C_TAEMS Language Specification

Version 2.04

Mark Boddy, Bryan Horling, John Phelps, Robert P. Goldman, Regis Vincent, A. Chris Long, Bob Kohout, Rajiv Maheswaran

Chapter 1: Introduction

This document defines the scope, syntax, and semantics of C_TAEMS, a derivative of the TAEMS language that will be employed for specifying task domains for DARPA's Coordinators program. In addition, this document specifies the interpretation and implications of ways in which that specification may be changed, for example by adding a new task or by shifting a deadline.

1.1 What This is Not

This document is not intended to do any of these things:

- Provide motivation or a rationale for TAEMS.
- Summarize the history, development, or current versions of the various flavors of TAEMS.
- Survey previous TAEMS applications
- Provide a TAEMS modeling tutorial.

For these and other more general TAEMS-related issues, a good place to start would be...

Some of these could conceivably be addressed in appendices, eventually.

1.2 Approach

The guiding philosophy behind this document is based on the following:

- Cleanly separate the statement of the problem (task structure and initial schedule), from decision-making about what to execute when (that's the Coordinators problem), and from the resulting execution (the *execution trace*).
- Provide a bottom-up semantics for task achievement and quality.
- Provide a semantics based on the evolving state of execution: what matters is previous history, not what might happen in the future.
- Include only what is necessary to support the modeling constructs defined.

1.3 Scope and Limitations

- There will be no costs in Phases 1 and 2. Costs may be introduced at a later time.

Chapter 2: C_TAEMS Model

This is mostly here to hang the meta-C_TAEMS discussion on. But we put it first to introduce terminology and provide something concrete.

2.1 Task Structure

A C_TAEMS *task structure* defines the problem to be solved. There is an *objective* task structure, which is the "real" view of the problem, seen from an omniscient observer's perspective. There are also *subjective* task structures, one for each agent. Subjective task structures will differ from the objective task structure in that they are incomplete, containing only what the local agent knows. It is also possible for a subjective task structure to be inconsistent with the objective task structure, but for now we will not introduce those complications. Objective and subjective task structures have the same basic elements, presented here at a high level, discussed in more detail in Chapter 6.

C_TAEMS supports the following relations on nodes:

- *Subtask links* are directed edges from tasks to nodes that are *subtasks* of those tasks. See Section 3.3 for what that means. Subtask links define a tree of task/subtask relationships.
- *Nonlocal Effects* (NLEs, also sometimes-but-not-here called interrelationships) are structures relating one node to another, indicating that the performance of the target node is dependent in some way on the execution of the source node. See Section 3.4.

A node that is not a subtask of any task is called a *root task*. By definition, there will be at least one root task in any task structure, but there may be more than one.

For each task structure, one root task has a special status as the *task group*. This is the task whose execution quality determines the quality of a given *execution trace* (Chapter 4).

Chapter 3: Semantics

3.1 Preliminaries

In this section we define the number model and a few other things we need to get started.

Time: Time values $t \in \mathbf{N}$ range over the natural numbers (non-negative integers). Individual time values are also referred to as *ticks*. We define time to start with 0, and to end at a special number **EOH** $\in \mathbf{N}^+$ (the positive integers), defined as part of the problem scenario. The first simulation tick is 1.

Performance metrics: Currently, there are two defined: cost and quality. Metrics $c,q \in \mathbf{R}^+ \cup 0$ range over the non-negative real numbers. (**Note: cost will not be used during Phases 1 and 2**).

Activity execution: We refer to *activity execution a* having the following attributes:

- $\text{start}(a) \in \mathbf{N}^+$ denotes the start time of a.
- $\text{end}(a) \in \mathbf{N}^+$ denotes the end time of a. $\text{end}(a) > \text{start}(a)$.
- $\text{cost}(a) \in \mathbf{R}^+ \cup 0$ denotes the cost of a.
- $\text{qual}(a) \in \mathbf{R}^+ \cup 0$ denotes the quality of a.

An activity execution a is the executed instantiation of a *method*, about which more later.

Execution traces An *execution trace* denotes a set of activity executions $e=\{a_1,a_2,...,a_n\}$.

3.2 Methods

3.2.1 Methods and Method Outcomes

A C_TAEMS method m has the following attributes:

- $\text{name}(m)$ is a symbol, denoting a unique name for the method.
- $\text{agent}(m)$ is a symbol, denoting the agent who *owns* this method. Method ownership means that the agent controls the method's execution, and receives status updates from the simulator regarding that execution. See Chapter 4 for details.
- $\text{outcomes}(m)$ is a discrete distribution $\{(o_1,p_1),(o_2,p_2),...,(o_k,p_k)\}$ over possible method outcomes for m. $\sum_{i=1}^{k} p_i = 1$.

A method outcome o_i consists of a set of discrete distributions, one for the *duration* of m, defined as

$$\text{duration}(m)=\text{end}(a_{m,e})-\text{start}(a_{m,e})$$

and one for each of the associated performance metrics, as discussed in Section 3.1. These distributions are used by the simulator to determine the corresponding values for executions of *m*, in other words for activity executions $a_{m,e}$ denoting the execution of *m* in some execution trace *e*. The duration distribution is over positive integers, the others over the non-negative reals.
For a given execution trace *e*, the relation between $a_{m,e}$ and *m* is bijective. Each activity execution is an execution for only one method, and each method has at most one activity execution in any given execution trace.

Please note that the mathematical definition of a discrete probability distribution is a function. Consequently, each value for which a probability is specified (o_i, in the distribution given above) must be unique. This restriction applies everywhere in this document that we discuss discrete distributions, i.e. everywhere the term "distribution" appears in connection with a discrete set of values (duration, cost, outcome, etc.).

3.2.2 Constraints on Methods

C_TAEMS includes temporal constraints on methods. The constraints supported are:

- Release time: $\text{release}(m)\in\mathbf{N}$
- Deadline: $\text{deadline}(m)\in\mathbf{N}$

From the subtask hierarchy above *m*, we define the *maximally restrictive* temporal constraints $release^*(m)$, $deadline^*(m)$ to be the maximum (resp. minimum) of $\text{release}(m)$ (resp. $\text{deadline}(m)$) and any release (resp. deadline) constraint on any task in the subtask hierarchy above *m*. See Section 3.3.2 for a more formal definition.

The interpretation of a release time constraint is that any activity execution a_m of *m* such that

$$\text{start}(a_m) < \text{release}^*(m)$$

will *fail*, meaning that $qual(a_m) = 0$, with duration and cost computed as normal. See Chapter 4.

The interpretation of a deadline constraint is that any activity execution a_m of *m* such that

$$\text{end}(a_m)>\text{deadline}^*(m)$$

will fail, in exactly the same sense.

3.2.3 Computing Method Expectations

We assume all outcome probabilities to be independent, within and across methods. For a single method with multiple outcomes, the distributions on quality, cost, and duration may not be independent.

Probabilities and expectations for all performance metrics are computed the same way. Given method *m* with outcome distribution $\{(o_1, p_1),(o_2, p_2),...,(o_k, p_k)\}$, and each outcome o_i having a quality distribution $\{(q_{i,1}, p_{i,1}),(q_{i,2}, p_{i,2}),...,(q_{i,l}, p_{i,l})\}$, the probability that *m* will execute with quality *q* is computed as

$$P(q)= \sum_{i,j} p_i * p_{i,j} \ : \ q_{i,j}=q$$

We compute the expected quality $E(q)$ in the obvious way:[1]

$$E(q)= \sum_{i=1}^{k} \sum_{j=1}^{l} p_i * p_{i,j} * q_{i,j}$$

Duration is computed in exactly the same way.

