Towards an Expert System for the Analysis of Multiple Agent Task Descriptions

A. N. Clark, G. L. Greatorex & A. G. Hill,
Computer Systems Division, GEC-Marconi Research Ltd.,
Great Baddow, Chelmsford, Essex, CM2 8HN, Great Britain
e-mail: anc@gec-mrc.co.uk, tel.: (0245) 473331

1 Abstract

2 Introduction

Some text to be inserted: Introducing the need for Human Factors in general, Task Descriptions and Analysis. Many tasks have common features which indicates the possibility of standard representations. Task Analysis is performed by a HF expert but is time consuming and has many repetitive features. Tasks can get complex and the analysis is error prone especially in the area of agent interaction such as occurs between a human and a computer based assistant. This area is ripe for computer based support in the form of an expert system for analysing task descriptions. Some text describing the competence of GEC-Marconi in this general area. Plan of paper.

3 Scope and Extent

This paper addresses the fundamental issues which are involved in precisely describing a *task* with a view to providing a basis for computer aided support of the task analysis which is performed by a Human Factors expert. In order to perform any such analysis it is important to have a precise notion of what is meant by all of the relevant terms, in this case tasks, agents, workload, agent interaction and sub-tasks. It is also important to define what is meant by task execution in order to give a complete picture of what is available for analysis, in terms of producing different properties of the task, constraining the task to behave in a particular way and observing the effect which modifications to the task description have on execution. Once such a complete picture is provided it will be possible to precisely define what is meant by terms such as workload analysis, validity and timeline analysis. Furthermore, a complete description of task representation and execution will permit analysis which detects potential problems in task descriptions arising, for example, from workload overloading, man-in-the-loop and stale information. As a consequence of this work, we hope to be able to employ Expert Systems technology to detect problems with task descriptions and to suggest modifications which can improve them.

We provide a simple but valid representation and meaning for tasks and an example which involves workload analysis. It is beyond the scope of this paper to thoroughly investigate all the possible different tasks including optional tasks, probabilities associated with tasks, task execution time and task exceptions; also we present a necessarily restricted view of task workload and agent overwork since this is a very complex area. We hope that the basic description of task components and workload analysis which is given, convinces the reader that this technology provides the basis for a broad range of computer aided task analysis.

We present a representation for task descriptions and a meaning for task execution in terms of extensions to a simple functional programming language. This representation and execution mechanism is not essential to our approach and is presented in order to give a concrete implementation of all the necessary terms. A functional language is particularly amenable to this approach because it has a simple representation and meaning and it is readily extended with new and exotic programming features.

4 Tasks

When we think of a *task* it is usual to envisage a series of actions which are performed, using a collection of tools and resources, in order to achieve a particular goal. For example, flying an aircraft along a preset course from waypoint to waypoint. Along the way there may be sub-tasks to be performed before the overall goal is achieved and it may not always be clear in which order the sub-tasks have to be performed. For example, whilst flying the aircraft, the pilot is responsible for broadcasting the aircraft position at preset times and maintaining the correct altitude along with other sub-tasks. Often it will not matter which order these sub-tasks are performed so long as the aircraft gets to the intended destination at the required time.

Each task is performed by a collection of *agents*, in the flying example the task is performed by a pilot, possibly in conjunction with a co-pilot and a navigator. Agents work concurrently to achieve a collection of tasks, interacting where necessary to form a composite agent which achieves an overall, larger task. Each agent proceeds by observing and acting upon attributes of the task, for example a pilot flies the plane by moving switches, pushing levers, requesting information, reading head up displays, *etc.* Agents interact by sending each other messages, for example the pilot requests the next waypoint from the navigator by sending a "*next waypoint*" message which will produce the reply message "*next waypoint at X*".

An agent performs a task by expending *workload* which is a crude measure of how difficult the agent finds the task. The amount of workload depends upon the agent and the task. For example the pilot will expend a large amount of workload while flying past a waypoint because the aircraft must be manoeuvered into the course for the next waypoint and this involves keeping track of many instrument readings *etc.* An agent is capable of performing more than one task in parallel, each of which is associated with its own workload. Each agent has a workload threshold above which the likelihood of successfully performing further tasks in parallel is greatly reduced. For example, if a pilot is approaching a waypoint and starting the manoeuvres which are necessary for the new course then it is unwise to start the procedure for position broadcast at the same time, because one or other of the tasks are likely to suffer.

