SOCS

A COMPUTATIONAL LOGIC MODEL FOR THE DESCRIPTION, ANALYSIS AND VERIFICATION OF GLOBAL AND OPEN SOCIETIES OF HETEROGENEOUS COMPUTEES

IST-2001-32530

Further Examples for the functioning of computees

Project number:	IST-2001-32530
Project acronym:	SOCS
Document type:	DN (discussion note)
Document distribution:	I (internal to SOCS and PO)
CEC Document number:	IST32530/UCY//DN/I/a2
File name:	4-a2[MB].ps
Editor:	Antonis Kakas
Contributing partners:	UCY,UNIBO
Contributing workpackages:	WP4
Estimated person months:	1
Date of completion:	29 January 2004
Date of delivery to the EC:	31 January 2004
Number of pages:	33

ABSTRACT

This document is a companion to the document [Aea03b] identifying further testing examples for the KGP model of computees proposed in deliverable D4 [KMS+03]. These examples are developed under a common scenario, which was posed in [KM03] together with challenging problems to be addressed. They are integrated examples in the sense that they bring together several of the components of an individual computee and they also involve non-trivially interaction between computees.

The examples are used to test further several Global Computing requirements of the model according to the (success) criteria for self-assessment and evaluation put forward in D3. They are also used to illustrate the integrated functionality of computees where most of their capabilities and reasoning with their cycle theory need to be involved together and how this can lead to an enhanced operation of the computees.

Copyright © 2004 by the SOCS Consortium.

The SOCS Consortium consists of the following partners: Imperial College of Science, Technology and Medicine, University of Pisa, City University, University of Cyprus, University of Bologna, University of Ferrara.

Further Examples for the functioning of computees

Neophytos Demetriou*,
Antonis Kakas*,
Paolo Torroni+

Departments of Computer Science,
* University of Cyprus, Cyprus
+ University of Bologna, Italy

ABSTRACT
This document is a companion to the document [Aea03b] identifying further testing examples for the *KGP* model of computees proposed in deliverable D4 [KMS+03]. These examples are developed under a common scenario, which was posed in [KM03] together with challenging problems to be addressed. They are integrated examples in the sense that they bring together several of the components of an individual computee and they also involve non-trivially interaction between computees.
The examples are used to test further several Global Computing requirements of the model according to the (success) criteria for self-assessment and evaluation put forward in D3. They are also used to illustrate the integrated functionality of computees where most of their capabilities and reasoning with their cycle theory need to be involved together and how this can lead to an enhanced operation of the computees.

Contents

1 Introduction

The companion document [Aea03b] of this document uses as its basis the Leaving San Vincenzo scenario [Sta02] to develop testing examples for the *KGP* model. Another scenario for the demonstration examples is the *Music for Beer* scenario. This is taken from the recent work of [KM03] who proposed this scenario in order to demonstrate their approach to cooperative problem solving via Linear Logic. The scenario was presented in [Tor03] as a challenging problem to be addressed in the SOCS project.

We will use the Music for Beer scenario extending its example in several ways in order to demonstrate various evaluation criteria for the SOCS model as presented in [LMM$^+$03]. This scenario complements the Leaving San Vincenzo scenario in demonstrating many of the same features thus reaffirming the desired properties of the model as general properties. It also demonstrates additional features particularly pertaining to the interaction between the computees in their attempt to solve their problems.

The structure of this document is as follows. First we describe the scenario that we will use, to formulate all examples in this document. We then present a standard integrated example in this scenario as presented in [KM03] to test the behaviour of individual and interacting computees (WP1). Variations of this standard example are developed to test further the model. Then we present briefly examples that link the behaviour of the interacting computees to the society that they belong and the expectations that this may raise.

2 The *Music for Beer* scenario

The *Music for Beer* scenario. as presented in [KM03] can be summarised as follows. Two students (computess for us in this document) John and Peter have the following goals. John has the goal to listen to music and Peter the goals of retuning his books to the library and drink beer.

John has 10 Euro[1], a CD, and a Broken CD player. He is able to play Music - by doing so he would consume the resources CD and CDPlayer - and additionally he would be able to return books to the library at zero cost (its on its way), consuming in this way the resource Books. Peter has 15 Euro and a bunch of Books to be returned. To get the Beer he needs 25 Euro. He is able to return the Books to the library, by paying 10 Euro for the taxi. He is also able to repair broken CD Players (whereas John does not have this ability himself).

Both computees are aware of each other but they do not necessarily know each others current resources or capabilities. Each computee simply knows which of its own goals or subgoals can be requested without knowing if a request can be satisfied by the other computee or not.Hence as in the *Leaving San Vincenzo* scenario we assume that for each computee a subset of the fluents in its language are like resource that therefore can be requested and exchanged between computees. For example, for John the fluent *working_CDPlayer* is such a fluent like the resource of *money*. Then the computees can request directly from another computee to "bring about" the truth or falsity of this fluent.

The main features that will be demonstrated in this scenario are:

- Adaptation of operation and decisions of computee in the face of new information acquired from other computees or the society at large.

[1]The original example in [KM03] uses dollars instead of Euro.

- Negotiation between the two computees in their attempt to satisfy their respective goals.
- Change of plan of a computee to satisfy goals as a result of the negotiation.
- Comparative executions of the same scenario under identical conditions except the policy of each computee to be cooperative or non-cooperative.

The main focus of this scenario and its examples is to demonstrate how the KGP model of computee allows for autonomous agents to interact together in different ways (cooperative or non co-operative) thus distributing their tasks and dynamically adapting there decisions in the face of new information provided by the other computees.

2.1 Relevance of *Music for Beer* to GC

The scenario and its examples are designed to test several of the evaluation criteria for the KGP model of computees in [LMM+03] which have been set out to address requirements of Global Computing. Specifically, this scenario and its examples address the criteria of:

Adaptability: Adjustment where the computee decides to respond differently to requests depending on the needs it can currently satisfy and/or on the needs that it currently has for itself.

Adaptability: Adjustment where the computee changes its requests according to new information from its environment (namely the other computee in this case).

Adaptability: Suspension/Introduction where the computee through its behaviour pattern of operation (defined via its cycle theory) introduces goals of response and prefers to suspend operation on other goals in favour of such response goals.

Partial Information: Conditional decisions where computees develop plans conditional on the assumption that a request for a need within the plan will be accepted and thus provided.

Distribution: Decentralization where a computee decides to achieve a goal with a plan which also involves other computees. The computee has thus distributed its task.

Heterogeneity: Overall Behaviour where the cycle theory and goal decision preference policies of a computee regulate its cooperative or not behaviour and the relative importance that a computee gives to requests from other computees.

Heterogeneity: Personality where the personality policy of the computee within its goal decision policy affects its decisions as to whether to accept or not a request to provide a need.

The examples to be developed will be integrated examples in the sense that they bring together several of the components of an individual computee and they also involve non-trivially interaction between computees. As such they will illustrate the integrated functionality of computees where most of their capabilities and reasoning with their cycle theory need to be involved together and how this can lead to an enhanced operation of the computees.

This scenario also tests to a large extend the criterion of *Modularity* of the KGP model where modular changes on the cycle theory or the goal decision policy of the computee are easily accomplished producing different required behaviour.

3 Integrated Examples for WP1: individual interacting computees

In this section, we will show the behaviour of the two computees $John$ and $Peter$ on a standard integrated example taken and extended from [KM03]. This is an integrated example where several features of the computee model will be illustrated.

We will first define the two computees by giving all their knowledge bases and their cycle theories and then describe their evolving behaviour in this standard example. In describing the knowledge bases of the computees we will give their main components here with more details in the appendices of this document.

3.1 Cycle theory of Peter and John

The cycle theory of the two computees will be the same. This is the *normal* cycle theory, as specified in deliverable [Aea03a] and in the companion example document for the Leaving San Vincenzo scenario [Aea03b], extended to cover a wider spectrum of behaviour. This extension is given by the following additional rules in the theory T_{cycle}^{normal}.

In the normal cycle theory after a passive observation (POI) the preferred next transition is goal introduction (GI), as ensured by the following priority rule:

$$h\_p(r_{POI|GI}(S'), r_{POI|*}(S, S')).$$

In particular, this means that computees perform a goal introduction immediately after they receive new messages from other computees. The basic rule $r_{POI|GI}(S, \_)$ that introduces the GI transition as the next transition is given by:

$$r_{POI|GI}(S) : GI(S') \leftarrow POI(S, Obs, S'), non\_terminal(Obs).$$

where $non\_terminal(Obs)$ is defined using the auxiliary predicate $terminal/1$ which specifies which communication messages received need no reply (see appendix for details).

More generally, we can impose that if a computee has unanswered messages then this will give higher priority to goal introduction (GI) in order to decide how to answer these messages. We achieve this by the following preference rule:

$$h\_p(r_{*|GI}(S, \_), r_{*|*}(S, \_)) \leftarrow unanswered(msg(from(\_), Message), T_{now}).$$

where $unanswered(msg(from(\_), Message), T_{now})$ is an auxiliary predicated (see appendix) defined via the temporal reasoning capability of the computee and T_{now} is the current time at which the cycle theory is called to decide the next transition.

In addition, the normal pattern of behaviour that we will use in our examples gives preference to planning communication goals and executing communication actions. This is accomplished via the following preference rules:

$$h\_p(r_{GI|PI}(S, Gs), r_{GI|*}(S, \_)) \leftarrow communication\_goal(Gs).$$

where $communication\_goal/1$ is a heuristic function that selects messaging goals, i.e. goals of the form $msg(to(\_), \_)$, and:

$$h\_p(r_{*|AE}(S, As), r_{*|*}(S, \_)) \leftarrow communication\_action(As).$$

where $communication\_action/1$ is also a heuristic function that selects communication actions, i.e. actions of the form $tell(\_,\_)$.