3.3 C_TAEMS Tasks

C_TAEMS *tasks* are abstract constructs that aggregate other nodes (tasks or methods), hereafter collectively called *subtasks*. The temporal extent of a task execution in a given execution trace is defined by the execution of its subtasks. The quality of a task execution is computed from the quality of the subtask executions, using a *Quality Accumulation Function* (QAF).

3.3.1 Task Definition

A C_TAEMS task *T* has the following attributes:

- name(*T*) is a symbol, denoting a unique name for the task.
- subtasks(*T*) is a set of nodes (tasks and methods).
- qaf(*T*) is the quality accumulation function for *T*. Possible values for qaf(*T*) and their semantics are defined in Section 3.3.4.

For now, we assume that task/subtask relationships form a tree. We define the following shorthand:

$$\text{supertasks}(T) = \{S \mid T \in \text{subtasks}(S)\}$$

Despite the assumption above, supertasks(*T*) denotes a set. We do this for two reasons. First, because the assumption that each node is a subtask of at most one task may in the near future be relaxed. Second, because the set will be empty for root tasks, and this is notationally convenient.

The overall quality of an execution trace is computed by recursively computing the quality of the *task group*, a specifically-defined root task. There may be other root tasks

[1]Computing expectation over a discrete distribution is an odd thing to be doing, of course. For example, it is entirely possible that $P(q=E(q))=0$.

which do not figure directly in the quality of the execution trace, and so it is not correct to use an "implicit Sum" or other means to gather the quality of an unconnected set of task hierarchies.

3.3.2 Constraints on Tasks

C_TAEMS includes temporal constraints on tasks. The constraints supported are:

- Release time: release(T) ∈ **N**
- Deadline: deadline(T) ∈ **N**

These constraints are defined to apply recursively downward: a subtask inherits the most restrictive of its own constraints and those of its parent(s). For a node n:

$$\text{release}^*(n) = \max(0, \text{release}(n), \max_{T \in \text{supertasks}(n)} (\text{release}^*(T))$$

$$\text{deadline}^*(n) = \min(\text{EOH}, \text{deadline}(n), \min_{T \in \text{supertasks}(n)} (\text{deadline}^*(T))$$

As previously discussed, these are then the temporal constraints applied to method execution. Very informally, the effect of this is that node n will accumulate quality only from methods whose execution is entirely within the interval defined by release*(n) and deadline*(n).

3.3.3 Task Executions

For a given execution trace e, we define a *task execution* of task T recursively, in terms of the task executions for all $S \in$ subtasks(T). When S is a method, this grounds out with the activity execution, as defined in Section 3.2. C_TAEMS is propositional, so there is no ambiguity regarding task/subtask relationships. Any method that appears in an execution trace is a leaf on a fixed path through the task/subtask hierarchy, all the way up to a root task.

- Active(T,t) – T has started execution at or before t.
- quality(T,t) is the quality accumulated so far by the execution of node T at tick t.

Task execution and quality accumulation over execution traces can be viewed as a set of transitions over time. At the lower level of method execution, that is how we define them. At every level above method execution, execution status and quality are defined in terms of other functions of the current tick, with the single exception of **Started(T,t).**

Notation

- n is a node
- *isTask(n)* is a Boolean that returns *TRUE* only if node n is a task node
- *isMethod(n)*is a Boolean that returns *TRUE* only if node n is a method node

- *I{.}* is an indicator function which returns 1 if the condition within {.} is *TRUE* and 0 otherwise
- *M* is a meta-CTAEMS event
- *occurs(M)* is the tick when the meta-CTAEMS event occurs
- *type(M)* is the type of the meta-CTAEMS event
- *descendants(n)* are all nodes which are part of the subtree of *n* via the *subtask()* relationship: $\{m: (m \in subtasks(n)) \vee (\exists\{n_1,\ldots,n_k\}$ s.t. $n_1 \in subtasks(n), n_2 \in subtasks(n_1), \ldots, n_k \in subtasks(n_{k-1}), m \in subtasks(n_k)\}$.

Active, Started, Starts, NumberActive

We modify the definition of *Active(n, t)* to mean that node *n* is executing at time *t*. *Started(n, t)* is introduced and *Starts(n)* is modified such that its usage in the definition of the SyncSum QAF is valid. *NumberActiveChildren(T, t)* is introduced for use in the ExactlyOne QAF definition.

- Initial Conditions:
  - A1. Started(n, 0) = False, ∀n
- Task Execution Status:
  - T1. $Active(T, t) = \bigvee_{S \in subtasks(T)} Active(S,t)$
  - T2. ¬Active(T, t-1) ∧ Active(T,t) ⇒ Started(T,t) = TRUE
  - T3. Started(T,t) ∧ (t < EOH) ⇒ Started(T,t+1)
  - T4. ¬Started(T,t) ∧ ¬Active(T,t+1) ∧ (t<EOH) ⇒ ¬Started(T,t+1)
  - T5. ¬Active(T, t-1) ∧ Active(T,t) ∧ ¬Started(T,t-1) ⇒Starts(T) = t
  - T6. ¬Started(T,EOH) ⇒ Starts(T) = EOH
- Method Execution Status:
  - T7. Active(m,t) = TRUE ⇔ $start(a_m) \le t < end(a_m)$
  - T8. $quality(m,t) = I_{t \ge end(am)} qual(a_m)$

  a_m is the activity execution for method *m* in the current execution trace and $qual(a_m)$ is the result from the sampling of a quality distribution of method m. T3 through T6 also apply to methods.

- New State for Tasks:
  - D1. NumberActiveChildren(T,t) = |{S∈subtasks(T): Active(S,t)=TRUE}|

3.3.4 QAFs

Quality Accumulation Functions are used to compute the quality of tasks (and ultimately overall quality) for a specific execution trace. C_TAEMS supports the following qafs:

- Sum
- Max
- Min
- SyncSum
- SumAnd
- ExactlyOne

Each of these is defined below. These qafs are currently defined in such a way that quality(T,t) is a monotonically nondecreasing function of t. That can be changed without breaking things in fundamental ways, but will require some care, and some modification to how NLEs are handled in addition to the revised definitions here.

Sum

$$quality(T,t) = \sum_{T' \in subtasks(T)} quality(T',t)$$

Max

$$quality(T,t) = \max_{T' \in subtasks(T)} quality(T',t)$$

Min

$$quality(T,t) = \min_{T' \in subtasks(T)} quality(T',t)$$

SyncSum

$$quality(T,t) = \sum_{T' \in subtasks(T) \wedge Starts(T')=Starts(T)} quality(T',t)$$

Under the current model of execution, all but one of the subtasks will necessarily be non-local in any agent's subjective model, because the local agent has a single thread of execution.

SumAnd

If isTask(n) = TRUE and qaf(n)=SumAnd, quality(n,t) =

$$I_{\{quality(S,t)>0, \forall S \in subtasks(n)\}} \sum_{S \in subtasks(n)} quality(S,t)$$

ExactlyOne

If isTask(n) = TRUE and qaf(n)=ExactlyOne, then quality(n,t) =

$$I^{DC-MPQC} \; I_{\{\forall u: u \le deadline^*(n), NumberActiveChildren(n,u) \le 1\}} \max_{S \in subtasks(n)} quality(S,t)$$

where

$$I^{DC-MPQC} = I_{\{t \ge deadline^*(n)\}} \; I_{\{|\{S \in subtasks(n): quality(S,t) > 0\}| = 1\}}$$

3.4 NLEs

C_TAEMS includes the following *non-local effects* (NLEs):

- Enables
- Disables
- Facilitates
- Hinders

Each NLE has a *source* node and a *target* node. The NLE affects the quality, cost, and duration of the target, depending on the state of the source, and the delay of the NLE, when the target **Starts**. The specific effect depends on the type of NLE.