Each agent interacts with the task attributes by making decisions based on its understanding of the situation at the time. The validity of the agent's decisions depends upon whether or not the information which it uses to make the decision is up-to-date or not. Many of the task attributes will change frequently and it is important that all agents which make decisions based upon these attributes keep their knowledge of the values current by monitoring the attributes at frequent intervals. For example, if the pilot aims to fly at a constant altitude then it is important that the altimeter is monitored frequently otherwise the decisions which are made involving increasing or decreasing the altitude will be invalid.

4.1 Task histories

The execution of a task produces a *task history* which is the sequence of steps from the start of the task to the end (which may or may not achieve the desired goal). The task history contains all of the information which is used at every stage of the task, showing, for example, messages which are sent between agents, the calculations which lead to decisions being made and actions which affect the task attributes. The task history for the aircraft example is like a perfect flight recorder which enables all of the steps of the flight to be played back in minute detail.

Given a task history, a Human Factors expert will be able to analyse the steps which were performed in order to identify evidence of bad task design. For example, certain tasks may not be achieved because the pilot tries to perform too many activities at once. In this sense, the Human Factors expert is debugging the trace of a buggy task execution – the analysis may be performed because the overall goal is not achieved and the expert is trying to trace back to the place in the history at which the task starts to go wrong; alternatively, the expert may be unhappy about the overall performance of some of the agents in the task and wishes to analyse the task history in order to identify areas where the agents can be made to perform better.

A task history is a sequence of steps which lead from the initiation of the task to the outcome (or goal) which is produced by the execution of the task. There are a number of different types of steps which are given in figure 1. Each task may initiate one or more sub-tasks either in sequence or in parallel. For example in order to broadcast the aircraft position, the pilot must request the current position from the navigator and then broadcast the information; or in

The *history* of a *task* is a sequence of *steps* producing an *outcome*
where
A *step* is either
initiation of one or more sub-tasks; or
performing a decision; or
producing an outcome; or
updating an attribute; or
requesting an agent to perform a task; or
failing to perform a task

Figure 1: Definition of task history

order to fly the aircraft, the pilot must concurrently monitor the speed, altitude and direction instruments. A task is performed by an agent which includes a collection of attributes, each task has access to all of the agent's attributes and may update them.

4.2 Agents

An agent is a value which is capable of initiating tasks. Agents are the values of task attributes and therefore tasks know about collections of agents and vice versa. Each agent is associated with a workload value which represents the limit of how many tasks the agent can perform concurrently. Above this threshold there is no guarantee that the agent will successfully perform a requested task. Initially, all agents are inactive, then as the overall task progresses, agents are requested to perform individual tasks, each of which may initiate sub-tasks. Agents may interact and send each other messages in order to co-operate when achieving tasks. When an agent receives a message it will attempt to initiate the request immediately. Any task performed by an agent may fail in whole or in part because of workload restrictions.

Figure 2 defines an agent. A system is a collection of agents; initially each agent is inactive and the system starts by concurrently activating one or more of the named tasks of each agent. Each of the activated tasks will proceed independently. A task may be a choice between two subtasks, in which case the subtask which is activated will depend upon the current values of the agent's attributes; tasks may be sequenced or activated in parallel; a task may update one of the agent's attributes; finally, a task may request another agent to perform a named task in order to produce an outcome in which case the activating task is referred to as the *client* and the activated task is referred to as the *server*.

An *agent* is
a collection of attributes; and
a workload limit; and
a collection of named tasks
where a *task* is either
a choice between two subtasks; or
a sequenced pair of subtasks; or
a parallel pair of subtasks; or
an attribute update; or
a request for another agent to perform a named task

Figure 2: Definition of an agent

At any time, an agent is performing 0 or more tasks. Each atomic task is associated with an effort value which describes how the task effects the agent in terms of workload. If any step performed by an agent would cause the aggregate of the effort values to exceed the workload limit for that agent then that step can fail. Failure of a task step can cause an attribute not to be updated, to fail to request another agent to perform a task, to fail to make a proper choice between two subtasks or to fail to produce a requested outcome.