The full T^{normal}_{cycle} theory is found in the first appendix.

3.2 State: Knowledge Bases of the Computees

In this subsection, we will present the various general knowledge bases, KB_{GD}, KB_{plan} and KB_{TR} of the two computees. These will include domain (example) independent parts and parts of specific knowledge for the examples.

For the standard example both computees will have the same theory of KB_{GD} encoding the same policy of deciding how to respond to requests and negotiate. Informally, this policy says:

> Accept a request for a need when this does not invalidate your current tasks. Prefer to get in return for accepting request a need, i.e set terms for accepting, that you (currently) have. Accept terms when these do not invalidate your current tasks otherwise if they do ask for alternative terms. When no alternative terms are offered then you can either accept the last term offered or refuse depending on other criteria relevant to the your needs and terms. Also you may have reasons to prefer some terms over other terms.

Needs and Terms are both a list of fluent literals referring to the special set of requestable fluents. This set of fluents will typically be different for different computees. Terms can be the empty list. Requests, can be any non-empty set of needs. A special case of a need fluent is $has(Resource)$ where $Resource$ names a resource.

Below the fluent $msg(to(ComputeeName), Message)$ will denote the property that the computee, with id $ComputeeName$, has the message, $Message$, e.g. a request or a response. Similarly, for $msg(from(ComputeeName), Message)$.

A KB_{GD} theory consists of three parts. We give the main components of each part here with more details in the appendix.

- KB^{low}_{GD}:

  A computee may accept or refuse an (initial) request.

  - A computee could accept a request only if it is able to satisfy the given request. A computee accepts the request under some $Terms$ that are part of its own (current) needs which it asks in return:

$$\begin{aligned}
&gd^{request,init}_{accept}(Peer, Reply):\\
&\quad msg(to(Peer), Reply)[RT] \leftarrow\\
&\qquad unanswered(msg(from(Peer), request(Request)), T_{now}),\\
&\qquad satisfiable(Request),\\
&\qquad current\_terms(Terms),\\
&\qquad Reply = (accept, request(Request), Terms),\\
&\qquad response\_time(T_{now}, RT).
\end{aligned}$$

Note that the list of $Terms$ could be empty in which case the acceptance of the $Request$ is unconditional. Here the various predicates that appear in the body of this rule are auxiliary predicates that belong to KB_{GD}^{aux} and may refer to the current state ($Goals$ and $Plan$) of the computee. These will be presented in the appendix. Similarly, we have a rule, with name $gd_{reject}^{request}$, that generates a reply of rejection (see appendix) to an (initial) request.

When a computee is offered $Terms$ to its request these become a request back to it. It can either accept these (when it can satisfy them) or it can ask for alternative terms, e.g. when it cannot satisfy them or when they are incompatible with its current needs, etc.

– The case of asking for alternative terms is given by the following rule:

$$\begin{array}{l} gd_{reject}^{response}(Peer, Reply): \\ msg(to(Peer), Reply)[RT] \leftarrow \\ \quad unanswered(msg(from(Peer), (accept, request(Request), Terms)), T_{now}), \\ \quad unsatisfiable(Terms), \\ \quad Reply = (reject, response(Request, Terms), alt\_terms), \\ \quad response\_time(T_{now}, RT). \end{array}$$

To complete the policy we need to formulate the response policy for the case when alternative terms are asked.

– A computee can offer alternative terms when requested to do so:

$$\begin{array}{l} gd_{accept}^{request,alt}(Peer, Reply): \\ msg(to(Peer), Reply)[RT] \leftarrow \\ \quad unanswered(msg(from(Peer), (reject, response(Request, Terms), alt\_terms)), T_{now}), \\ \quad alternative\_terms(Alt\_Terms), \\ \quad Reply = (accept, request(Request), Alt\_Terms), \\ \quad response\_time(T_{now}, RT). \end{array}$$

Note that $alternative\_terms/2$ (an auxiliary predicate) may return an empty list and thus this rule also provides t he option for the computee to eventually accept the initial request with no terms in return (after all its terms offered were rejected).

– If the computee has no alternative terms left to offer then it could reject the request.

$$\begin{array}{l} gd_{reject}^{request,alt}(Peer, Reply): \\ msg(to(Peer), Reply)[RT] \leftarrow \\ \quad unanswered(msg(from(Peer), (reject, response(Request, Terms), alt\_terms)), T_{now}), \\ \quad alternative\_terms(Alt\_Terms), Alt\_Terms = [], \\ \quad Reply = (reject, request(Request), []), \\ \quad response\_time(T_{now}, RT). \end{array}$$

Note that these two last rules are two contradictory options of reply when there are no alternative terms left.

- KB^{high}_{GD}: The following priorities are applied on the generation rules above to complete the preference policy. In general, a computee prefers to accept requests, ie. the rule $gd^{request,init}_{accept}$ has priority over the rule $gd^{request,init}_{reject}$. Computees also prefer to accept with some terms over accepting unconditionally, i.e. with empty terms:

$$\begin{aligned}&gd\_pref^{request}_{accept,cond}(Peer, Reply):\\ &\quad h\_p(gd^{request,*}_{accept}(Peer, (accept, request(R), Terms)),\\ &\qquad gd^{request,*}_{accept}(Peer, (accept, request(R), []))) \leftarrow Terms \neq [].\end{aligned}$$

  Accepting with empty $Terms$ has no personal gain for a computee and thus a computee may prefer to reject the request over accepting it with no Terms:

$$\begin{aligned}&gd\_pref^{request}_{reject}(Peer, Reply):\\ &\quad h\_p(gd^{request,*}_{reject}(Peer, Reply),\\ &\qquad gd^{request,*}_{accept}(Peer, (accept, request(R), []))).\end{aligned}$$

  Then higher order preferences can be used to distinguish between what we might consider a *cooperative* and a *non-cooperative* computee. A simple example for a cooperative computee gives preference to accepting a request even with no terms:

$$\begin{aligned}&gd\_pref^{request}_{coop,accept}(Peer, Reply):\\ &\quad h\_p(gd\_pref^{request}_{accept}(Peer, Reply), gd\_pref^{request}_{reject}(Peer, Reply')).\end{aligned}$$

  A non cooperative computee gives preference to rejecting a request over accepting it with no terms. This is captured by::

$$\begin{aligned}&gd\_pref^{request}_{non-coop,reject}(Peer, Reply):\\ &\quad h\_p(gd\_pref^{request}_{reject}(Peer, Reply), gd\_pref^{*}_{accept}(Peer, (accept, request(\_), []))).\end{aligned}$$

- KB_{plan} **- Domain Independent:**
  Both computees contain a domain independent part in their KB_{plan} to help them plan to achieve their message communication goals, i.e. plan how to send a message. We will assume that this is very simple and that it is simply captured via the rule:

$$initiates(tell(Peer, msg(from(self), Message)), T, msg(to(Peer), Message)).$$

  i.e. to satisfy a goal of $msg(to(Peer), Message)$, we generate a simple plan with the only action of $tell(Peer, msg(from(Self), Message))$.[2]

[2]For ease of presentation, we have used a tell action with only two arguments. An action $tell(Peer, msg(from(Self), Message))$ stands for $tell(Self, Peer, msg(from(Self), Message), D)$ where D is a unique dialog id.

More importantly, their KB_{plan} contains the following domain independent rules:

$$initiates(get(from(Peer), Need), T, Need).$$
$$precondition(get(from(Peer), Need), ass\_promise(from(Peer), Need))).$$

that allows them, during planning to generate an action *get* in their plan, when a computee wants to satisfy a goal or subgoal, $Need$, that it knows it *could* be provided by some computee. In order to execute such a *get* action the computee needs the precondition of a promise from the $Peer$ computee from whom it will get the $Need$. Therefore, in a plan *get* will need to be preceded by an action that generates the promise. This is accomplished by the rule:

$$initiates(tell(Peer, msg(from(self), request(Need))), T,$$
$$ass\_promise(from(Peer), Need))$$

Hence in this way when a computee plans for a (sub)goal, which it knows can be asked from some other computee, it can generate a plan consisting of a *tell* action to request this and assuming that this will succeed (i.e. the request will be accepted) a *get* action will achieve the (sub)goal.

It is important though that a computee does not execute a $get(from(Peer), Need)$ action until it has a real promise for the $Need$ after its request for this. One way to ensure that the computee will indeed wait before executing the *get* action is to include a suitable check in the core action selection function, of the cycle theory, so that $get(from(Peer), Need)$ is selected only if $holds\_at(promise(from(Peer), Need), T_{now})$ is true (see appendix).

- KB_{plan} **- Domain Dependent:**

  Following the example as presented in [KM03] the two computees have the following simple specific planning knowledge that captures what [KM03] calls the capabilities of each agent(computee).

  - KB_{plan} for computee John: John can return books and play music. This means that it knows what these actions do. We express this simply by

    $$initiates(return(Books), T, books\_returned(Books))$$
    $$initiates(play(CD), T, music) \leftarrow holds\_at(working\_CDPlayer, T)$$
    $$precondition(return(Books), has(Books))$$
    $$precondition(play(CD), has(CD))$$
    $$terminates(play(CD), T, has(CD))$$

  - KB_{plan} for computee Peter: Peter can repair cd players, return books and buy beer. Its planning knowledge base thus contains:

    $$initiates(return(Books), T, books\_returned(Books)) \leftarrow holds\_at(has(money(books, 10), T)$$
    $$initiates(buy(beer), T, has(beer)) \leftarrow holds\_at(has(money(beer, 25), T)$$

$$initiates(repair\_CDPlayer, T, working\_CDPlayer)$$

$$precondition(return(Books), has(Books))$$

$$precondition(repair\_CDPlayer, neg(working\_CDPlayer))$$

$$terminates(buy(beer), T, has(money(beer, 25)))$$

$$terminates(return(books), T, has(money(books, 10)))$$

Note that we could use here a general theory of how needs which are having resources, e.g. $has(money(\_)$ or $has(CD)$ are consumed and when these are available or not. This is not important for the examples and it is beyond the scope of this example document.