An NLE *r* of any of these four types has the following attributes:

- type(*r*)∈{Enables,Disables,Facilitates,Hinders}
- source(*r*) is the source node for *r*
- target(*r*) is the target node for *r*
- delay(*r*)∈**N** is the delay from the time that source(*r*) changes status until that change is apparent at target(*r*).

Facilitates and Hinders NLEs have other attributes as well, described below.

If the target node for *r* is a method, *r* applies to that method. If the target node is a task *T*, *r* is interpreted as applying individually to each of the methods in the subtask hierarchy below *T*, according to the objective model. In particular, the time to be used in evaluating the NLE's activation status for each method is the start time of the individual method, not of the task as a whole. This also means that NLE effects on target quality, cost, and

duration as described below are computed by the simulator, and applied to the activity execution.

3.4.1 Enables

For an NLE r, if type(r) = Enables, n_s = source(r), n_t = target(r), and isMethod(n_t)=TRUE, then

$$\text{quality}(n_s, \max\{0, \text{start}(n_t) - \text{delay}(r)\}) = 0 \Rightarrow \text{quality}(n_t, \text{end}(n_t)) = 0$$

For an NLE r, if type(r) = Enables, n_s = source(r), n_t = target(r), and isTask(n_t)=TRUE, then it is equivalent to having Enables NLEs to all method node descendants of n_t using the above definition.

3.4.2 Disables

For an NLE r, if type(r) = Disables, n_s = source(r), n_t = target(r), and isMethod(n_t)=TRUE, then

$$\text{quality}(n_s, \max\{0, \text{start}(n_t) - \text{delay}(r)\}) > 0 \Rightarrow \text{quality}(n_t, \text{end}(n_t)) = 0$$

For an NLE r, if type(r) = Disables, n_s = source(r), n_t = target(r), and isTask(n_t)=TRUE, then it is equivalent to having Disables NLEs to all method node descendants of n_t using the above definition.

3.4.3 Facilitates

Facilitates and Hinders have some additional attributes:

- qDist(r) is a discrete distribution over quality coefficients qf, $0 \le qf \le 1$.
- cDist(r) is a discrete distribution over cost coefficients cf, $0 \le cf \le 1$.
- dDist(r) is a discrete distribution over duration coefficients df, $0 \le df < 1$.

In addition, we need to describe the *proportional quality* of r at tick t:

$$\text{pq}(r, t) = \min(1, \text{quality}(\text{source}(r), t) / \text{MaxQ}(\text{source}(r))) \quad \text{when MaxQ(source(r))} > 0$$
$$\text{pq}(r, t) = 1 \quad \text{when MaxQ(source(r))} = 0$$

The MaxQ of a task T is recursively defined from the MaxQ of the subtasks. The exact form depends on qaf(T). For Sum, SumAnd, and SyncSum:

$$MaxQ(T) = \sum_{T' \in subtasks(T)} MaxQ(T')$$

For Max and ExactlyOne:

$$MaxQ(T) = \max_{T' \in subtasks(T)} MaxQ(T')$$

For Min:

$$MaxQ(T) = \min_{T' \in subtasks(T)} MaxQ(T')$$

MaxQ(m) for a method m is defined to be the maximum value appearing in any quality distribution in any outcome of m. Note that MaxQ is *not*, in general, the maximum quality of a cTAEMS node. The latter quantity is not efficiently computable. MaxQ is an efficiently-computable analog of maximum quality.

Facilitates affects quality, cost, and duration, all in (almost) the same way. Given a Facilitates NLE r with source T_1 and target T_2, if T_2 **Starts** at tick t and T_1 has positive quality at t-delay(r), then the quality computation for T_2 at any tick $t' \geq t$ is modified as follows:

$$quality'(T_2, t') = quality(T_2, t')(1 + qf * pq(r, t - delay(r)))$$

where qf is a quality coefficient sampled from qDist(r). In other words, the Facilitates NLE r results in adding a percentage markup to the reported quality for T_2.

Cost and duration effects are defined similarly but with a sign change, and duration is coerced to be an integer, using the ceiling function:

$$\cos t'(T_2) = \cos t(T_2)(1 - cf * pq(r, t - delay)))$$

$$duration'(T_2) = \lceil duration(T_2)(1 - df * pq(r, t - delay))) \rceil$$

3.4.4 Hinders

The Hinders NLE functions exactly the same way as Facilitates does, but the signs of the effects are changed: quality is diminished, cost and duration are increased.

3.4.5 Multiple NLEs on the Same Target

With these definitions, NLEs are cleanly compositional. If any Disables is **Enabling**, the target fails. If any Enables is not **Enabling**, the target fails. Any Hinders or Facilitates that are **Enabling** can be applied in any order, achieving the same effect (because multiplication is commutative).

3.4.6 Coefficient Sampling

For an NLE r, where type(r) = Facilitates or type(r) = Hinders, when choosing qf(r), df(r) from qDist(r), dDist(r) respectively, the distributions will only be sampled once during a trial run of a scenario. If the target of r is a task then the implied NLE r_d which points from source(r) to d ∈ descendants(target(r)) will have identical NLE coefficients:

isTask(target(r)) = TRUE ⇒ qf(r_d) = qf(r), df(r_d)=df(r) ∀d∈descendants(target(r))

The cost coefficient cf(r) chosen from cDist(r) will not be used in Phase 2.

3.4.7 Outcome-based NLEs

For this to be accepted it is necessary that method outcomes are visible event parameters, i.e., agents should be told the name of the outcome that a method achieves (in addition to the quality and duration) when the method completes. This requires changes to the Phase 1 structure of descriptions of NLEs in cTAEMS files.

For an NLE r to be outcome-based, we require isMethod(source(r)) = TRUE. Given O^r_s,

a set of outcomes for the NLE source method s = source(r), NLE r "fires" for a subset of outcomes $O^r_s \subseteq O_s$. Let the outcome of a method be o_s. If $o_s \in O^r_s$, then the NLE r "fires" and the specified rules of the NLE follow. If $o_s \notin O^r_s$, then the NLE r does not "fire" which implies that the target sees the quality of the source as zero, regardless of the actual quality received by the source. This has the foolowing implications:

- If type(r) = Disables, target(r) is not disabled.
- If type(r) = Enables, target(r) is not enabled.
- If type(r) = Facilitiates, target(r) is not facilitated.
- If type(r) = Hinders, target(r) is not hindered.

Essentially, if the outcome is outside O^r_s, it is as if the NLE did not exist, except in the case of an enables NLE where getting an outcome outside O^r_s forces the quality of the target to zero.

3.5 C_taems Schedules

A *schedule S* represents an intent to execute some set of methods.

3.5.1 Definition of a Schedule

A C_TAEMS *schedule S* represents an intent to execute some set of methods. Within Coordinators, schedules are needed for the following purposes:

- The initial problem posed to a Coordinator society includes an initial schedule for each Coordinator.
- ? ?

There is no requirement that Coordinators use C_TAEMS schedules as an internal representation, to pass around between themselves, nor even as the method for communication with their human counterparts, if any. Since performance is judged based on execution, schedules are not needed as output.

Each Coordinator's execution is a single thread. Executing methods may be terminated via an *abort* event, but may not be restarted. A schedule *S* for a single Coordinator consists of:

- A sequence of activities $\langle A_1, A_2, ..., A_l \rangle$, such that each A_i is executed in turn.
- A start time $start(S) \in \mathbf{N}^+$, which is the intended start time of A_1.

Each activity *A* in *S* has the following attributes:

- method(*A*) is the method whose intended execution is represented by *A*.
- start(*A*) is the intended start time of *A*.

The intended meaning of the start time is that if the activity preceding *A* completes before start(*A*), execution of *A* will be delayed until that tick. If that activity completes after start(*A*), execution of *A* commences immediately.[2]

[2]This is an "intended" meaning because this schedule is being handed to the Coordinator, which is free to execute methods in any order, at any time.

