lesser, Keith Decker, and Alan Garvey, "Iterative Tasks in TAEMS: Thoughts and Issues," *MAS Lab Technical Memo 97-01*, October, 1997.

Last and Seq Last

The q_last and q_seq_last QAFs add to Taems the idea that a task's quality is that of the subtask beneath it which last completed. Thus, the quality of such a task is not monotonic, it can rise or fall as its subtasks complete. A q_seq_last QAF adds the further constraint that the methods must also be performed in order, and all must be performed for any quality to be accrued. In this case, the quality of a task with a q_seq_last QAF will be zero until the final subtask has completed, at which time the task will obtain the quality of that final subtask. (Note the contraint that all subtasks must be performed for any quality to be accrued is slightly different than the semantics of other seq QAFs (other than q_seq_min), which can obtain partial quality if a subset of their subtasks are performed in order.)

Note that the subtasks do not necessarily have to complete *successfully*, they just need to have been attempted, so that a failed method (which has a final quality of 0) will not prevent its supertask from accruing any quality unless it is also the most recently completed method.

The figure and table below show how these QAFs work in practice. Each row in the table corresponds to a different execution sequence, with each action's resultant quality given in parenthesis.

1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	3
A (5)	B (2)	C (7)	-	7
B (2)	A (5)	D (3)	C (7)	7
A (5)	B (2)	D (0)	C (7)	7

Figure: Final quality examples for q_last.

1st Action (Q)	2nd Action (Q)	3rd Action (Q)	4th Action (Q)	Final Quality
A (5)	B (2)	C (7)	D (3)	3
A (5)	B (2)	C (7)	-	0
B (2)	A (5)	D (3)	C (7)	0
A (5)	B (2)	D (0)	C (7)	0

Figure: Final quality examples for q_seq_last.

References

General References

- Taems Related MASL Publications
  Bibliographical information for papers published by the MASL group related to Taems.
- Taems Research Overview
  Background information, links to more related research and comments on the direction of Taems research.
- Java Taems API
  The API reference documentation for our current Taems implementation.
- TTaems working spec
  A less verbose version of the TTaems specification.

Applications

- DTC - Design-to-Criteria
  The Design-to-Criteria scheduler uses Taems task structures, along with fixed or potential commitments and user preferences, to generate activity schedules. Design-to-Criteria is a heuristic scheduling technique that analyzes the goal achievement process encoded in the task structure and generates and ranks schedules that satisfy that process, while respecting the criteria and environmental constraints given to it.
- GPGP - Generalized Partial Global Planning
  GPGP is designed to both discover and take advantage of interdependancies that occur in task structures belonging to different agents. The goal of the project is to develop a generic set of coordination protocols allowing agents using Taems structures to represent their knowledge to easily exhibit sophisticated behaviors.
- BIG - Bounded Information Gathering
  The BIG agent uses a dynamically generated Taems structure as its internal problem solving representation. This structure is fed to a design-to-time scheduling component which generates an efficient schedule of activity which satisfies both the information gathering and processing needs of the agent.
- MASS - The Multi Agent Systems Simulator
  The MASS simulator uses a objective Taems structure to derive the quantitative characteristics of agent activities and environmental effects produced by the agents under its control.
- IHome - The Intelligent Home Project
  All of the agents operating in the Intelligent Home use Taems as their internal problem solving model. Using the Taems structure scheduling techniques are used to direct agent activity, and interdependancies between other agents and resources in the environment are coordinated over when necessary.

Appendix

This section contains information about topics which closely relate to Taems, but are not within the scope of the more Taems specific sections above. The inconsistencies section enumerates some parts of the Taems modeling system which arguably need to be better specified or redesigned. The remainder of the sections relate to objects which are rarely used outside of the context of a Taems structure.

Commitments

Commitments are used by an agent to represent the quantitative aspects of negotiation and coordination. The pages in this section describe the two generic styles of commitments currently supported in Taems. While these objects do have slots for many (if not most) of the values that need to be represented as part of negotiation, you will likely find that as you develop new protocols not all of your data will fit into the provided fields. When this is the case, we recommend you use the attributes field of the commitment to store the data in question. So long as both parties agree on which attribute keys are used to store which data, you may also use textual representation of commitments as your primary communication vehicle - instead of creating a new message type you simply send commitments. This offers several benefits: you don’t have to create a new message format, you can directly include textual representations of other Taems objects and you can make use of existing textual Taems parsing utilities.