- **KB_{TR} and KB_0;**

  As mentioned above, promises for requests are evaluated using the temporal reasoning capability of the computee. Both computees contain a domain independent part of KB_{TR} to reason about promises.

  A computee keeps a record, via Passive Observation and Action Execution transitions, of requests and responses as executed actions (i.e. events) in the KB_0 part of its knowledge base. The KB_{TR} of the computee is then extended with the following axioms so that the computee can reason from these events to promises held by itself to other computees and vice-versa.

  For an unconditional acceptance (i.e. without terms), we have rules of the form:

$$\begin{aligned}&initiates(tell(Peer, msg(from(Self), (accept, request(Request), [])), T,\\ &\quad promise(to(Peer), Request)).\end{aligned}$$

  Similarly, for promises for terms offered, we have rules such as:

$$\begin{aligned}&initiates(tell(Self, msg(from(Peer), (accept, response(Request, Terms), [])), T,\\ &\quad promise(from(Peer), Terms))\end{aligned}$$

  generating a promise from the $Peer$ computee since it has accepted the computee's $Terms$.

  For conditional acceptance the response only initiates a conditional promise pending (forward in time) on the acceptance of the terms asked for. So we have for example:

$$\begin{aligned}&initiates(tell(Peer, msg(from(Self), (accept, request(Request), Terms))), T,\\ &\quad cond\_promise(to(Peer), Request, Terms))\end{aligned}$$

  where $cond\_promise(to(Peer), Request, Terms)$ means that the computee has promised the $Request$ provided that the other computee shall promise back the $Terms$.

  The conditional promise of a request under some terms together with an unconditional promise of the terms link together to give an unconditional promise of the request. This is captured rules such as:

$$\begin{aligned}&initiates(tell(Self, msg(from(Peer), (accept, response(Request, Terms), []))), T,\\ &\quad promise(to(Peer), Request)) \leftarrow\\ &\qquad holds\_at(cond\_promise(to(Peer), Request, Terms), T).\end{aligned}$$

The "book keeping" of recognizing to which requests/responses a computee has already replied to and which ones are left unanswered is done, as mentioned already above using the auxiliary (for the cycle theory and KB_{GD} knowledge base) predicate of $unanswered(msg(from(Sender), Message), T)$. This is evaluated via the temporal reasoning capability in a way analogously to promises. The details of this are in the appendix.

- **Initial state of Computees**

  Finally, initially the state of the two computees contains, according to the standard example of [KM03], the following facts. For John we have:

$$holds\_initially(\neg working\_CDPlayer)$$

$$holds\_initially(has(cd))$$

$$holds\_initially(has(money, 10)).$$

  For Peter we have:

$$holds\_initially(has(books))$$

$$holds\_initially(has(money, 15)).$$

  For both computees, *Goals* and *Plan* starts empty.

3.3 Behaviour of the Computees

In this subsection, we describe the behaviour of the two computees on the standard example of [KM03]. We give a summary of this behaviour as an operational trace for each computee stating the next transition and its effect on the state of the computee. At various points as the example evolves we will show in bold the specific evaluation criterion/characteristic demonstrated at those points.

The example shows how the integrated capabilities of an individual computee work together in order for the computee to exhibit various Global Computing characteristics such as adaptability, conditional decisions and decentralization of task. We note again that this example will also demonstrate how the operation of the computee is enhanced when it uses all its various capabilities and cycle theory together according to the KGP model proposed in this project.

In general, the two computees start together but operate asynchronously. For this standard example we will assume only one point of synchronization, namely that the computees start communicating with each other before they start executing any other actions in their plans. In the following section of variations of the standard example we will consider (briefly) various other examples with different synchronization (or lack of synchronization).

In this standard example, we will also not be concerned how the computees generate their top-level goals (via their GD capability). Again below in the section of variations we will see examples of this. Hence we assume that the state of the computees starts with empty *Plan* and *Goals* given by:

$$\begin{array}{l} Goals(John): \\ \quad g_1^j = \langle music[T], \bot, \{T > 10, T \leq 15\}\rangle \end{array}$$

$$\begin{aligned}&Goals(Peter):\\&\quad g_1^p = \langle beer[T], \bot, \{T < 20\}\rangle\\&\quad g_2^p = \langle books\_returned(books)[T1], \bot, \{T1 < 20\}\rangle\end{aligned}$$

Then the behaviour (operational trace) of the computees evolves as follows.

- Next at time 1, via PI transition(s) the two computees plan for these goals changing their states to:

$$\begin{aligned}&State(John):\\&\ Goals:\\&\quad g_1^j = \langle music[T], \bot, \{T > 10, T \leq 15\}\rangle\\&\quad g_2^j = \langle working\_CDPlayer[T1], g_1^j, \{\}\rangle\\[1em]&\ Plan:\\&\quad a_1^j = \langle play(cd)[T'''], g_1^j, \{has(cd)[T''']\}, \{T''' < T1\}\rangle\\&\quad a_2^j = \langle tell(peter, msg(from(john), request(working\_CDPlayer)))[T'],\\&\qquad\quad g_2^j, \{\}, \{T' < T''\}\rangle\\&\quad a_3^j = \langle get(from(peter), working\_CDPlayer)[T''],\\&\qquad\quad g_2^j, \{ass\_promise(from(peter), working\_CDPlayer)\}, \{T'' < T'''\}\rangle\end{aligned}$$

Note here that John has made some choices in the planning. He has chosen to use his own cd rather than request one. For the goal $working\_CDPlayer[T]$ John knows that it cannot plan for it (it does not have actions that can bring it about) and therefore he necessarily treats this as a need which he plans by adding a request action, namely a_2^j. Note also that we are assuming here that John's top level goal of music is not of the type that he can request it.

$$\begin{aligned}&State(Peter):\\&\ Goals:\\&\quad g_1^p = \langle beer[T1], \bot, \{T1 < 20\}\rangle\\&\quad g_2^p = \langle books\_returned(books)[T2], \bot, \{\}\rangle\\&\quad g_3^p = \langle has(money(beer, 10))[T1'], g_1^p, \{\}\rangle\\&\quad g_4^p = \langle has(money(books, 10))[T2'], g_2^p, \{\}\rangle\\[1em]&\ Plan:\\&\quad a_1^p = \langle tell(john, msg(from(peter), request(has(money(beer, 10)))))[T],\\&\qquad\quad g_3^p, \{\}, \{T < T1'', T < T2''\}\rangle\\&\quad a_2^p = \langle tell(john, msg(from(peter), request(has(money(books, 10)))))[T'],\\&\qquad\quad g_4^p, \{\}, \{T' < T1'', T' < T2''\}\rangle\\&\quad a_3^p = \langle get(from(john), has(money(beer, 10))[T1''], g_3^p,\\&\qquad\quad \{ass\_promise(from(john), has(money(beer, 10)))\}, \{T1'' < T1'''\}\rangle\end{aligned}$$

$$\begin{aligned}a_4^p &= \langle get(from(john), has(money(books,10))[T2''], g_4^p,\\ &\qquad \{ass\_promise(from(john), has(money(books,10)))\}, \{T2'' < T2'''\}\rangle\\ a_5^p &= \langle buy(beer)[T1'''], g_1^p, \{\}, \{T1''' < T1\}\rangle\\ a_6^p &= \langle return(books)[T2'''], g_2^p, \{has(books)\}, \{T2''' < T2\}\rangle\end{aligned}$$

Note that we have assumed here that when Peter plans for these two goals he uses 15 euros that he has for the goal of drinking beer. Therefore, Peter needs to request 10 euros for the goal of returning the books, namely g_2^p, and also request 10 euros for buying beer. We are also assuming that Peter contains rules in its knowledge base that enable him to reason about the cumulative resource of having money. With this, he can work out what amount of money he needs with respect to the amount of money it has.

[**Distribution - Decentralization**]: of goals by requesting tasks from other computees. [**Partial Information - Conditional decisions**]: operating in an unknown environment under the assumption that a computees will accept request made to it.

- At time 2 the cycle theory of John will choose an Action Execution (AE) transition for the communication action a_2^j. Executing this action adds the following in John's KB_0:

$$executed(tell(peter, msg(from(john), request(working\_CDPlayer))), 2).$$

- Now assume that at time 5 Peter receives this message of request. Then via a POI transition it ecords in its KB_0:

$$observed(msg(from(john), request(working\_CDPlayer)), 5).$$

- Peter's cycle theory will choose as the next transition GI to decide how to respond to this. The GD capability called by this transition will generate the new goal to accept this with some terms. We assume that Peter asks as Terms all its current needs, i.e. $[has(money(beer, 10))$ and $has(money(books, 10))$. We also assume that Peter knows how to add these two needs together and ask for $has(money(20))$.

  Therefore, goal introduction (GI) for Peter introduces the communication goal of

$$\begin{aligned}g_4^p = \langle &msg(to(john), (accept, request(working\_CDPlayer),\\ &[has(money(20))]))[9], \bot, \{\}, \{\}\rangle\end{aligned}$$

  to be achieved by time 9 say, arranged by the auxiliary predicate $response\_time/2$ in KB_{GD}.

  Actually, Peter has a choice here between asking as $Terms$ 20 euro or 10 euros and the task of $books\_returned(books)$ since we assume that this type of goal is one that can be possibly asked for from other computees.