The Coordinator may also know about non-local tasks and methods, i.e., tasks and methods appearing in the schedules and task structures of other agents. Non-local tasks and methods do not appear in the sequence of activities, because they are not executed by this agent.

3.5.2 Commitments

There will be eventually commitments in the initial schedule, but not in years 1 and 2 (2005, 2006). Both semantics and syntax are TBD.

Chapter 4: TAEMS Execution Model

4.1 Interface between agents and a simulator

The multi-agent agent simulator (MASS) is the only simulator that enforces the execution model of TAEMS. The Coordinator program will use a new version of MASS built by GITI. This chapter describes what to expect when a Coordinator agent uses this new version of MASS. Something in the Coordinator testbed will generate the TAEMS task structure that the simulator will enforce. The agents will receive a part of this task structure, as their *subjective model*.

The interface between the agent and the simulator is defined as follow:

- The agent sends an execution request to the simulator to start executing a specific method immediately.
- The simulator sends a message to the agent indicating whether the method was successfully started or not, at the next tick.
- When the execution is completed, the simulator sends back the quality, cost and duration generated by the method.
- There is no *status* interface defined, it's a black box execution, the agent has no way of knowing if the execution is progressing and how long it has before completion.
- An agent can stop an execution by sending an abort command and in this case the simulator will reply with a message indicating whether the abort was successful or not, at the next tick.

In a task structure, what can be executed?

The answer is quite simple: only local methods. Methods represent what your agent can actually do.

How does it work?

The agent sends an execution request to the simulator (see section **Error! Reference source not found.** for the exact message). When the simulator receives it, it draws the method outcome that will be simulated (using the outcome distribution). From the selected outcome, the simulator draws the quality, cost and duration that this execution will take. When the method is completed, then the simulator sends back to the agent the final cost, quality and duration. At that point, the simulator will automatically *activate* any outgoing NLEs [3] . The last point is to propagate in the task structure the quality using the quality accumulation function defined in the structure. MASS computes it in its global task structure and the agent should do the same in in its *subjective* view.

The following picture (Figure 4.) shows the result after execution. Note the quality accumulation value of 30.0 showing up in the task Prepare after the completion of the method *Review_Slides*.

[3]If any NLE has a delay specified in its activation, the simulator will draw the expected delay and enforce it before activating the NLE.

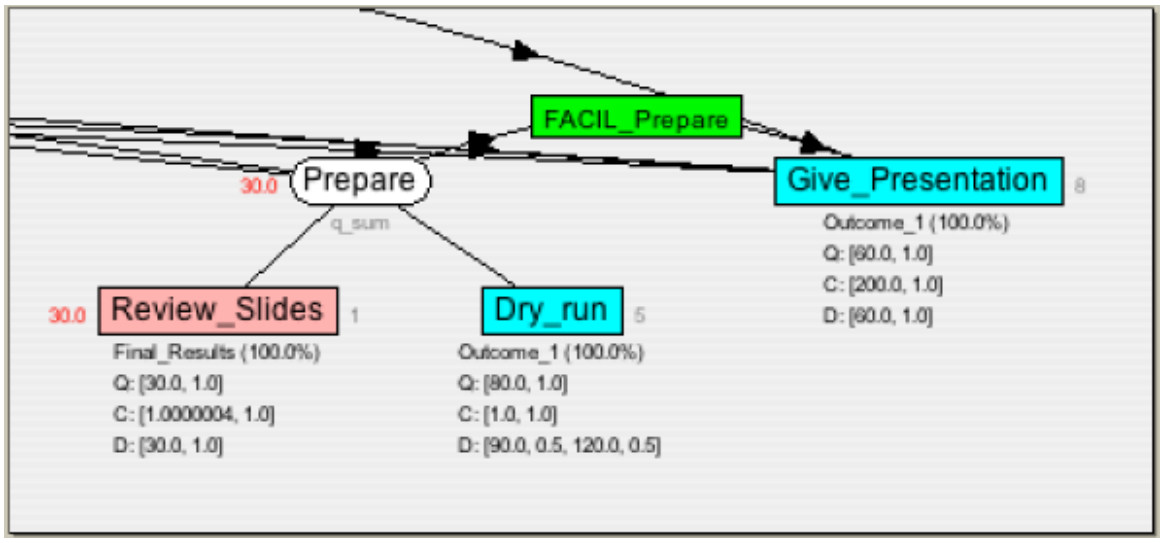

Figure 4.1: Detail on the quality accumulation function after one execution. Note that the facilitate relationship *FACIL_Prepare* is now active and will facilitate the method *Give_Presentation*.

How does my agent get results back?

When the execution is completed, the simulator sends a message back to the agent who has started the execution. An important note, neither the simulator nor the TAEMS execution model specify anything about sharing data about execution results. If an agent A has a representation in its task structure of another agent (B) method (as a non local method), the simulator will **NOT** send anything to the agent A when the agent B finishes the execution of the method. It is the responsibility of agent B to explicitly share this information with A.

Is an execution failure identical to getting a quality of 0 as an execution result?

Yes, a failure is represented in TAEMS with a 0 quality.

What quality, duration and cost can I expect from executing a method?

The method's duration, cost, and quality in execution will be derived from the "objective" (i.e. the one the simulator has) task structure, which must include tasks as well as methods, because method quality is dependent on NLEs that are only explicit if you have the entire task structure. When executing a method the simulator will select values according to the different characteristic distributions. It is important to remember that NLEs will affect the method distribution (see Section 3.4).

If I execute a method more than once, do I get more quality?

No. It will fail to execute the second and successive times.

Can I get negative quality when executing a method?

No, quality has only strictly non-negative values.

Can I execute a method that is not enabled?

Yes, but the execution will always fail. Executing a method not enabled, will take time to complete and have some cost but will always get 0 quality.

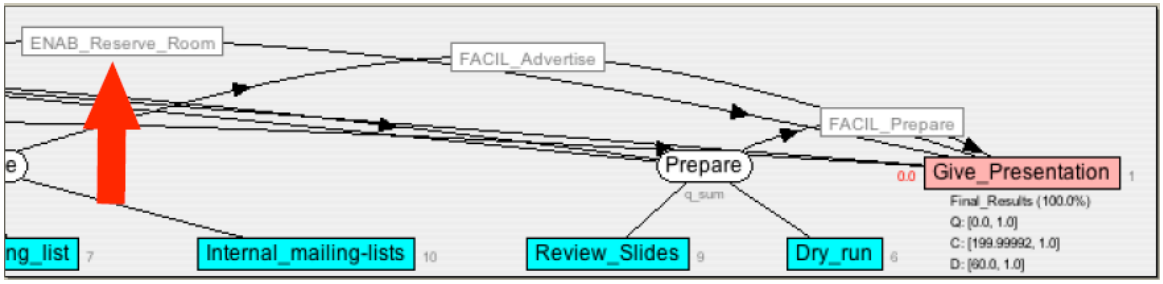

Figure 4.2: Method Give_Seminar was executed before any prerequisite method. It fails with quality 0.

Can I execute a method that belongs to another agent?

No.

If my method fails, can I re-execute a method already executed?

No, we don't allow multiple executions of the same method.

Can I execute a method earlier than released?

Yes, but the method will always fail.

Where is the total cost of the task?

The cost of a task is the sum of all the subtask costs.

What happen if I exceed my budget?

Nothing. There is no budget or maximum allowance defined in TAEMS. Cost is a dimension the agent are trying to minimize but there is no enforcement done for the cost. Some TAEMS scheduler, let you specify the maximum cost an agent is willing to spend on a task.

Can an agent tell MASS to start executing another method before the previous method has completed?