Commitment (Local)

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
type	Predefined Symbol	no
from_agent	Agent Symbol	no
to_agent	Agent Symbol	no
task	Symbol List	no
importance	Float	yes
minimum_quality	Float	yes
earliest_start_time	Integer	yes
deadline	Integer	yes
dont_interval_start	Integer	yes
dont_interval_end	Integer	yes
time_satisfied	Integer	yes

Figure A.1.1.1: Specification of spec_commitment.

```
(spec_commitment
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label com12)
   (type deadline)
   (from_agent Agent_A)
   (to_agent Agent_B)
   (task Method2 Method6)

   (importance 10)
   (minimum_quality .01)
   (earliest_start_time -1)
   (deadline 14)

   (dont_interval_start -1)
   (dont_interval_end -1)

   (time_satisfied -1)
)
```

Figure A.1.1.2: Example commitment in TTaems.

Field Overview

`spec_attribute`

See Elements. The attributes field is sometimes used in a commitment to store commitment or protocol-specific information which doesn't fit into any of the supplied fields. Some examples of data stored in here are CoordinateID (the ID of the coordination task the commitment belongs to), Resource (the name of the resource being coordinated over), Quantity (the amount being coordinated over), and cost_distribution (the cost associated with the coordination).

`label`

The label field should provide some sort of description of the commitment, it does not need to be unique within the set of commitments known to the agent.

`type`

Defines the type of commitment being described. This implicitly specifies both the set of fields pertinent to the commitment, and the semantics behind them. What follows is a brief description of each, in each case the recipient is the `to_agent`, while the source is the `from_agent`.

`do`

The commitment recipient is expected to perform one or more tasks for the source agent.

`dont`

The commitment recipient is expected to not perform one or more tasks for the source agent.

`deadline`

The commitment recipient is expected to complete one or more tasks for the source agent by a given deadline.

`earliest_start_time`

The commitment recipient is expected to start one or more tasks for the source agent by a certain time.

`est_and_dl_and_do`

A combination of the respective types above, this type enforces a do commitment with both an earliest start time and a deadline, if those fields have a non-negative value. If neither field is positive, this type is functionally equivalent to a simple `do`.

`special`

This tag type indicates that the commitment does not follow any of the above styles, and therefore no assumptions should be made about what fields are used, or the exact semantics behind them. This field was added to provide the flexibility needed to describe commitments which fall outside the scope we give here, while maintaining syntatic compatibility with existing tools. You should assume that generic tools will ignore all commitments with the `special` type.

Accepted symbols: `do`, `dont`, `deadline`, `earliest_start_time`, `est_and_dl_and_do`, `special`.

`from_agent`

The source, or originator, of the commitment. If agent A asks agent B to perform a task, then agent A is the `from_agent`, and should be represented as such in the commitments stored by both agents.

`to_agent`

The destination, or target, of the commitment. If agent A asks agent B to perform a task, then agent B is the `to_agent`, and should be represented as such in the commitments stored by both agents.

`task`

A list of the tasks or methods associated with the commitment. It may indicate those tasks which

should be performed (do-style) or those which should not (dont-style).
Note: Some tools may interpret this list as a logical OR - the commitment is satisfied when any subset is completed (or avoided) while respecting the other constraints on the commitment. Other tools may view it as an AND - all tasks must be completed (or avoided) to satisfy the commitment. Refer to the specific documentation for the tool in question for more details.

`importance`
: The importance associated with the commitment. This describes how important it is to the `from_agent` that the commitment be satisfied. This information may be used (along with other information) to determine which commitments to satisfy under bounded circumstances. In general, a negative value indicates a what-if style question, zero a normal commitment, and positive one that cannot be broken.

`minimum_quality`
: The minimum quality bound that can be achieved for the commitment to be considered satisfied by the `from_agent`. This implies that it is possible for the `to_agent` to perform the task successfully (i.e. obtain non-zero quality), yet still fail to complete the task in the eyes of the `from_agent`.