- Next Peter through a Planning Introduction (PI) for this goal the following action in its Plan:

$$\begin{aligned}a_7^p = \langle &tell(john, msg(from(peter),\\ &(accept, request(working\_CDPlayer), [has(money(20))])))[T],\\ &g_4^p, \{\}, \{T < 9\}\rangle\end{aligned}$$

- Peter immediately afterwards at time 5 chooses an AE for this communication action adding in its KB_O

$$\begin{aligned}&executed(tell(john, msg(from(peter),\\&\quad (accept, request(working\_CDPlayer[T]), \{has(money(20))\}))), 5)\end{aligned}$$

- John will receive this response through a Passive Observation and record this message in its KB_0.

- Then as above for Peter, John's cycle theory will choose GI to decide on a response and then PI and AE in order to send the response back to Peter. The GI transition will set the new goal:

$$g_3^j = \langle msg(to(peter), (reject, response(working\_CDPlayer, \{has(money(20))\}), alt\_terms)[8], \bot, \{\}, \{\}\rangle$$

since the GD policy will prefer to ask for alternative terms in view of the fact that it cannot satisfy the requested terms. The AE transition at time 7 will update John's KB_0 by:

$$\begin{aligned}&executed(tell(peter, msg(from(john),\\&\quad (reject, response(working\_CDPlayer, \{has(money(20))\}), alt\_terms))), 7)\end{aligned}$$

[**Adaptability - Adjustment:**] of decision to the current circumstances of the computee.

- Now assume that Peter receives this (rejection) message at time 8, via a POI, and therefore its cycle theory will choose next GI, PI and AE in order to send back a reply in the same way as we have seen above. The GI transition will produce the new communication goal that in effect replaces the goal g_4^p:

$$\begin{aligned}g_5^p = \langle &msg(to(john), (accept, request(working\_CDPlayer),\\&\{has(money(10))\}))[T], \bot, \{\}, \{T < 11\}\rangle\end{aligned}$$

where the GD capability chooses the preferred answer of asking for the terms of $has(money(10))$, selected via the auxiliary predicate $alternative\_terms/2$ to satisfy at least one of its goals.

[**Adaptability - Adjustment:**] of decision to the new information received.

- Again, via PI and AE transitions, Peter communicates this to John who receives it via a PO transition and decides (via GI and GD) to accept the terms since now they can be satisfied.

- John communicates its acceptance of the terms via subsequent PI and AE transitions as above and Peter receives this acceptance via a new POI.

- This message is a terminal message and when Peter receives it, its cycle theory will not choose GI now as the next transition as there is no need to reply back.

- At this stage, at time 11, the cycle theory for both computees chooses AE transitions since PI cannot be choosen as there are no unplanned goals in the states of the two computees. Amongst the actions to be chosen for action execution (AE), the normal cycle theory will choose the *get* actions for John and Peter, i.e.:

  $get(from(peter), working\_CDPlayer)$ and $get(from(john), has(money(10)))$,

  respectively (Peter chooses one of the two get actions for 10 euros that it has).

  This is because now, the computees can reason with their Temporal Reasoning capability (inside the cycle theory) to conclude that promises hold true on either side, i.e.: John will be able to conclude

  $$promise(from(peter), request(working\_CDPlayer))$$

  holds and similarly, Peter will be able to conclude

  $$promise(from(john), has(money(10))).$$

  [**Adaptability - Adjustment:**] of decision (of cycle theory now) to new information received and reasoned from.

- Then, at time 13 say, John will have a $working\_CDPlayer$ and its cycle theory would choose action execution (AE) for the remaining action for $play(cd)$ that generates $music$ from time 14 onwards.

- John's operation then continues with GR and PR transitions whose execution makes its *Goals* and *Plan* both empty.

- For Peter at time 13 its cycle theory will choose AE for one of the actions of buy beer or return books. Let us assume that it chooses the first one. It therefore executes this and thus satisfies the goal of beer.

- After this the cycle theory of Peter proposes as the next transition to be that of GR since there are no executable actions left in its plan (the action $return(books)$ is not executable since its preconditions are now false). This GR will be followed by PR and Peter's state becomes such that it only includes the goal:

  $$g_2^p = \langle books\_returned(books)[T1], \bot, \{T1 < 20\}\rangle$$

- The next transition for Peter is a PI transition for this goal that introduces a new plan that consists of the actions

  $$\begin{aligned} a_8^p = \langle & tell(john, msg(from(peter), request(books\_returned(books))))[T1'], \\ & \{ass\_promise(from(john), books\_returned(books))\}, \{\}, \{T1' < T1''\}\rangle \\ a_9^p = \langle & get(from(john), books\_returned(books))[T1''], \\ & g_2^p, \{ass\_promise(from(john), books\_returned(books))\}, \{T1'' < T1\}\rangle. \end{aligned}$$

  [**Adaptability - Adjustment:**] different plan (request) is chosen in the face of the new state of affairs (of no money).

- Now, via an action execution (AE) transition Peter will execute a_8^p.
- John will receive (via POI) the message of the request which will prompt a GI transition to decide his reply. John will accept this unconditionally as he has no current needs now and so has no terms to ask for in return. John and Peter both have a KB_{GD} policy for cooperative negotiation. It sends this reply, $msg(from(John)(accept, request(books\_returned(books))), [])$, via an AE as we have seen above.
- Now Peter can reason with its Temporal reasoning that he has a promise for the request and its cycle theory would select an AE transition to execute the get action $get(from(john), books\_returned(books))$.
- Finally, as there are no actions left Peter's cycle theory chooses a GR and then PR transitions that leaves him also with empty *Goals* and *Plan*.

4 Variations on the Standard Integrated Example

In this section, we discuss briefly variations of the standard example (from [KM03]) presented above in order to show the scope of the solution to this type of problems offered by the KGP model of computees. Also these variations show strongly the modularity of the KGP model as local changes in the definition of the computees can accommodate different required behaviour.

There are at least three factors that give interesting variations to the RAP problem. These are: (1) changing the personalities (e.g. cooperative or non-cooperative) of the two computees, (2) the relative timing at which each computees top goals are introduced in its state (i.e acquire top priority via the computees goal decision policy), and (3) the relative timing between the execution of the different operations of the two computees, e.g. which one requests first and what it requests.

The examples below mainly demonstrate the **Heterogeneity** of the computees in their overall behaviour stemming from their heterogeneity in Personality.

4.1 Varying the Goal Decision circumstances and KB_{GD}

The standard example has made the assumption that both computees were cooperative, i.e. they would satisfy a request even if they could not get anything in return. This is exactly what happens in the latter stages where John satisfies Peter's request of $books\_returned(books)$ even though he no longer had any needs.

- **Non-Cooperative Computees:** On the contrary, if John had been a non-cooperative computee and therefore his KB_{GD}^{high} included the preference rule $gd\_pref_{non-coop,reject}^{request}$ instead of $gd\_pref_{coop,accept}^{request}$, then Peter would have not been able to achieve its other goals. In that case, John would refuse the request of Peter to return the books as he would not have had any terms to ask for in return. Note that John would still get what he wants – $working\_CDPlayer$ – by giving 10 euros in the first round of negotiation.

  Note that the only difference between a cooperative and non-cooperative computee is the modular local change in the KB_{GD}^{high} of replacing the rule $gd\_pref_{coop,accept}^{request}$ with $gd\_pref_{non-coop,reject}^{request}$.

Consider another variation example where the goal of returning books for Peter is of much higher priority (under the circumstances) and it was therefore the only one introduced initially in its state and so the only one when John sends its request. Then Peter gets the request from John to fix the CD player. Peter does not have any current needs as he can achieve his only goal of returning the books on his own using his 10 euros.

- **Cooperative Computees:**
  - At this stage since Peter has a cooperative KB_{GD} policy he will decide to unconditionally accept the request. Then John gets his working CDPlayer and Peter executes by himself the return of books.
  - After this Peter generates as a new top goal to have beer for which he needs an extra 20 euros from what he has. He plans for this via a request from John but now John, despite the fact that he is cooperative, he refuses because he cannot satisfy the request. Peter is left with one of his goals unachieved.
- **Non-Cooperative Computees:**
  - At this stage Peter with a non-cooperative KB_{GD} policy will reject the request as currenty he has no terms to ask in return.Therefore John will not be able to satisfy his goal.
  - After this Peter as above will also not be able to satisfy the new goal of beer as his request of 20 euros will again be rejected.

5 Examples for WP1 and WP2: Society expectations

In this section, we examine another variation of the standard example where now we bring in the factor of the Society in which the computees operate and the expectations that a society may raise on the computees.

Expectations are received by computees via Passive Observation introduction at some current time τ and therefore they are recorded in their KB_0 as facts of the form:

$$observed(E(A, T, C[T]), \tau)$$

where C[T] is a list of temporal constraints, e.g. $[T < t_{ans}]$, on the (existential) time T of the expectation. Note that we have added these inside the expectation so that they be recorded with the expectation.

- **State:** KB_{GD}^{low} The goal decision knowledge base KB_{GD} of each computee will contain the following rules in order to reason from expectations of the society to goals and actions that the society requires it to do.

$$\begin{array}{l} gd_{expectation}(A_g): \\ \quad goal(A_g[T']), T' < t_{ans} \leftarrow \\ \quad\quad holds\_at(E(A, T, [T < t_{ans}]), T_{now}), \\ \quad\quad unsatisfied(goal(A), T_{now}), \\ \quad\quad ground\_expectation(A, T_{now}, A_g). \end{array}$$

where $E(G, T, [T < t_{ans}])$ is an expectation that the computee has become aware off and updated its KB_0 via a Passive Observation transition. Note that this is really a schema to generate goals from expectations. For consistency of syntax since expectations are defined to be actions we have wrapped these with the special predicate "goal" to make them goals. We assume that these goals are planned simply by the action that they enclose.