No, first you must abort the currently-executing method.

4.2 Monitoring

Query

Abort

Can an agent stop executing a method?

Yes, but you will forfeit any quality and pay the full cost for it. Suppose that a method execution has a coinflip quality of 40, a coinflip duration of 10 and a coinflip cost of 40. If after 5 ticks, the agent stops the execution, then the simulator will return quality of 0, duration of 5 and cost of 40.

Can the method be resumed at a later time?

No.

Failure

If an method execution fails, the simulator will return a quality of 0. In general, if a method execution returns a quality of 0, then there was something wrong with this execution.

Chapter 5: Meta-C_TAEMS

> "There are more things in heaven and earth, Horatio,Than are dreamt of in your philosophy."
> –Shakespeare, Hamlet, Act I

We define the extra-linguistic concepts and constructs needed to support dynamic changes to both the problem definition and execution (hereafter collectively referred to as "meta-C_TAEMS").

For the initial release, allowed meta-C_TAEMS events will be limited to:

1. unmodeled method failure (aka "unpredictable failures"),
2. changes to node temporal constraints referring only to future times,
3. the addition of new task structures (see Chapter 6), and
4. changes to the outcome quality/cost/duration distributions for methods that have not yet started execution.

5.1 Partial Execution Trace

There is a *current time tick* t_{now}, which we previously didn't need to talk about because at the beginning of a scenario t_{now}=0.[4]

Meta-C_TAEMS changes take place in the context of a *partial execution trace*. For a single agent, a partial execution trace consists of a prefix of activity executions (Section 3.1), each specifying the start time, end time, and outcome performance metrics for some method in the agent's current task structure. For all activity execution s *a* in the execution trace, start(a) and end(a) must be less than or equal to t_{now}. There may also be some method that has started execution, but not yet completed.[5]

An execution trace, partial or otherwise, is a construct that describes a set of events occuring in the simulator. There is no prescribed representation (or even a requirement that there *be* a representation) for a partial execution in any Coordinator, though the Coordinator will receive notification from the simulator for the events in the respective single-agent execution trace.

We may also talk about the *scenario execution trace*, meaning the combined execution traces of all the Coordinators in the scenario. This is the simulator's view of execution.

[4]The first simulator tick updates the time to 1.
[5]Unless the single-thread-of-execution assumption is dropped, there will be at most one executing method for each agent.

5.2 Rescheduling

Rescheduling happens in the context of a partial execution trace. The representation of the previous schedule, the algorithm used to derive the new schedule, and how the new schedule will be represented and implemented are all the responsibility of the individual teams. Interaction with the simulator is only in terms of the events described in Chapter **Error! Reference source not found.**.

5.3 Unexpected Execution

Here are some possible forms of unexpected execution.

5.3.1 Unexpected Method Outcomes

The agent finds out about method execution through an event returned by the simulator. Execution parameters (duration, quality, and cost) are currently determined by coin-flips (Section 3.2), possibly modified by activated NLEs (Section 3.4). Duration is used to schedule an event to be sent to the agent, in which will be communicated the resulting cost and quality (Chapter **Error! Reference source not found.**).

Any unexpected outcome must be in terms of the values of one or more of these parameters.

Method failure is currently defined as incurring the normal cost and duration, but quality 0. Methods can fail because their execution extends outside the bounds of their temporal constraints or because of the effect of an NLE.[6]

With one exception, method outcomes will always be drawn from the modeled outcome distributions, modified at execution as appropriate based on temporal constraints and NLE activations. That exception is that the method may fail, in exactly the above sense, without that being the result of an explicitly modeled outcome.

5.3.2 Abort

As described in Section 4, an agent can stop an executing method by sending a "MethodAbort" command Chapter **Error! Reference source not found.**). The simulator will reply with a MethodAbort message at the next tick. The method will get 0 quality, and the full execution cost..

This is something you do to yourself, though it is unmodeled.

[6]It is also possible for 0 quality to be a modeled outcome, with associated distributions for duration and cost.

5.3.3 Modifying the Execution Trace

One possible source of unexpected behavior would be for the agent to find out that what they expected to happen, and thought had happened, is not what happened. This occurs all the time in the real world. It can occur in Coordinators, too. The individual agents have only a partial view of execution, and so their expectations on what happened in other agents' executions can be confounded.

There will be no explicit support for revising existing execution traces, as simulated and subsequently reported to the agents.

5.4 Changes to the Task Structure

5.4.1 Adding New Tasks

New tasks, methods, and NLEs will be added to the objective view during the simulation. Agents will be notified of additions and consequent visible-to changes that are visible to them, using visibility rules defined in Section 6.3 See Section 7.9 for relevant syntax.

If *type(M) = AddTask*, then

- If the new task and descendant nodes are to be a subtask of an existing task node *n (IsTask(n) = TRUE)*, then the addition is permissible
    - At any time, *occurs(M)*, for *qaf(n)* ∈ *{Max, Sum, SyncSum, ExactlyOne}*
    - If *release*(n) > occurs(M)* for *qaf(n)* ∈ *{Min, SumAnd}*
- Nodes in the new substructure can be sources of an NLE *r*, if *release*(target(r)) > occurs(M)*.
- Nodes in the new substructure can be targets of an NLE *r*.

5.4.2 Changing Temporal Constraints

Changing temporal constraints (deadlines and release times) for tasks and methods in the existing task structure that have not finished executing does not require any semantic changes, though we need a syntax for communicating those changes.[7]

Changing temporal constraints referring to tasks and methods that have finished executing is problematic, because NLEs can affect method duration. So, if temporal constraints can be revised back in time, this may have the result of editing the existing execution trace, changing not just quality and cost, but how long a method execution actually took (you only *thought* you saw when it ended).

[7]Changing a temporal constraint so as to fail a currently-executing method will also require additional changes to the MASS simulator.

This should be avoided. We permit changes to temporal constraints on tasks and methods, only when both the old and new time value of the constraint is $\geq t_{now}$.

5.4.2.1 Release Events

If *type(M) = Release*, then *release(n)* may be changed to *u*, if *release*(n) > occurs(M)* and $u \geq occurs(M)$.

5.4.2.2 Deadline Events

If *type(M) = Deadline*, then *deadline(n)* may be changed to *u*, if *release*(n) > occurs(M)* and $u \geq occurs(M)$. **NOTE:** this allows *release(n) > deadline(n)*.

5.4.2.3 Distribution Events

If *type(M) = Distribution*, then outcome, quality, and duration distributions for a method *m* may be changed, if *release*(m) > occurs(M)*.

5.4.3 Editing Existing Task Structures

There are several ways one might edit existing task structures:

- Change task/subtask relationships
- Change NLEs
- Change QAFs

In relation to future execution, these are all straightforward as long as the resulting structures still obey the C_TAEMS definition.

Equally, all of them suffer from the problem described above for changing temporal constraints referring to past execution: it is possible to rewrite past executions in ways that falsify simulated and "observed" behavior in counter-intuitive ways.

For now, at least, rather than distinguish between past and future task structure, we prevent all editing of existing task structure, with the exception of adding of new task structure as described immediately above.

5.4.4 Changes to Method Outcome Distributions

Changes to method outcome distributions are supported, for methods that have not started execution. These changes must obey the current assumptions regarding agreement between the subjective and objective models, and will be accomplished through explicit notification events, to the agents in the visible-to sets of those methods.

Chapter 6: Task Structures, Assignments, and Limited Knowledge

Related to the spec, though not necessarily *part* of the spec, we need to determine how task structures are employed, where they are represented, and the relations between those representations. Specifically, we need to specify the way C_TAEMS is used to describe:

- Who can do what (agent capabilities)
- Who is supposed to do what (agent assignments)
- Who knows about what (view of the task structure).