`earliest_start_time`
: The earliest point in time at which the `to_agent` should begin the task or behavior associated with the commitment. A negative value indicates no bound is placed on the commitment's starting time.

`deadline`
: The latest point in time by which the `to_agent` should have completed the task or behavior associated with the commitment. A negative value indicates no bound is placed on the commitment's ending time.

`dont_interval_start`
: Analogous to `earliest_start_time`, this field specifies for `dont` type commitments the starting point in time when the `to_agent` should not perform a certain task.

`dont_interval_end`
: Analogous to `deadline`, this field specifies for `dont` type commitments the ending point in time when the `to_agent` should not perform a certain task. Together with `dont_interface_start`, this provides the bounds on a dont commitment. The `to_agent` should is asked to not perform some task for all times *t* where (`dont_interval_start` <= *t* <= `dont_interval_end`).

`time_satisfied`
: Indicates the actual time at which the commitment was satisfied or completed. A negative value indicates the commitment has not yet been satisfied.

Commitment (Non Local)

Field Name	Data Type	Optional
spec_attributes	Special	yes
label	Symbol	no
from_agent	Agent Symbol	no
to_agent	Agent Symbol	no
task	Symbol List	no
quality_distribution	Distribution	no
time_distribution	Distribution	no

Figure A.1.2.1: Specification of spec_nonlocal_commitment.

In contrast to the normal Taems commitment, which is kept local to the agent who makes it, nonlocal commitments are used to represent a commitment made by some other agent. Normally, an agent gets to know a nonlocal commitment by receiving it through agent communication. Agents adhering to this model do not therefore exchange local commitments; an agent has to make a conversion from a local commitment to a nonlocal commitment when it needs to tell other agents about its local commitments. (NOTE: this is the "traditional" view, other developers work under the belief that the nonlocal commitment is not needed at all, although the nonlocal method is conceptually simpler, though restrictive).

A nonlocal commitment has fewer slots than local commitments. This implies, of course, that information is lost, perhaps most importantly the type of commitment. The simple specification of nonlocal commitment makes a it look more or less like a usual contract offering, where all that matters is when a task is to be completed and what quality is requires. The implication for this is that it does not support dont type local commitments, as it is impossible to convert a dont type local commitment to a nonlocal commitment.

For do, deadline, est_and_dl_and_do, or earliest_start_time type of local commitments, to derive a nonlocal commitment (essentially, to figure out the distribution of quality and completion time) usually involves looking at the agent's current schedule. For example, if you have a do commitment, you look at the schedule to find the referenced method/task, find out its finish time distribution, and its quality outcome distribution.

However, these distributions are often hard to compute from the schedule, because of all the uncertainty about the preceding tasks, and the interrelationships. So often you'll try to do some approximation to avoid the extra computation. Usually you want the approximation to be faithful, i.e., do not claim earlier completion time or higher quality outcome than the schedule indicated. Sometimes you can use the information in the local commitment, such as minimum_quality and deadline, to give an approximation.

```
(spec_nonlocal_commitment
   (spec_attributes
      (some_attribute 1)
      (some_other_attribute foo bar)
   )
   (label nlc3)
   (from_agent Agent_A)
   (to_agent Agent_B)
   (task Method2)

   (quality_distribution 12 0.5 15 0.5)
   (time_distribution 13 1.0)
)
```

Figure A.1.2.2: Example nonlocal commitment in TTaems.

A further question arises when one considers how the nonlocal commitment should be used in scheduling. Here we only discuss the case where the scheduling agent is the `to_agent`. Since, when the scheduling agent is the `from_agent`, the agent should use the original local commitment from which the nonlocal commitment is derived; and, if the scheduling agent is not `to_agent` nor `from_agent`, this commitment has nothing to do with the scheduling agent. In the old Taems model, where many agents' views were mixed altogether in one textual file, agents could use the `from_agent` and `to_agent` slot to pick the nonlocal commitments related to them when constructing their individual Taems model.

The task indicated in the `task` field must appear as a nonlocal method in the scheduling agent's Taems structure, although the same name could refers to a task or even a task group in the `from_agent`'s structure.