The variables of an expectation are all existentially quantified so we need to ground them when we are generating goals from them. The auxiliary predicates $ground\_expectation(A, T_{now}, A_g)$ do this. It is defined for each type of expectation A specifically to ground the expectation. For example, for the expectation of replying with a message we have:

$$\begin{aligned}&ground\_expectation(tell(Asker, Receiver, message((accept, request(Request), []))), T_{now},\\&\quad tell(Asker, Receiver, msg((accept, request(Need), [])))) \leftarrow\\&\qquad unanswered(msg(from(Receiver), request(Need)), T).\end{aligned}$$

in order to ground the *Request* to the *Need* actually requested.

- **State:** KB_{GD}^{high} Now depending on the type of the computee its KB_{GD} will also contain priority rules to assign the relative preference of these goals generated via expectations. For example, a computee which is totally conforming to the expectations will have a priority rule:

$$gd\_pref_{Conformant} : h\_p(gd_{expectation}(\_), gd_{Name}(\_)) \leftarrow Name \neq expectation.$$

- **Society** $SOKB$ We assume that the society contains the following social integrity constraint for generating expectations about the behaviour of a computee according to the "demand" of its manager.

$$\begin{aligned}&IC_1 :\\&H(tell(Manager, Receiver, message(accept, requests(from(Asker), Request)), Deadline), T),\\&manager(Manager, Receiver) \quad \Rightarrow\\&EN(tell(Receiver, Asker, message(accept, Request, []), T_{ans}) : T_{ans} < Deadline.\end{aligned}$$

Hence when an event happens where a manager tells a computee to accept requests, e.g. an event:

$$H(tell(bob, peter, message(accept, requests(from(john), Request)), t_{ans}), 2)$$

where *bob* is the manager of *peter* then the expectation is raised for peter to accept *requests* from *john* before t_{ans}.

- **Behaviour:** We will consider now the standard example described above but where we will assume that before John executes its request to Peter for *working_CDPlayer* the society posts an expectation that Peter should accept unconditionally (without terms) requests from John when and if such such a request exists. This expectation is raised by an event of:

$$H(tell(bob, peter, message(accept, requests(from(john), Request)), t_{ans}), 1)$$

as described above. Hence the expectation will be:

$$E(tell(peter, john, message(accept, request(Request), [])), T, [T < t_{ans}])$$

where t_{ans} is a fixed deadline by which this expectation must be met by Peter.

– Peter via a POI transition, at time 3 say, will add the following observation in its KB_0:

  $observed(E(tell(peter, john, message(accept, request(n\_request), []))), T, [T < t_{ans}]), 3).$

  where *n_request* is a unique ground term that names the existential variable *Request*.
– Peter then receives, as in the standard example (via POI), the request for *working_CDPlayer* from John and adds this in its KB_0 exactly as before.
– The next transition for Peter is Goal Introduction (GI) to decide the reply to this request. But now the Goal Decision capability decides via the expectation generation rule and its priority over all other rules of KB_{GD}^{low} to accept unconditionally this request. The existential *n_request* request in the expectation has been grounded with the particular request of *working_CDPlayer*.

  Hence Peter updates its Goals with the communication goal:

  $$g^p_{4'} = \langle msg(to(john), (accept, request(working\_CDPlayer), []))[9], \perp, \{\}, \{\}\rangle$$

  Hence we see that get a different behaviour from the standard example where at this stage the goal $g^p_{4'}$ is added, instead of g^p_4, where Peter accepts but under the terms of 20 euros.
– From this stage onwards the example continues in an analogous way. John has a promise so he executes the get action to get the *working_CDPlayer* and then plays music. Peter is left with its goals unsatisfied still.
– Then Peter will start executing its tell actions to request the money that it needs for its goals. Now if John is non-cooperative he will refuse. Then Peter will do GR and PR and then PI to plan its goals in another way. If John is cooperative he will accept the request and so Peter will at least be able to satisfy one of its two goals now. Then it will do GR and PR and then PI to plan it one remaining goals in another way.

6 Conclusions

We have presented several examples within the Music for Beer scenario that demonstrate the behaviour of (interacting) computees. This serves to show (additionally to the companion document [Aea03b]) how the *KGP* model meets several criteria of Global computing as these were set out in the evaluation document of [LMM+03].

The main characteristics that are illustrate through these examples are the autonomy of computees to make their own decision in an open and changing environment, the heterogeneity of computees (cooperative or non co-operative), their ability to distribute their tasks and to adapt their decisions and behaviour in the face of new information e.g. as provided by the other computees.

The relatively complex scenario and examples illustrate how the different capabilities of the computee are composed together in order to produce the enhanced functionality required by the global computing setting. They show the synthesis of the isolated components of the model together with the cycle theory and how it is possible to exploit different parts of the knowledge of the computee specified *modularly* in different components of the model.

References

[Aea03a] M. Alberti and A. Bracciali et al. A computational approach to (societies of) computees. Technical report, SOCS Consortium, 2003. Deliverable D8.

[Aea03b] M. Alberti and A. Bracciali et al. Examples of the functioning of computees and their societies. Technical report, SOCS Consortium, 2003. Examples Document.

[KM03] P. Küngas and M. Matskin. Linear logic, partial deduction and cooperative problem solving. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003. Melbourne, Victoria, July 15th, 2003. Workshop Notes*, pages 97–112, 2003.

[KMS+03] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees. Technical report, SOCS Consortium, 2003. Deliverable D4.

[LMM+03] E. Lamma, P. Mello, P. Mancarella, A. Kakas, K. Stathis, and F. Toni. Self-assessment: parameters and criteria. Technical report, SOCS Consortium, 2003. Deliverable D3. Distribution restricted to the GC programme.

[Sta02] K. Stathis. Location-aware socs: The "leaving san vincenzo" scenario. Technical report, SOCS Consortium, 2002. IST32530/CITY/002/IN/PP/a2.

[Tor03] P. Torroni. A new example for the demonstration? Technical report, SOCS Consortium, 2003. Presentation Slides, SOCS Meeting, Ferrara, 15-17 September, 2003.

Appendix A: Normal Cycle theory

In this appendix we reproduce the *normal cycle theory* of computees essentially as given in deliverable [Aea03a].

The *normal cycle theory*, specifies a pattern of operation where the computee prefers to follow a sequence of transitions that allows it to achieve its goals in a way that matches an expected "normal" behaviour. Basically, the "normal" computee first introduces goals (if it has none to start with) via GI, then reacts to them, via RE, and then repeats the process of planning for them, via PI, executing (part of) the chosen plans, via AE, revising its state, via GR and PR, until all goals are dealt with (successfully or revised away). At this point the computeee returns to introduce new goals via GI and repeats the process above.

Whenever in this process the computee is interrupted via a passive observation, via POI, it chooses to introduce new goals via GI, to take into account any changes in the world. Whenever it has actions which are "unreliable", in the sense that their preconditions definitely need to be checked, the computee senses them (via SI) before executing the action. Whenever it has actions which are "unreliable", in the sense that their effects definitely need to be checked, the computee actively introduces actions that aim at sensing these effects, via AOI, after having executed the original actions.

We will also add to this at the end of this appendix an extra element in this normal behaviour to treat the processing of communication goal and actions with higher priority over other tasks.

The normal cycle theory has (as any other cycle theory) three parts $\mathcal{T}_{initial}$, $\mathcal{T}_{basic}$, $\mathcal{T}_{interrupt}$ and $\mathcal{T}_{behaviour}$ together with an auxiliary part.

- $\mathcal{T}_{basic}$ defines the basic cycle steps allowed by the theory. For the normal cycle theory this consists of the following rules.
  - The rules for deciding what might follow an AE transition are as follows:
    $r_{AE|PI}(S', Gs) : PI(S', Gs) \leftarrow AE(S, As, S'), Gs' = c_{GS}(S', \tau), Gs' \neq \{\}$
    $r_{AE|AE}(S', As') : AE(S', As') \leftarrow AE(S, As, S'), As' = c_{AS}(S', \tau), As' \neq \{\}$
    $r_{AE|AOI}(S', Fs) : AOI(S', Fs) \leftarrow AE(S, As, S'), Fs = c_{FS}(S', \tau), Fs \neq \{\}$
    $r_{AE|PR}(S') : PR(S') \leftarrow AE(S, S')$

    Namely, AE could be followed by another AE, or by a PI, or by an AOI, or by a PR. Any other possibility, e.g. for GR to follow AE, is excluded within this particular $\mathcal{T}_{basic}$ theory.
  - The rules for deciding what might follow GR are as follows
    $r_{GR|PR}(S') : PR(S') \leftarrow GR(S, S')$

    Namely, GR can only be followed by PR. Indeed, GR and PR are naturally coupled, since removing some goals in the state might lead to removing some actions.
  - The rules for deciding what might follow PR are as follows
    $r_{PR|PI}(S') : PI(S', Gs) \leftarrow PR(S, S'), Gs = c_{GS}(S', \tau), Gs \neq \{\}$
    $r_{PR|GI}(S') : GI(S') \leftarrow PR(S, S'), Gs = c_{GS}(S', \tau), Gs = \{\}$

    Namely, PR can only be followed by PI or GI, depending on whether there are goals to plan for or not in the state.
  - The rules for deciding what might follow PI are as follows
    $r_{PI|AE}(S', As) : AE(S', As) \leftarrow PI(S, Gs, S'), As = c_{AS}(S', \tau), As \neq \{\}$
    $r_{PI|SI}(S', Ps) : SI(S', Ps) \leftarrow PI(S, Gs, S'), Ps = c_{PS}(S', \tau), Ps \neq \{\}$