Task structures are functionally overloaded. They serve as an execution model for the world (executed by the simulator, in this case), as a description of local knowledge (loosely speaking, the *subjective task structure*), and can be viewed (though we will choose not to) as representing task assignments, in the sense that if you have certain tasks in your task structure, there is some onus on you to execute them. We can even talk about a sense of community knowledge about how the world works, by taking the union of what is locally known over a community of agents.[8]

6.1 Definitions and Assumptions

We will use the following terms:

- The *objective task structure* is the task structure held by the simulator, and used by the simulator to determine the quality associated with an execution trace.
- The *subjective task structure* is the task structure held by any individual agent. It represents what the agent can do, and some, most likely limited, view of the capabilities of other agents as well.

More details regarding the properties of both the subjective and objective task structures are provided below.

We will for now make the following assumptions, regarding task structures:

- Subjective task structures are *consistent* with objective task structures. Although in general the subjective task structure may be only loosely tied to the corresponding objective task structure, for now we assume them to be the same. In other words, the subjective task structure is a perfectly accurate representation of the part of the objective task structure it corresponds to. [9]
- Objective task structures contain a *task group* (Section 3.3), which is used to compute the quality of an execution trace. They may contain other root tasks as well.

[8]There is no particular reason to assume that this combination of local information will be equivalent to, or even consistent with, objective reality.

[9]As a specific point: the method outcome distributions will be the same, if present (you might not know distributions for nonlocal methods, but if you have them they will be correct).

- Global execution quality is defined to be the final (post-execution) value of the QAF of the objective task group.

6.2 Objective Task Structures

There is a single, global view of the methods that can be executed for a given scenario, and of the task structure that will be used to evaluate the quality of the resulting execution trace.

6.2.1 Methods Executable by Multiple Agents

One issue that has repeatedly arisen is the possibility of having methods that can be performed by multiple agents. The simplest approach to this is to replicate the methods, one for each possible performer, and this is the approach we will take for now.

An alternative model to be investigated for the future is one where a single method can appear in the task structure with a set of possible agents that could execute it. This has some nice properties, but is not obviously preferable to the single-agent-per-method model, and opens enough of a can of worms to be deferred for now.

6.2.2 Tasks "Executable" by Multiple Agents

Task structures are abstractions, not executable. We permit shared task structure (including identity of task names) between agents.

6.3 Subjective Task Structure

Each agent has a subjective task structure, which is consistent with the objective task structure, but may contain less information. For now, this can occur in exactly one way:

- There may be information in the objective task structure not in the subjective task structure.

The relationship between the objective model and each agent's subjective model is described in Section 6.3.1.

An agent can execute any method in their task structure for which they are the owner. How they decide which ones to execute is up to them.[10]

6.3.1 How Subjective Task Structures are Constructed

Each node and NLE in the objective task structure will have a visible-to field, which will be a set of agent names determined from the objective task structure as follows:

Let agents-below(node) be the set of agents that own methods that are descendants of the indicated node, i.e.

[10]The assignment question (who is supposed to do what) has apparently traditionally been folded in with the task structure, but we're not going to do that.

1. If N is a method, agents-below(N) is owner(N).
2. If N is a task, agents-below(N) is the union of agents-below(S) for each node S that is a subtask of N.

A node is visible to all agents in its agents-below set; the source of an NLE is visible to all agents in the target's agent-below set; the target of an NLE is visible to all agents in the source's agent-below set; an NLE is visible to all agents in the agents-below sets of its source and target.

3. For each node N, visible-to(N) is the union of agents-below(N) and agents-below(N') for each node N' that is the source/target of an NLE that has target/source N.
4. For each NLE N, visible-to(N) is the union of agents-below(source(N)) and agents-below(target(N)).

Subjective Task Structures

For all tasks and NLE's N, an agent A will have N in it's subjective view iff A is an element of visible-to(N). If a node N is in an agent A's subjective view, then the agent will know

1) the name of N
2) whether N is an NLE, a task, or a method.
3) The names (and attributes, if any) of agents to which N is visible
4) If N is a method, the owner of the method
    a. Method outcomes
    b. Method constraints
5) If N is a task
    a. the QAF of N, but not (necessarily)the number or names of the children of N.
    b. whether N' is a subtask of N, for all tasks N' in A's subjective view (i.e., A will know of sub-task relationships that exist in the objective view iff both the parent and child are in A's subjective view)
6) if N is an NLE,
    a. the type (subtype) of the NLE, i.e., whether N enables, disables, facilitates, or hinders
    b. both the source and target for the NLE
    c. the delay
    d. the power distributions for facilitates and hinders NLEs
7) The attributes of the node, if any. Note that objective views given to the simulator may contain attributes that are only for simulator use (e.g., EVA_* attributes). Such attributes will not be part of any subjective view.

6.4 Changes to the Task Structure

The allowed changes are those described in Chapter 5. Because the subjective and objective models are required to be consistent, the only additional information required is 1) who gets which updates and 2) who sees which nodes, whether local or nonlocal.

Chapter 7: Hyperparameters

Hyperparameters are a set of attributes with ranges of values that give agents an idea about what to expect from a scenario. They focus primarily on meta-CTAEMS related information. In Phase II every ctaems scenario file will contain hyperparameter values. The lowest and highest possible values for each hyperparameter are indicated in square brackets. These are a first pass at reasonable values for the ranges. They should not be considered to be unchangeable at this point. Hyperparameters will always be reported in the form of a range. It is probably most helpful to think of the range as being the low and high values in a uniform distribution.

Hyperparameters related to new task structure arrival:

Number of times new task structures will arrive – number of times during a scenario that new task structures will arrive [0 – 100]

Urgency of new node arrival – difference between new task structure arrival times and new task structure release times [1 – 1000]

Percentage of nodes arriving dynamically – overall percentage of the nodes for the entire objective scenario that are expected to arrive dynamically [0.0 – 0.75]

Hyperparameters related to changes to existing nodes:

Number of times a release/deadline change will occur – the number of times during a scenario that a release time or deadline will be changed [0 – 200]

Percentage of nodes that will be affected by a release/deadline change – the percentage of nodes that will either have their explicitly defined release time or deadline change or will have their implicit propagated release time or deadline change. To be clear "affected" means the node whose deadline or release time is changed directly and all those descendants whose propagated deadline or release time (deadline*, release*) is changed. There may be descendants whose deadline* or release* does not change, and they are not affected. [0 – 0.75]

Percentage of methods that will be affected by a distribution change – the percentage of methods that will have at least one of their duration distribution or quality distribution change [0 – 0.75]

Percent of release changes that are earlier – percent of the changes to release times that result in an earlier release time [0 – 1.0]

Percent of deadline changes that are earlier – percent of the changes to deadline that result in an earlier deadline [0 – 1.0]

Percent of duration distribution changes that are longer – percent of the changes to duration distributions that result in a longer expected duration [0 – 1.0]

Percent of quality distribution changes that are higher – percent of the changes to quality distributions that result in a higher expected quality [0 – 1.0]

Chapter 8: Syntax

This chapter outlines the syntax of the textual representation of C_TAEMS. The meta-syntax is similar to standard BNF, but not identical:

- Square brackets ("[" and "]"): Enclosed element(s) may appear zero or one time.
- Curly braces ("{" and "}"): Group elements together.
- Parentheses: Represent literal parenthes. Note: this is different from standard BNF.
- Plus ("+"): Preceeding element may appear one or more times.
- Astrisk ("*"): Preceeding element may appear zero or more times.

Note that, although it is not directly expressed by the BNF notation below, the reader should presume that there is no ordering constraint among the individual fields that appear within an individual agent, node, or nonlocal effect in the structure.

All comments in cTAEMS files start with a **;** (a semi-colon) and continue to the end of the line.