On the other hand, scheduling with a nonlocal method needs a nonlocal commitment to support it. You won't be able to schedule a nonlocal method (i.e. obtain a schedule) without a nonlocal commitment. So an agent needs to first incorporate a nonlocal method (usually via communication), then receive a nonlocal commitment (also via communication), then do the scheduling. This order cannot be altered.

Field Overview

`spec_attribute`
: See Elements. The attributes field is sometimes used in a commitment to store commitment or protocol-specific information which doesn't fit into any of the supplied fields.

`label`
: The label field should provide some sort of description of the commitment, it does not need to be unique within the set of commitments known to the agent.

`from_agent`
: This is the agent who had the original local commitment from which this nonlocal commitment is derived, i.e., the agent offering this commitment.

`to_agent`
: The agent being offered the nonlocal commitment.

`task`
: The name of the task or method involved. Note here a nonlocal commitment allow you to specify one task only, where as in a local commitment you can specify more than one.

`quality_distribution`
: The expected quality distribution which should be obtained when the task is performed.

`cost_distribution`
: The expected duration distribution which should be obtained when the task is performed.

Schedules

Field Name	Data Type	Optional
`schedule_elements`	Special	no
`start_time_distribution`	Distribution	no
`finish_time_distribution`	Distribution	no
`quality_distribution`	Distribution	no
`duration_distribution`	Distribution	no
`cost_distribution`	Distribution	no
`rating`	Float	no

Figure A.2.1: Specification of `spec_schedule`.

The schedule object gives the agent and easy way of representing activity sequences derived from a particular task structure. They also provide a consistent and simple way for a scheduling component to report its results to the agent. The schedule itself will typically consist of a ordered series of *schedule elements*, each of which represents a particular activity that should take place. When the schedule was generated from a particular Taems task structure, the schedule elements themselves will each correspond to a particular method contained in that task structure. Each element will also be annotated with its expected performance characteristics, as will the schedule as a whole.

```
(spec_schedule
   (schedule_elements
      (Method4
         (start_time_distribution  0.0 1.0)
         (finish_time_distribution  10.0 1.0)
         (quality_distribution  50.0 1.0)
         (cost_distribution  8.0 1.0)
         (duration_distribution  10.0 1.0)
      )
      (Method2
         (start_time_distribution  10.0 1.00
         (finish_time_distribution  20.0 1.0)
         (quality_distribution  35.0 1.0)
         (cost_distribution  6.0 1.0)
         (duration_distribution  10.0 1.0)
      )
   )
   (start_time_distribution 0.0 1.0)
   (finish_time_distribution 20.0 1.0)
   (quality_distribution  85.0 1.0)
   (duration_distribution  20.0 1.0)
   (cost_distribution  14.0 1.0)
   (rating 1.0)
)
```

Figure A.2.2: Example schedule in TTaems.

Two other fields, `task_quality_info` and `commitment_info`, are proposed extensions to this specification. The first is a list of task/quality pairs, while the second provides information on commitment satisfaction or violation.

Field Overview

`schedule_elements`

This "field" is where the bulk of the schedule data is contained. Within the field are contained an ordered list of zero or more items, called schedule elements, which represent the proposed sequence of actions or methods defined by the schedule. Each schedule element begins first with the name of the method or action which should be taken, followed by several fields describing the expected performance of that method. A more complete description of each field follows.

Field Name	Data Type	Optional
`start_time_distribution`	Distribution	no
`finish_time_distribution`	Distribution	no
`quality_distribution`	Distribution	no
`cost_distribution`	Distribution	no
`duration_distribution`	Distribution	no

Figure A.2.3: Specification of schedule element.

Each schedule element definition begins with the name of the method or action which is to be performed. This will typically correspond to the `label` of a method contained in the task structure the schedule was generated from. For instance, in Figure A.3.2, we see the actions Method4 and Method2 are to be performed. Some schedulers may also use special keywords here to indicate non-action events. For instance, IDLE or SLACK_MYTIME may be used to indicate a period during which the agent should wait or may use for other tasks.

`start_time_distribution`

This distribution describes the expected start time of the method, e.g. the point in time at which the scheduler expects the agent will begin performing the associated action.

`finish_time_distribution`

This distribution describes the expected finish time of the method, e.g. the point in time at which the scheduler expects the agent will complete performing the associated action.