    The second rule is here to allow the possibility of sensing the preconditions of an action before its execution.
  - The rules for deciding what might follow GI are as follows
    $r_{GI|RE}(S', Gs) : RE(S', Gs) \leftarrow GI(S, S')$
    $r_{GI|PI}(S', Gs) : PI(S', Gs) \leftarrow GI(S, S'), Gs = c_{GS}(S', \tau), Gs \neq \{\}$

    Namely, GI can only be followed by RE or PI, if there are goals to plan for.
  - The rules for deciding what might follow RE are as follows
    $r_{RE|PI}(S', Gs) : PI(S', Gs) \leftarrow RE(S, S'), Gs = c_{GS}(S', \tau), Gs \neq \{\}$
    $r_{RE|SI}(S', Ps) : SI(S', Ps) \leftarrow RE(S, S'), Ps = c_{PS}(S', \tau), Ps \neq \{\}$
  - The rules for deciding what might follow SI are as follows
    $r_{SI|AE}(S', As) : AE(S', As) \leftarrow SI(S, Ps, S'), As = c_{AS}(S', \tau), As \neq \{\}$
    $r_{SI|PR}(S') : PR(S') \leftarrow SI(S, Ps, S')$

- The rules for deciding what might follow AOI are as follows

  $r_{AOI|AE}(S', As) : AE(S', As) \leftarrow AOI(S, Fs, S'), As = c_{AS}(S', \tau), As \neq \{\}$

  $r_{AOI|GR}(S') : PR(S') \leftarrow AOI(S, Fs, S')$

  $r_{AOI|SI}(S', Ps) : SI(S', Ps) \leftarrow AOI(S, Fs, S')Ps = c_{PS}(S', \tau), Ps \neq \{\}.$

- The $\mathcal{T}_{interrupt}$ component of any cycle theory consists of the following rules:

  $r_{POI|GI}(S') : GI(S') \leftarrow POI(S, S')$

  $r_{POI|RE}(S') : RE(S') \leftarrow POI(S, S')$

  $r_{POI|GR}(S') : GR(S') \leftarrow POI(S, S')$

  $r_{POI|POI}(S') : POI(S') \leftarrow POI(S, S')$

• The $\mathcal{T}_{behaviour}$ part of the normal cycle theory consists of the following rules:

- GI should be given higher priority if there are no goals in *Goals* and actions in *Plan* in the state: [3]

  $R^{T}_{GI|T'} : h\_p(r_{T|GI}(S), r_{T|T'}(S, X)) \leftarrow empty\_goals(S), empty\_plan(S)$

  for all transitions T, T', $T' \neq GI$.

- After GI, the transition RE should be given higher priority:

  $R^{GI}_{RE|T} : h\_p(r_{GI|RE}(S), r_{GI|T}(S, X))$

  for all transitions $T \neq RE$.

- After RE, the transition PI should be given higher priority:

  $R^{RE}_{PI|T} : h\_p(r_{RE|PI}(S, Gs), r_{RE|T}(S, X))$

  for all transitions $T \neq PI$;

- After PI, the transition AE should be given higher priority, unless there are actions in the actions selected for execution whose preconditions are "unreliable" and need checking, in which case SI will be given higher priority:

  $R^{PI}_{AE|T} : h\_p(r_{PI|AE}(S, As), r_{PI|T}(S, X)) \leftarrow not\, unreliable\_pre(As)$

  for all transitions $T \neq AE$;

  $R^{PI}_{SI|T} : h\_p(r_{PI|SI}(S, Ps), r_{PI|T}(S, As)) \leftarrow unreliable\_pre(As)$

  for all transitions $T \neq SI$;

  Here we assume that the auxiliary part of $\mathcal{T}_{cycle}$ specifies whether a given set of actions contains any "unreliable" action, in the sense described above.

- After SI, the transition AE should be given higher priority

  $R^{SI}_{AE|T} : h\_p(r_{SI|AE}(S, As), r_{SI|T}(S, X))$

  for all transitions $T \neq AE$.

- After AE, the transition AE should be given higher priority until there are no more actions to execute in *Plan*, in which case either AOI or GR should be given higher priority, depending on whether there are actions which are "unreliable", in the sense that their effects need checking, or not:

  $R^{AE}_{AE|T} : h\_p(r_{AE|AE}(S, As), r_{AE|T}(S, X))$

[3]Instead of the conditions $empty\_goals(S), empty\_plan(S)$ in this rule, we could have these conditions in each rule in $\mathcal{T}_{basic}$ which indicates as viable any transition that could be a competitor of GI after any given transition.

for all transitions $T \neq AE$. Note that, by definition of $\mathcal{T}_{basic}$ below, the transition AE is applicable only if there are still actions to be executed in the state.

$$R^{AE}_{AOI|T} : h\_p(r_{AE|AOI}(S, Fs), r_{AE|T}(S, X)) \leftarrow BC^{AE}_{AOI|T}(S, Fs)$$

for all transitions $T \neq AOI$, where the behaviour condition $BC^{AE}_{AOI|T}(S, Fs)$ is defined (in the auxiliary part) by:

$$BC^{AE}_{AOI|T}(S, FS) \leftarrow empty\_executable\_plan(S), unreliable\_post(S)$$

Similarly, we have:

$$R^{AE}_{GR|T} : h\_p(r_{AE|GR}(S), r_{AE|T}(S, X)) \leftarrow BC^{AE}_{GR|T}(S)$$

for all transitions $T \neq GR$ where:

$$BC^{AE}_{GR|T}(S) \leftarrow empty\_executable\_plan(S), not\, unreliable\_post(S)$$

Here, we assume that the auxiliary part of $\mathcal{T}_{cycle}$ specifies whether a given set of actions contains any "unreliable" action, in the sense expressed by *unreliable_post*, and defines the predicate *empty_executable_plan*. Intuitively, $empty\_executable\_plan(S)$ succeeds if all the actions which can be selected for execution have already been executed.

– After GR, the transition PR should have higher priority:

$$R^{GR}_{PR|T} : h\_p(r_{GR|PR}(S, \_), r_{GR|T}(S))$$

for all transitions $T \neq PR$;

– After PR, the transition PI should have higher priority:

$$R^{PR}_{PI|T} : h\_p(r_{PR|PI}(S, Gs), r_{PR|T}(S))$$

for all transitions $T \neq PI$. Note that, by definition of $\mathcal{T}_{basic}$ below, the transition PI is applicable only if there are still goals to plan for in the state. If there are no actions and goals left in the state, then rule $R^{T'}_{GI|T}$ would apply.

– After any transition, POI is preferred over all other transitions:

$$R^{T}_{PI|T'} : h\_p(r_{T|POI}(S), r_{T|T'}(S, X))$$

for all transitions T, T', $T' \neq POI$, i.e. POI acts as an interrupt.

Extending the normal cycle theory

The normal cycle theory can be extended to cove an extra element in its normal behaviour to treat the processing of communication goal and actions with higher priority over other tasks. This extension is given by the following additional rules in the theory T^{normal}_{cycle}.

In the normal cycle theory after a passive observation (POI) the preferred next transition is goal introduction (GI), as ensured by the following priority rule ($T \neq GI$):

$$h\_p(r_{POI|GI}(S'), r_{POI|T}(S, S')).$$

In particular, this means that computees perform a goal introduction immediately after they receive new messages from other computees. The basic rule $r_{POI|GI}(S, \_)$ that introduces the GI transition as the next transition is given by:

$$r_{POI|GI}(S) : GI(S') \leftarrow POI(S, Obs, S'), non\_terminal(Obs).$$

where $non\_terminal(Obs)$ is defined using the auxiliary predicate $terminal/1$ which specifies which communication messages received need no reply (see appendix for details).

More generally, we can impose that if a computee has unanswered messages then this will give higher priority to goal introduction (GI) in order to decide how to answer these messages. We achieve this by the following preference rule:

$$h\_p(r_{*|GI}(S,\_), r_{*|*}(S,\_)) \leftarrow unanswered(msg(from(\_), Message), T_{now}).$$

where $unanswered(msg(from(\_), Message), T_{now})$ is an auxiliary predicated (see appendix) defined via the temporal reasoning capability of the computee and T_{now} is the current time at which the cycle theory is called to decide the next transition.

In addition, the normal pattern of behaviour that we will use in our examples gives preference to planning communication goals and executing communication actions. This is accomplished via the following preference rules:

$$h\_p(r_{GI|PI}(S,Gs), r_{GI|*}(S,\_)) \leftarrow communication\_goal(Gs).$$

where $communication\_goal/1$ is a heuristic function that selects messaging goals, i.e. goals of the form $msg(to(\_),\_)$, and:

$$h\_p(r_{*|AE}(S,As), r_{*|*}(S,\_)) \leftarrow communication\_action(As).$$

where $communication\_action/1$ is also a heuristic function that selects communication actions, i.e. actions of the form $tell(\_,\_)$.

Appendix B: Knowledge Bases of Example Computees

In this appendix we give more details on the different knowledge bases and the cycle theory of the two computees of Peter and John.

The KB_{GD} knowledge base

For the standard example both computees will have the same theory of KB_{GD} encoding the same policy of deciding how to respond to requests and negotiate. Informally, this policy says:

> Accept a request for a need when this does not invalidate your current tasks. Prefer to get in return for accepting request a need, i.e set terms for accepting, that you (currently) have. Accept terms when these do not invalidate your current tasks otherwise if they do ask for alternative terms. When no alternative terms are offered then you can either accept the last term offerred or refuse depending on other criteria relevant to the your needs and terms. Also you may have reasons to prefer some terms over other terms.

Needs and Terms are both a list of fluent literals referring to a special set of fluents. Terms can be possibly the empty list. Requests, *Request*, can be any need. A special case of a need fluent is *has*(*Resource*) where *Resource* names a resource.