8.1 Skeleton

```
<taems>         ::= <version> <spec_eoh> [ <hyperparams> ]
                    { <agent> | <node> | <nle> | <schedule> | <changes> |
                    <additions> | <failures> }*
                    [ <attributes> ]
<spec_eoh>      ::= (spec_eoh <pos-integer>)
<node>          ::= <taskgroup> | <task> | <method>
<nle>           ::= <enables> | <disables> | <facilitates> | <hinders>
```

8.2 Common Components

```
<identifier>    ::= <character> <alphanumeric>*
<literal>       ::= <string> | <float> | <integer>
<string>        ::= '"' { <extended-alphanumeric> | ' ' }* '"'

<distribution> ::= { <value> <prob> }+
<pos-int-distribution> ::= { <pos-int-value> <prob> }+
<value>         ::= <float>
<pos-int-value>::= <pos-integer>
<prob>          ::= <float>

<extended-alphanumeric> ::= <alphanumeric>  | '~'|'`'|'!'|'@'| '#'|'$'|
'%'|'^'|'&'|'*'|'-' |'+'|'=' | '[' | '{' | ']' | '}'| '|'| '\' |':' |';'|'"' |
'<' |',' |'>' |'.' |'/' |'?'

<alphanumeric> ::= <character> | <digit> | '-'
<character>    ::= 'a'..'z' | 'A'..'Z' | '_'

<float>         ::= <integer> '.' <digit>+
```

```
<integer>       ::= [ '-' ] <pos-integer> | <digit>
<pos-integer>   ::= '1'..'9' <digit>*
<digit>         ::= '0'..'9'

<non-neg-integer>
                ::= <digit> | <pos-integer>

<percent>       ::= { [ '1' .. '9' ] <digit>  } | 100


<visible-to>    ::= ( visible_to <agentname>+ )

<version>       ::= ( spec_version <literal> )

<attributes>    ::= ( spec_attributes
                      <attribute>*
                    )
<attribute>     ::= ( <identifier> <literal> )
```

8.3 Agents

```
<agent>         ::= ( spec_agent
                      ( label <agentname> )
                      [ <visible-to> ]
                      [ <attributes> ]
                    )
<agentname>     ::= <identifier>
```

8.4 Tasks and Task Groups

```
<task-contents>::= ( qaf <qaf> )
                   ( subtasks <subtaskname>* )
                   [ <visible-to> ]
                   [ ( earliest_start_time <integer> ) ]
                   [ ( deadline <integer> ) ]
                   [ <attributes> ]
<taskgroupname>::= <identifier>
<taskname>      ::= <identifier>
<subtaskname>   ::= <taskname> | <methodname>

<taskgroup>     ::= ( spec_task_group
                      ( label <taskgroupname> )
                      <task-contents>
                    )

<task>          ::= ( spec_task
                      ( label <taskname> )
                      <task-contents>
                    )

<qaf>           ::= q_sum | q_sync_sum |
                    q_max | q_sum_and
                    q_min | q_exactly_one
```

8.5 Methods

```
<method>        ::= ( spec_method
                      ( label <methodname> )
                      ( agent <agentname> )
                      [ <visible-to> ]
                      [ ( earliest_start_time <integer> ) ]
                      [ ( deadline <integer> ) ]
                      <outcomes>
                      [ <attributes> ]
                    )
<methodname>    ::= <identifier>

<outcomes>      ::= ( outcomes
                      <outcome>+
                    )
<outcome>       ::= ( <outcomename>
                      ( density <float> )
                      ( quality_distribution <distribution> )
                      ( duration_distribution <pos-int-distribution> )
                    )
<outcomename>   ::= <identifier>
```

8.6 Non-Local Effects

Note that disables and hinders NLEs will not be used in Year 1.

```
<nle-contents> ::= ( label <nlename> )
                   ( from <nlesource> [ <for-outcomes> ])
                   ( to <nletarget> )
                   [ <visible-to> ]
                   [ ( delay <integer> ) ]
                   [ <attributes> ]
<nlename>      ::= <identifier>
<nlesource>    ::= <taskname> | <methodname>
<nletarget>    ::= <taskname> | <methodname>
<for-outcomes> ::= ( for_outcomes
                      <outcomename>+ )
<outcomename>  ::= <identifier>

<enables>      ::= ( spec_enables
                      <nle-contents>
                    )
<disables>     ::= ( spec_disables
                      <nle-contents>
                    )
<soft-nle-contents> ::=
                      <nle-contents>
                      ( quality_power <distribution> )
                      ( duration_power <distribution> )
<facilitates>  ::= ( spec_facilitates
                      <soft-nle-contents>
                    )
<hinders>      ::= ( spec_hinders
                      <soft-nle-contents>
                    )
```

8.7 Schedules

```
<schedule>      ::= ( spec_schedule
                      ( schedule_elements
                         <scheduleelem>*
                      )
                      [ <attributes> ]
                    )

<scheduleelem> ::= ( <methodname>
                      ( start_time <integer> )
                      [ <attributes> ]
                    )
```

Note: multiple spec_schedule structures should not list the same method. This should result in a syntax error.

8.8 Commitments

Commitments will not apply during the first or second years (2005 & 2006).

The contents of this object are still TBD and subject to change.

```
<commitment>    ::= ( spec_commitment
                      ( label <commitmentname> )
                      ( type <commitmenttype> )
                      ( from_agent <agentname> )
                      ( to_agent <agentname> )
                      ( task <methodname> )
                      [ ( start_time <integer> ) ]
                      [ ( deadline <integer> ) ]
                      [ <attributes> ]
                    )
<commitmentname> ::= <identifier>
<commitmenttype> ::= do | dont
```

8.9 Meta-Level Changes

In this section we provide the syntax the system will use to communicate task structure changes to the simulator and the agents. These correspond to concepts covered in Section 5.4. The contents of the change structure apply to the agent's local subjective view. In particular, the indicated label can be used to find the appropriate node, and the remaining fields used to replace the corresponding fields in that node. For example, upon receipt of a `<methodchange>` with an `<outcomes>` field, those outcomes should replace the set of outcomes originally given by that method.

The time field in the following syntax specifies to the simulator when the changes should take effect.

```
<changes>       ::= ( spec_changes
                      ( time <integer> )
                      <change>*
                       [ <attributes> ]
                    )
```

```
<nlechange>    ::= ( spec_nle
                     ( label <nlename> )
                     [ <visible-to> ]
                     [ <attributes> ]
                   )

<change>       ::= <taskchange> | <methodchange> | <nlechange>




<taskchange>   ::= ( spec_task
                     ( label <taskname> )
                     [ <visible-to> ]
                     [ ( earliest_start_time <integer> ) ]
                     [ ( deadline <integer> ) ]
                     [ <attributes> ]
                   )

<methodchange> ::= ( spec_method
                     ( label <methodname> )
                     [ <visible-to> ]
                     [ ( earliest_start_time <integer> ) ]
                     [ ( deadline <integer> ) ]
                     [ <outcomes> ]
                     [ <attributes> ]

                   )
```

Note: the *visible-to* set for taskchanges and methodchanges will replace the existing one, rather than add to the existing set.

New structures will be specified to the simulator using the following syntax (see 5.4.1 for more on new tasks during simulation):

```
<parentname> ::= <taskname> | <taskgroupname>

<parent> ::= parent <parentname>

<splice> ::=  ( splice
                 ( <parent> | root )
                 ( subtasks <subtaskname>+ )
                 [ <attributes> ]
              )

<additions> ::= ( spec_newtaems
                    ( time <pos-integer> )
                    { <splice> | <agent> | <nle> | <failures> |
                      <schedule> |  <node> | <newnodes>
                    }*
                    [ <attributes> ]
                  )

<newnodes> ::= ( newnodes <identifier>+ )
```

Note that after Phase I, methods as well as tasks may be spliced in.