`quality_distribution`

This describes the expected quality the method will accrue.

`cost_distribution`

This describes the expected cost the method will accrue.

`duration_distribution`

This describes the total amount of time the method is expected to take to complete.

`start_time_distribution`

This describes the actual point in time at which the scheduler anticipates the agent will begin working on the schedule. This will typically be the same as the `start_time_distribution` of the first

schedule element.

`finish_time_distribution`
: This describes the actual point in time at which the scheduler anticipates the agent will complete working on the schedule. This will typically be the same as the `finish_time_distribution` of the last schedule element.

`quality_distribution`
: This represents the expected total quality which may be accumulated by performing all methods in the schedule.

`duration_distribution`
: This represents the expected total duration which may be needed to perform all methods in the schedule.

`cost_distribution`
: This represents the expected total cost which may be accrued by performing all methods in the schedule.

`rating`
: This value indicates how well the scheduler determined the particular scheduler satisfied the scheduling criteria, which covers such things as desired cost, quality, duration, uncertainty and commitment satisfaction. This will typically be a float between 0 and 1.

Schedule Setting Bundles

Field Name	Data Type	Optional
`label`	Symbol	no
`quality_value`	Float or Predefined Symbol	no
`duration_value`	Float or Predefined Symbol	no
`cost_value`	Float or Predefined Symbol	no

Figure A.3.1: Specification of `spec_schedule_setting_bundle`.

While distributions are part of DTC's computation and scheduling model, expected values are also used during scheduling to compose schedules. For example if method A is the first item on the schedule, but its duration is uncertain (having more than one possible value), the next method scheduled after it inherits the uncertain duration of A. Say B follows A, B's start time is thus uncertain because the finish time of A is uncertain. When queried when (exactly) B will execute, the scheduler will generally respond with the expected value of B's start time distribution. The value used when a single value is required by an algorithm or a scheduling client defaults to the expected value of the distribution.

However, because expected values are sensitive to outliers (in a statistical sense) and to support scheduling for different environments, scheduler clients can instruct the scheduler to use something other than the expected value. For example, to always build schedules that have no probability of exceeding their stated end times, a client may wish to schedule with the max statistic in duration rather than the expected value. (Note, with respect to hard deadlines and the like, the probability that a schedule will cross a hard deadline is reflected in the expected quality distribution of the schedule.) The scheduler is designed to support scheduling with any percentile range, as well as the minimum, maximum, mean, median, and trimmed mean values.

The scheduling preference is specified via the `spec_schedule_setting_bundle`, and different statistics can be used for quality, cost, and duration. The bundle itself is used just like a Taems object, and can be inserted anywhere in the TTaems representation (outside of other objects).

```
(spec_schedule_setting_bundle
   (label ssb)
   (quality_value 0.5)
   (duration_value mean)
   (cost_value max)
)
```

Figure A.3.2: Example schedule setting bundle in TTaems.

Field Overview

`label`
: See Elements.

`quality_value`
: Accepted symbols: A Float, or `min`, `max`, `mean`, `median`.

`duration_value`
: Accepted symbols: A Float, or `min`, `max`, `mean`, `median`.

`cost_value`
: Accepted symbols: A Float, or `min`, `max`, `mean`, `median`.

Scheduler Criteria

Field Name	Data Type	Optional
label	Symbol	no
goodness_quality_slider	Float	no
goodness_cost_slider	Float	no
goodness_duration_slider	Float	no
threshold_quality_slider	Float	no
threshold_cost_slider	Float	no
threshold_duration_slider	Float	no
quality_threshold	Float	no
cost_limit	Float	no
duration_limit	Float	no
uncertainty_quality_slider	Float	no
uncertainty_cost_slider	Float	no
uncertainty_duration_slider	Float	no
threshold_certainty_quality_slider	Float	no
threshold_certainty_cost_slider	Float	no
threshold_certainty_duration_slider	Float	no
quality_certainty_threshold	Float	no
cost_certainty_threshold	Float	no
duration_certainty_threshold	Float	no
meta_goodness_slider	Float	no
meta_threshold_slider	Float	no
meta_uncertainty_slider	Float	no
meta_uncertainty_threshold_slider	Float	no

Figure A.4.1: Specification of `spec_evaluation_criteria`.