Below the fluent $msg(to(ComputeeName), Message)$ will denote the property that the computee $ComputeeName$ has the $Message$, e.g. a request or a response. Similarly, for $msg(from(ComputeeName), Message)$.

A KB_{GD} theory consists of three parts, KB_{GD}^{low}, KB_{GD}^{high} and KB_{GD}^{aux}.

- KB_{GD}^{low}: There are two possible answers for a request, i.e. either to accept (with terms) or reject the request.

  - A computee can reject a request [4]:

$$
\begin{aligned}
& gd_{reject}^{request,init}(Peer, Reply):\\
& \quad msg(to(Peer), Reply)[RT] \leftarrow\\
& \qquad unanswered(msg(from(Peer), request(Request)), T_{now}),\\
& \qquad Reply = (reject, request(Request), []),\\
& \qquad response\_time(T_{now}, RT).
\end{aligned}
$$

  - A computee could accept a request only if it is able to satisfy the given request. A computee accepts the request under some $Terms$ that are part of its own (current) needs which it asks in return:

$$
\begin{aligned}
& gd_{accept}^{request,init}(Peer, Reply):\\
& \quad msg(to(Peer), Reply)[RT] \leftarrow\\
& \qquad unanswered(msg(from(Peer), request(Request)), T_{now}),\\
& \qquad satisfiable(Request),\\
& \qquad current\_terms(Terms),\\
& \qquad Reply = (accept, request(Request), Terms),\\
& \qquad response\_time(T_{now}, RT).
\end{aligned}
$$

Note that the list of $Terms$ could be empty in which case the acceptance of the $Request$ is unconditional. Here the various predicates that appear in the body of this rule are auxiliary predicates that belong to KB_{GD}^{aux} and may refer to the current state ($Goals$ and $Plan$) of the computee. Some like $satisfiable/1$ use the temporla reasoning capability of the computee.

When a computee is offered $Terms$ to its request this becomes a request back to it. It can either accept these (when it can satisfy them) or it can ask for alternative terms, e.g. when it cannot satisfy them or when they are incompatible with its current needs, etc.

  - The following rule captures the case when the computee can satisfy the required terms:

$$
\begin{aligned}
& gd_{accept}^{response}(Peer, Reply)[RT]:\\
& \quad msg(to(Peer), Reply) \leftarrow\\
& \qquad unanswered(msg(from(Peer), (accept, request(Request), Terms)), T_{now}),\\
& \qquad satisfiable(Terms),\\
& \qquad Reply = (accept, response(Request, Terms), []),\\
& \qquad response\_time(T_{now}, RT).
\end{aligned}
$$

[4]Here and below the predicates *unanswered* and *response_time* are auxiliary. The first uses temporal reasoning to evaluate it and we will see more details about it when we describe KB_{TR}.

– The case of asking for alternative terms is given by the following rule:

$$\begin{array}{l} gd^{response}_{reject}(Peer, Reply): \\ \quad msg(to(Peer), Reply)[RT] \leftarrow \\ \quad\quad unanswered(msg(from(Peer), (accept, request(Request), Terms)), T_{now}), \\ \quad\quad unsatisfiable(Terms), \\ \quad\quad Reply = (reject, response(Request, Terms), alt\_terms), \\ \quad\quad response\_time(T_{now}, RT). \end{array}$$

To complete the policy we need to formulate the response policy for alternative terms.

– A computee can offer alternative terms when requested to do so:

$$\begin{array}{l} gd^{request,alt}_{accept}(Peer, Reply): \\ \quad msg(to(Peer), Reply)[RT] \leftarrow \\ \quad\quad unanswered(msg(from(Peer), (reject, response(Request, Terms), alt\_terms)), T_{now}), \\ \quad\quad alternative\_terms(Alt\_Terms), \\ \quad\quad Reply = (accept, request(Request), Alt\_Terms), \\ \quad\quad response\_time(T_{now}, RT). \end{array}$$

Note that *alternative_terms*/2 (an auxiliary predicate) may return an empty list and thus this rule also provides t he option for the computee to eventually accept the initial request with no terms in return (after all its terms offered were rejected).

– If the computee has no alternative terms left to offer then it can reject the request.

$$\begin{array}{l} gd^{request,alt}_{reject}(Peer, Reply): \\ \quad msg(to(Peer), Reply)[RT] \leftarrow \\ \quad\quad unanswered(msg(from(Peer), (reject, response(Request, Terms), alt\_terms)), T_{now}), \\ \quad\quad alternative\_terms(Alt\_Terms), Alt\_Terms = [], \\ \quad\quad Reply = (reject, request(Request), []), \\ \quad\quad response\_time(T_{now}, RT). \end{array}$$

Note that these two last rules are two condradictory options of reply when there are no alternative terms left.

Finally, we show here other rules that KB^{high}_{GD} may contain (although these will not take part in the examples of this document) that relate to the generation of new goals for a computee as a result of promises that it has made.

A promised need becomes an own goal in order to be able to deliver this as promised. Hence we have:

$$\begin{array}{l} gd_{promise}(Peer, Need) : \\ \quad delivery(Need, to(Peer))[RT] \leftarrow \\ \quad\quad holds\_at(promise(to(Peer), Need), T_{now}), \\ \quad\quad delivery\_time(Need, T_{now}, RT). \end{array}$$

where $delivery(Need, to(Peer))$ are goals of a special type, that informally are satisfied when a computee satisfies the goal $Need$ and then with a final action delivers (or gives) this to $Peer$.

- KB^{high}_{GD}: The following priorities are applied on the generation rules above to complete the preference policy. In general, a computee prefers to accept requests, ie. the rule $gd^{request,init}_{accept}$ has priority over the rule $gd^{request,init}_{reject}$.

$$\begin{array}{l} gd\_pref^{request}_{accept}(Peer, Reply) : \\ \quad h\_p(gd^{request}_{accept}(Peer, Reply), gd^{request}_{reject}(Peer, Reply')). \end{array}$$

Computees also prefer to accept with some terms over accepting uncoditionally, i.e. with empty terms:

$$\begin{array}{l} gd\_pref^{request}_{accept,cond}(Peer, Reply) : \\ \quad h\_p(gd^{request,*}_{accept}(Peer, (accept, request(R), Terms)), \\ \quad\quad gd^{request,*}_{accept}(Peer, (accept, request(R), []))) \leftarrow Terms \neq []. \end{array}$$

Accepting with empty $Terms$ has no personal gain for a computee and thus a computee may prefer to reject the request over accepting it with no Terms:

$$\begin{array}{l} gd\_pref^{request}_{reject}(Peer, Reply) : \\ \quad h\_p(gd^{request,*}_{reject}(Peer, Reply), \\ \quad\quad gd^{request,*}_{accept}(Peer, (accept, request(R), []))). \end{array}$$

Then higher order preferences can be used to distinguish between a *cooperative* or a *non-cooperative* computee. A simple example for a cooperative computee gives preference to accepting a request even with no terms:

$$\begin{array}{l} gd\_pref^{request}_{coop,accept}(Peer, Reply) : \\ \quad h\_p(gd\_pref^{request}_{accept}(Peer, Reply), gd\_pref^{request}_{reject}(Peer, Reply')). \end{array}$$

A non cooperative computee gives preference to rejecting a request over accepting it with no terms. This is captured by::

$$\begin{aligned}&gd\_pref^{request}_{non-coop,reject}(Peer, Reply):\\&\quad h\_p(gd\_pref^{request}_{reject}(Peer, Reply), gd\_pref^{*}_{accept}(Peer, (accept, request(\_), []))).\end{aligned}$$

This part of the theory would contain also priority rules to specify the preference between own goals and promised goals (generated by $gd_{promise}$ rules) for other computees, depending on the personality of the computee and the relative role of the other computees to which the promise has been made. As this is beyond the scope of this document we will not give any details here.

- KB^{aux}_{GD}:

  KB^{aux}_{GD} contains the definition of the special predicate $incompatible/2$ that renders any two distinct reply goal messages incompatible:

  $incompatible((msg(to(Peer), Reply), T), (msg(to(Peer), Reply'), T)) \leftarrow different(Reply, Reply')$

  where $different/2$ is defined in the obvious way.

  KB^{aux}_{GD} also includes definitions, in terms of an ordinary (normal - but deterministic) logic program, of the following auxiliary predicates:

  - $current\_terms/1$ – Returns a list of terms based on the current needs of the computee as in its current state. If there are no current needs the returned list is empty.
  - $alternative\_terms/2$ – Returns a list of terms based on the current needs of the computee, however, excluding a given list of terms (those terms already rejected by the other computeee).
  - $satisfiable/1$ – Checks whether it can satisfy a given list of predicates/needs. This can make use the temporal reasoning capability, e.g. for needs that are resources it checks whether currently it has such a resource. For more complicated needs it can also check to see if it can satisfy them through a viable plan using its planning capability.
  - $unsatisfiable/1$ – True whenever any of the given predicates/needs cannot be satisfied.
  - $unanswered/2$ – Checks whether a message is unanswered at a given timepoint, i.e. if the computee has not responded to it already. This uses the temporal reasoning capability. It is explained below in KB_{TR}.
  - $communication\_goal/1$ – Returns a communication goal currentlly present in the state of the computee.
  - $communication\_action/1$ – Returns a communication action currentlly present in the state of the computee.
  - $response\_time/2$ – Given a timepoint (e.g. the current timepoint), it computes an appropriate response time for a message.
  - $delivery\_time/3$ – Given a timepoint (e.g. the current timepoint), and a need it computes an appropriate delivery time for the need.

Some of these auxixliary predicates are also used by the cycle theory as well.