This syntax allows nodes new to the objective view to be distinguished from nodes that are not new to the objective view (but only new to the agent's subjective view). To be clearer, there are two categories of nodes added to the agent's subjective view at time t during the simulation. These are nodes that were

1) Added to the objective view at time t
2) Already present in the objective view at time t (either present at the start of the simulation or added before time t), but not in the agent's subjective view until time t.

Spec_newtaems structures contain both types of nodes; the categories are disambiguated with the <newnodes> structure to represent nodes that belong to category 1.

Only the new structures will be included in this specification, both to the simulator and to agents; existing structures will not be included.

A scenario may also specify that certain methods will fail when executed, regardless of their specified outcome(s) in the TAEMS structure. Such unpredictable or unmodeled failures are specified to the simulator as follows:

```
<failures>     ::= ( spec_failures
                     <methodname>*
                   )
```

Note that agents will not see this syntax since they learn of this in the same way as other method failures.

8.10 Hyperparameters

```
<hyperparams> ::=
   (spec_hyperparam
      (new_node_arrivals <non-neg-integer>  <non-neg-integer> )
      (new_node_urgency <non-neg-integer>  <non-neg-integer> )
      (nodes_percent_dynamic <percent> <percent> )
      (release_deadline_changes <non-neg-integer>  <non-neg-integer> )
      (nodes_percent_release_deadline_changes <percent> <percent> )
      (methods_percent_distribution_change <percent> <percent> )
      (release_changes_percent_earlier <percent> <percent> )
      (deadline_changes_percent_earlier <percent> <percent> )
      (duration_changes_percent_longer <percent> <percent> )
      (quality_changes_percent_higher <percent> <percent> )
   )
```

Notes

- hyperparams do not have to be specified for it to be legal ctaems.
- Unspecified hyperparams do not rule out the possibility of meta-taems changes (they mean you have no information).

- Any hyperparam specification must specify ALL parameters (but the range can be as loose as you want...)
- spec_hyperparam is a top-level form, not a component of something else like the taskgroup.

Chapter 9: GMASS APIs

This chapter describes the events that GMASS must support, and the API.

9.1 Events

The GMASS simulator and an individual Coordinator will communicate through the exchange of the following events.

9.1.1 GMASS to agent

GMASS will send the following types of messages to Coordinator agents. See the Javadocs for the package `com.globalinfotek.coordinator.message` for details.

Pulse — sent to all agents at start of each tick. Includes tick number and may include any of the other message types as well (except Disconnect).

MethodStart — report whether method start request succeeded, and if so, on which tick.

MethodAbort — report whether method start request succeeded, and if so, on which tick.

MethodComplete — report q/c/d and name of outcome achieved. Subsumes failure, because we can report quality 0.

CTaemsEvent — See Chapter 5. (Note: For details on the events, see the package com.globalinfotek.coordinator.ctaems.event.)

9.1.2 Agent to GMASS

Agents make requests to GMASS using the `com.||globalinfotek.coordinator.Simulation` interface. Status information is reported back to agents via the messages listed in the previous section, except as noted below for the "guaranteed" request types.

StartMethodGuaranteed — start the specified method in the current tick, but only if the API can guarantee that the method will start this tick. Returns true or false to indicate whether the guarantee can be honored.

StartMethod — start the specified method as soon as possible. Optionally, the agent may request that the method only start if the request is received in time for it to start this tick.

AbortMethodGuaranteed — stop the specified method in the next tick, but only if the API can guarantee that the method will abort next tick. Returns true or false to indicate whether the guarantee can be honored.

AbortMethod — stop the specified method as soon as possible. Optionally, the agent may request that the method only abort if the request is received in time for it to abort next tick.

Bye — tell GMASS that the agent is exiting normally.

AbortSimulation — tells GMASS that the agent as encountered an exceptional condition and that the entire simulation must halt. Should only be used as a last resort.

For additional details, see the Javadocs.

Appendix A: Resources, Extensions, Stuff We Left Out, Revision History

10.1 Possible Extensions

This is not remotely all the possible extensions, this is just ones that have been mentioned w.r.t. the Coordinators program...

- Enable sequencing via NLEs with thresholds
- Continuous distributions (over which of q/c/d?)
- Iterative and conditional constructs
- Resources
- Parameterized TAEMS (relation to pTAEMS?)
- Allowing additional fields to be modified and individual nodes to be added (or potentially removed) via the change syntax.
- Additional synchronization QAFs

10.2 Unresolved Issues

Parameterized TAEMS (ptaems)

Representing contingent schedules

Partial orders? (just say no...)

10.3 Deprecated QAFs

These we do not intend to support.

SeqSum

SeqMax

SeqMin

SeqLast

All

The white paper says that All *requires all substasks to be attempted, but if they fail (quality 0), that's ok. Quality accumulates as with* Sum.

Use Min.

Last

Sigmoid

No idea what it is…

ExactlyOne

Effective definition of an XOR depends on supporting task quality as a nonmonotonic function of time (which we could do, but won't do yet), and possibly also (depending on the exact definition you want) on the downward enforcement of task constraints in a way we very much do not want to do.

Resources

C_TAEMS does not include the TAEMS resource models, at least for now.

10.4 Revision History

- **Version 1.00, 4/19/05:** initial release.

- **Version 1.01, 5/17/05:** modified the semantics for the q_min QAF as per Mark Boddy's email to the TAEMS Standards list on May 3, 2005. q_min now accumulates quality incrementally (instead of only when the node is *complete*).

- **Version 1.02, 6/24/05:**
  - Describes how subjective task models are derived from the objective model (Section 6.3.1)
  - Removes requirements for a subjective task group
  - Removes references to subjective task groups
  - Modifies schedule syntax

- **Version 1.03, 8/12/05**
  - Revised schedule syntax
  - Revised meta-cTAEMS syntax

- **Version 1.04, 8/25/05**
  - Added *visible-to* syntax

- **Version 1.05, 9/6/05**
  - Modified spec_newtaems syntax

- **Version 1.06, 10/3/05**
  - Clarified syntax

- **Version 1.07, 5/10/06**
  - Fixed duraton distribution typo, page 13, referenced in bug #435
  - Added comment syntax

  - Removed CONTRIB as per Mark Boddy's email, May 2005
  - Fixed typo in definition of MIN QAF (bug #399)
- **Version 2.00, 7/19/06**
  - Changed meta-syntax to use curly braces for grouping only, and to use plus and asterisk with their usual meanings (i.e., one or more and zero or more, respectively).
  - Added Phase 2 changes
    - QAFs
    - Meta-ctaems events (release, deadline, distribution, task additions)
    - NLEs
    - Outcome-based NLEs
    - Coefficient sampling
    - NLE syntax modification
- **Version 2.01, 8/1/06**
  - Incorporated Chris Long's suggested changes (see 7/21/06 email)
  - Incorporated Tim Lebo's request for additional characters
- **Version 2.02, 9/18/06**
  - Changed visible-to in NLEs
  - Added hyper-parameters
- **Version 2.03, 1/4/07**
  - Revised definition of duration power for facilitates and hinders.
  - Added definitions of MaxQ for new QAFs.
  - Modified terminology to avoid the use of the phrase "maximum quality".
  - Added optional attribute blocks to syntax
  - Remove costs (for Phases 1 and 2)
  - Added spec_eoh construct
  - Modifed definitions of release* and deadline* in Section 3.3.2
- **Version 2.04, 2/28/07**
  - Note that discrete probability distribution is a function in Section 3.2.1.
  - Modify definition of ACTIVE (pg. 10, T7).
  - Refine definition of pq (Section 3.4.3).