These values are sent to the scheduler to control how it should evaluate and compare the schedules that it produces. For instance, one set of values might tell the scheduler that the best overall quality is desired, while another might direct it to select the cheapest alternative. The notion of a set of "sliders" that describe some property furthermore enables the agent to specify these in relation to one another - for instance that the agent weights its schedule preferences as 40% quality, 30% cost and 20% duration.

The sliders themselves consist of sets of three or four values (the `goodness_*_slider` set has three, for instance, while `meta_*_slider` has four). These values are used relative to one another, so that if the value of one slider is set to 0.8 and another to 0.4, the component using those values should translate that as meaning the first value is twice as important as the first - the values themselves have no units or meaning on their own. Some implementations may enforce a total bound on a set, that the slider values should sum to 1 or 100 for instance, but this is for cosmetic purposes only and is not part of the exact specification. Thus, slider settings of (0.4, 0.1, 0.5) would be functionally equivalent to (40, 10, 50) or (8, 2, 10).

```
(spec_evaluation_criteria
   (label my_scheduling_criteria)
   (goodness_quality_slider .60)
   (goodness_cost_slider 0.30)
   (goodness_duration_slider 0.10)
   (threshold_quality_slider 0.33)
   (threshold_cost_slider 0.33)
   (threshold_duration_slider 0.34)
   (quality_threshold 25)
   (cost_limit 5)
   (duration_limit 10)
   (uncertainty_quality_slider 0.2)
   (uncertainty_cost_slider 0.7)
   (uncertainty_duration_slider 0.1)
   (threshold_certainty_quality_slider .1)
   (threshold_certainty_cost_slider    .2)
   (threshold_certainty_duration_slider   .3)
   (quality_certainty_threshold  1)
   (cost_certainty_threshold  2)
   (duration_certainty_threshold  3)
   (meta_goodness_slider 1.0)
   (meta_threshold_slider 0.0)
   (meta_uncertainty_slider 0.0)
   (meta_uncertainty_threshold_slider 0.0)
)
```

Figure A.4.2: Example schedule evaluation criteria in TTaems.

To make the slider metaphor clearer, we have included the figure below, which shows a graphical mock-up of what an interface to this object might look like. Note that in this figure, the slider sets used a fixed total of 100 (percent). This cap means that the individual values are no longer independent, so a user or agent using this interface would see reactions to other slider values in the set when changes were made to one of them.

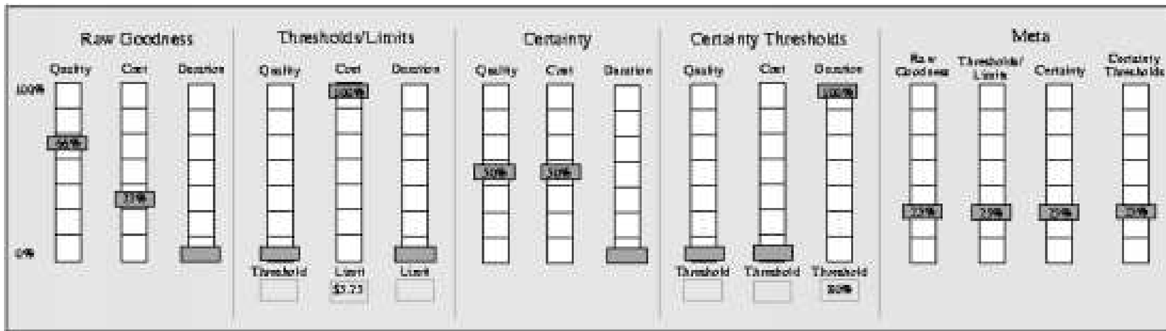


Figure A.4.3: Example slider interface to evaluation criteria.

Field Overview

`label`

See Elements.

`goodness_quality_slider`
`goodness_cost_slider`
`goodness_duration_slider`

This slider set contains sliders for each dimension - quality, cost and duration. Its purpose is to describe the relative importance of each dimension in the schedule. For example, setting quality to 0.5 and cost and duration to 0.25 (each) expresses the notion that quality is twice as important as each of the other dimensions, and that it should weigh twice as heavily as each when evaluating schedules or alternatives. Alternately, this also says that cost and duration combined are equally as important as quality.