The KB_{plan} knowledge base

Both computees contain a domain independent part in their KB_{plan} to help them plan to achieve their message communication goals, i.e. plan how to send a message. We will assume that this is very simple and that it is simply captured via the rule:

$$initiates(tell(Peer, msg(from(self), Message)), T, msg(to(Peer), Message)).$$

i.e. to satisfy a goal of $msg(to(Peer), Message)$, we generate a simple plan with the only action of $tell(Peer, msg(from(Self), Message))$.[5]

More importantly, their KB_{plan} contains the following domain independent rules:

$$\begin{array}{l} initiates(get(from(Peer), Need), T, Need). \\ precondition(get(from(Peer), Need), ass\_promise(from(Peer), Need))). \end{array}$$

that allows them, during planning to generate an action *get* in their plan, when a computee wants to satisfy a goal or subgoal, *Need*, that it knows it *could* be provided by some computee. In order to execute such a *get* action the computee needs the precondition of a promise from the *Peer* computee from whom it will get the *Need*. Therefore, in a plan *get* will need to be preceded by an action that generates the promise. This is accomplished by the rule:

$$\begin{array}{l} initiates(tell(Peer, msg(from(self), request(Need))), T, \\ \quad ass\_promise(from(Peer), Need)) \end{array}$$

This $ass\_promise(from(Peer), Need)$ can be terminated by an action from the *Peer* that refuses the requested *Need*. Specifically,

$$\begin{array}{l} terminates(tell(Self, msg(from(Peer), (reject, request(Need), []))), T, \\ \quad ass\_promise(from(Peer), Need)) \end{array}$$

Hence in this way when a computee plans for a (sub)goal, which it knows can be asked from some other computee, it can generate a plan consisting of a *tell* action to request this and assuming that this will succeed (i.e. the request will be accepted) a *get* action will achieve the (sub)goal.

It is important though that a computee does not execute a $get(from(Peer), Need)$ action until it has a real promise for the $Need$ after its request for this. One way to ensure that the computee will indeed wait before executing the *get* action is to include a suitable check in the core action selection function, of the cycle theory, so that $get(from(Peer), Need)$ is selected only if $holds\_at(promise(from(Peer), Need), T_{now})$ is true.

Specifically, the core selection function c_{AS} would need an extra condition $check\_promise/1$ in its definition when the selected action is a *get* action. This is defined as follows:

$$\begin{array}{l} check\_promise(A) \leftarrow \\ \quad A = \langle (get(from(Peer), Need),)\ \_, \_, \_\rangle, \\ \quad holds\_at(promise(from(Peer), Need), T_{now}) \end{array}$$

[5] For ease of presentation, we have used a tell action with only two arguments. An action $tell(Peer, msg(from(Self), Message))$ stands for $tell(Self, Peer, msg(from(Self), Message), D)$ where D is a unique dialog id.

The computees also have an analogous standard planning theory for *deliver* actions.

$$initiates(deliver(Need, to(Peer)), T, delivery(Need, to(Peer))).$$
$$precondition(deliver(Need, to(Peer)), Need))).$$

where again the cycle theory will not allow a deliver action to be selected unless Need holds. Deliver actions when executed terminate (fulfil) the promise:

$$terminates(deliver(Need, to(Peer)), T, promise(to(Peer), Need)).$$

KB_{plan} - Domain Dependent

Following the example as presented in [KM03] the two computees have the following simple specific planning knowledge that captures what [KM03] calls the capabilities of each agent(computee).

- KB_{plan} for computee John: John can return books and play music. This means that it knows what these actions do. We express this simply by

$$initiates(return(Books), T, books\_returned(Books))$$
$$initiates(play(CD), T, music) \leftarrow holds\_at(working\_CDPlayer, T)$$
$$precondition(return(Books), has(Books))$$
$$precondition(play(CD), has(CD))$$
$$terminates(play(CD), T, has(CD))$$

- KB_{plan} for computee Peter: Peter can repair cd players, return books and buy beer. Its planning knowledge base thus contains:

$$initiates(return(Books), T, books\_returned(Books)) \leftarrow holds\_at(has(money(books, 10), T)$$
$$initiates(buy(beer), T, has(beer)) \leftarrow holds\_at(has(money(beer, 25), T)$$
$$initiates(repair\_CDPlayer, T, working\_CDPlayer)$$
$$precondition(return(Books), has(Books))$$
$$precondition(repair\_CDPlayer, neg(working\_CDPlayer))$$
$$terminates(buy(beer), T, has(money(beer, 25)))$$
$$terminates(return(books), T, has(money(books, 10)))$$

Note that we could use here a general theory of how needs which are having resourses, e.g. $has(money(\_)$ or $has(CD)$ are consumed and when these are available or not. This is not importnat for the examples and it is beyond the scope of this example document.

The KB_{TR} knowledge base

A computee keeps a record (via Passive Observation and Action Execution transitions) of requests and responses as executed actions (i.e. events) in the KB_0 part of its knowledge base alongside other actions that it executes, e.g. the *get* and *deliver* actions. The KB_{TR} of the computee is then extended with the following axioms so that the computee can reason from these events to promises held by itself to other computees and vice-versa.
For an unconditional acceptance (i.e. without terms), we have the rules:

$$\begin{aligned}&initiates(tell(Peer, msg(from(Self), (accept, request(Request), [])), T,\\&\quad promise(to(Peer), Request)\\&initiates(tell(Self, msg(from(Peer), (accept, request(Request), [])), T,\\&\quad promise(from(Peer), Request).^6\end{aligned}$$

Similarly, for promises for terms offered, we have the following two rules:

$$\begin{aligned}&initiates(tell(Self, msg(from(Peer), (accept, response(Request, Terms), [])), T,\\&\quad promise(from(Peer), Terms)\\&initiates(tell(Peer, msg(from(Self), (accept, response(Request, Terms), [])), T,\\&\quad promise(to(Peer), Terms).\end{aligned}$$

The first rule generates a promise from the $Peer$ computee since it has accepted the computee's $Terms$. The second rule generates a promise to the $Peer$ computee for the $Terms$ it has asked in exchange of the $Request$ from the computee.

For conditional acceptance the response only initiates a conditional promise pending (forward in time) to the acceptance of the terms asked for. So we have for example:

$$\begin{aligned}&initiates(tell(Peer, msg(from(Self), (accept, request(Request), Terms))), T,\\&\quad cond\_promise(to(Peer), Request, Terms))\end{aligned}$$

where $cond\_promise(to(Peer), Request, Terms)$ means that the computee has promised the $Request$ provided that the other computee shall promise back the $Terms$. Also

$$\begin{aligned}&initiates(tell(Self, msg(from(Peer), (accept, response(Request, Terms), []))), T,\\&\quad cond\_promise(from(Peer), Request, Terms))\end{aligned}$$

where $cond\_promise(from(Peer), Request, Terms)$ means that the computee has promised the $Terms$ in exchange of the computee to satisfy the $Request$.

The conditional promise of a request under some terms together with an unconditional promise of the terms link together to give an unconditional promise of the request. This is captured by the following rules:

$$\begin{aligned}&initiates(tell(Self, msg(from(Peer), (accept, response(Request, Terms), []))), T,\\&\quad promise(to(Peer), Request)) \leftarrow\\&\qquad holds\_at(cond\_promise(to(Peer), Request, Terms), T).\end{aligned}$$

$$
\begin{array}{l}
initiates(tell(Peer, msg(from(Self), (accept, response(Request, Terms), []))), T, \\
\quad promise(from(Peer), Request)) \leftarrow \\
\qquad holds\_at(cond\_promise(from(Peer), Request, Terms), T).
\end{array}
$$

The book keeping of recognizing to which requests/responses a computee has already replied to and which ones are left unanswered is done via the temporal reasoning capability, using the auxiliary predicate $unanswered/2$.

An unanswered message is defined as a message which was introduced in the state of the computee as a passive observation and there does not exist a corresponding communication action to reply back to the sender of the message. This is easily captured via the following temporal reasoning rules in the KB_{TR} of the computee:

$$
\begin{array}{l}
unanswered(msg(from(Sender), Message), T) \leftarrow \\
\quad holds\_at(msg(from(Sender), Message), T), \\
\quad not\, terminal(Message). \\
terminates(tell(Peer, msg(from(Self), Reply)), T, msg(from(Peer), Message)) \leftarrow \\
\quad isreplyto(Reply, Message).
\end{array}
$$

The predicate $terminal/1$ is an auxiliary predicate that defines which messages do not need a reply and hence they are considered answered.

$$
\begin{array}{l}
terminal((accept, request(\_), X)). \\
terminal((accept, response(\_, \_), [])). \\
terminal((reject, request(\_), [])).
\end{array}
$$

The $isreplyto/2$ predicate defines which message is a reply to a received message.[7]

$$
\begin{array}{l}
isreplyto(Reply, Message) \leftarrow \\
\quad Reply = (\_, request(Request), \_), \\
\quad Message = request(Request). \\
\\
isreplyto(Reply, Message) \leftarrow \\
\quad Reply = (\_, response(Request, \_), \_), \\
\quad Message = (\_, request(Request), \_). \\
isreplyto(Reply, Message) \leftarrow \\
\quad Reply = (\_, request(Request), \_), \\
\quad Message = (reject, response(Request, \_), alt_terms).
\end{array}
$$

More generally, this definition of $isreplyto/2$ would refer to unique message ids which we are not using here for simplicity. The reference in our case is the content of the message itself, i.e. $request(Need)$ or $response(Need, Terms)$.

[7]Note that, $isreplyto/2$ is used here in KB_{TR} as an auxiliary predicate which in the strict syntax of KB_{TR} is not allowed. If we want to use the strict syntax, we can apply the definition of $isreplyto/2$ inside the head of the aforementioned rule for $terminates/2$.