`threshold_quality_slider`
`threshold_cost_slider`
`threshold_duration_slider`

This slider set also models the dimensions seen above, however, in this set each slider is paired with an absolute value that describes thresholds (lower bounds) or limits (upper bounds). This set allows agents to set minimum desired quality thresholds (`quality_threshold`) and maximum duration (`duration_limit`) and cost (`cost_limit`) limits, and then to describe how important these limits are relative to each other. They essentially denote the criticality of meeting this specified bounds in the schedules that are generated. Note we do not mean threshold or limit in the hard constraint sense - schedules that fail to meet these bounds may still be returned for execution.

At first glance, it is intuitive to think that the sliders in this set should be omitted and that the relative importance described in the raw goodness slider set should be used instead. While the relative importances expressed in the two sets of sliders may be identical, this separation allows the client to specify concepts like "Cost, quality and duration are equally important in general, but schedules whose quality is over my threshold are particularly important."

`quality_threshold`
`cost_limit`
`duration_limit`

These three values specify the bounds for quality cost and duration for the threshold sliders shown above. These are considered "soft constraints", as the scheduler will attempt to meet them, but there is some possibility schedules which do not meet these requirements are returned. The degree to

which the scheduler attempted to adhere to these limits is dictated by their respective goodness and threshold sliders.

`uncertainty_quality_slider`
`uncertainty_cost_slider`
`uncertainty_duration_slider`

Whereas the other slider sets address quality, cost and duration issues, this slider set describes how important uncertainty is to the client. In particular applications it may be more desirable to pick a slower, more costly schedule that returns lower expected quality because the certainty about these values is very high. This slider set thus contains a slider for each dimension, quality, cost and duration, and it defines how important reducing uncertainty in each dimension is relative to the other dimensions.

`threshold_certainty_quality_slider`
`threshold_certainty_cost_slider`
`threshold_certainty_duration_slider`

Just as the previous threshold slider set placed bounds on Q/C/D values of the schedule, so does this set place thresholds on the allowed (un)certainty of quality, cost and duration. This allows the agent, for instance, to say that it is important for the agent to meet a particular certainty threshold for the quality of a schedule.

`quality_certainty_threshold`
`cost_certainty_threshold`
`duration_certainty_threshold`

These three values specify the bounds for quality cost and duration for the threshold certainty sliders shown above.

`meta_goodness_slider`
`meta_threshold_slider`
`meta_uncertainty_slider`
`meta_uncertainty_threshold_slider`

This slider set relates the important of the four previous slider sets (goodness, threshold, uncertainty and threshold certainty (i.e. uncertainty threshold)). This allows the agent to focus on specific issues within a related set, and then take a step back and decide how important each of the different aspects of the slider set are relative to each other. For instance, if `meta_goodness_slider` is set to 1.0, while the other three are set to zero, the scheduling process would take it to indicate that only the raw goodness sliders should really be used when generating and analyzing schedules.

Inconsistencies and Deficiencies

In this section of the appendix we'll look at some of the inconsistencies present in the current incarnation of Taems. These are basically things that we know aren't designed quite to our satisfaction, but have not yet devoted the resources to to remedy quite yet.

1. Limits interrelationships. The current model of limits has the same interrelationship becoming active regardless of which bound the resource has violated. A better model would separate these two cases, or possibly integrate the limits effects into the produces and consumes interrelations themselves.
2. Sigmoid QAF. The textual syntax of this QAF is different from the others.
3. Interrelationship effects. Produces, consumes and limits support midstream interruption and effects, while other types do not. Some effects are percentage based, others are absolute.
4. Multiple interrelationships. There is no way to model OR-based effects by multiple interrelationships (e.g. either this or that enables must be active for the method to be enabled).
5. Commitments. The current, mostly static, field definitions for commitments often don't have fields required by many protocols, while at the same time posess fields that most protocols don't need. Care needs to be taken if this is redesigned that components analyzing commitments (such as the scheduler and diagnosis components) may not have the necessary protocol-specific information to know which attributes fields contain pertinant information.