4.3 Task Analysis

Given a task description as a system of interacting agents which provide an interface of named tasks, a task history is the result of observing the entire execution of the task. Due to the possibility that tasks can occur concurrently and some depend upon conditions, a single task description gives rise to a collection of possible task histories which we can define as its meaning. Given a precise description of how tasks are described and what they mean, it is now possible to define properties of tasks, such as the possibility of agent overloading or whether certain tasks will always be performed before other tasks. Furthermore, it is possible to define when task descriptions will cause likely problems, for example, due to invalid information being available because one agent fails (because of overwork) to respond with up to date information when requested by another agent. Such a situation would occur when at least one of the task histories associated with a task description can be shown to fail due to an agent being overworked and thereby preventing a second agent from receiving up to date information which it requires in order to make a decision. In addition to detecting potential problems with task descriptions, using task histories as the meaning of the tasks allows us to observe what happens when task descriptions are modified. Expert Systems technology, in the form of heuristic rules, can be used to guide a program through a collection of task modifiers in order to remove the source of a problem.

Although we believe that the proposed task descriptions and meanings will allow a sophisticated degree of task analysis, we present a simple example in this paper based on agent overloading. Given the representation for agents given in figure 2 and the task execution mechanism described above, we will show how this analysis can be captured by constructing a workload graph which is inspected for maximum possible agent workload levels.

A *workload graph* is a collection of *agent nodes* each of which contains an agent and has an edge leading to a *task node* for each of the named tasks which the agent can perform. A task node contains the name of the task and the maximum effort which the task is likely to take. Each task node represents the possible executions of a task, since we are interested in workload, a task node is also associated with all of the possible combinations of server tasks which it can request in parallel. Each task node has a collection of edges, each of which leads to a set of task nodes.

A workload graph is *overloaded* if, given an initial collection of tasks, it is possible to exceed the workload limit of an agent node by constructing a collection of paths as follows: for each current task node, select a set of server task nodes which it activates in parallel, add these nodes to the current collection and repeat. The workload of an agent is exceeded when there is a path which activates enough of its tasks to cause it to potentially fail. If there is a path which loops through an agent's task, then without further information to the contrary this will cause the agent's workload limit to be automatically exceeded.

Having defined an agent and a workload graph, we now define how to construct the workload graph. Given an agent a, construct an agent node containing a with edges leading to a task node for each task t of a. Each task node contains the name of t and the maximum possible effort which the t can expend. For each set of tasks requested in parallel by t, the task node has an edge leading to the set of corresponding task nodes.

The maximum possible effort expended by a task t depends upon the construction of t: if t is a conditional task with subtasks t_1 and t_2 then the result is the maximum of the maximum effort of t_1 and t_2; if t is a sequence of two subtasks t_1 and t_2 then the result is the maximum of the effort of t_1 and t_2; if t is two subtasks t_1 and t_2 in parallel then the result is the aggregate of the effort for t_1 and t_2; if t is an attribute update then the effort is some predefined amount depending upon the attribute; finally, if t is a task request then the effort is some predefined amount depending upon the request.

The set of sets of parallel tasks requested by a task t depends upon the construction of t: if t is a conditional task with subtasks t_1 and t_2 then neither of the tasks requested by t_1 and t_2 can be performed concurrently so the result is the union of the tasks requested for both t_1 and t_2; if t is a sequence of two subtasks t_1 and t_2 then, similarly, the result is the union of the tasks requested for both t_1 and t_2; if t is two subtasks t_1 and t_2 in parallel then any of the tasks requested in parallel by t_1 can be performed in parallel with those of t_2; if t is an update then there are no requested tasks; finally, if t is a task request then the result is the task node corresponding to the requested task from the given agent.

5 Example scenario

The aircraft scenario is composed of three agents, the cockpit, the pilot and the navigator. The cockpit agent is responsible for maintaining the information relating to the speed, altitude, direction and joystick position. The cockpit provides instrument readings for the speed and altitude which are available to the pilot and navigator. The cockpit also provides a task which is used to update the joystick position. The main task of the cockpit is to regularly update the speed, altitude and direction values using the position of the joystick. The workload of the cockpit is unlimited.

The pilot agent is responsible for keeping the aircraft on course by monitoring the instrument dials and updating the position of the joystick through the cockpit agent. The pilot is also responsible for requesting the next waypoint from the navigator when the current waypoint is being approached. At predetermined waypoints, the pilot must broadcast the aircraft position. The pilot's workload limit and the effort for each task, are predetermined values.

The navigator agent is responsible for maintaining an up to date position of the aircraft and for responding to the pilot's request for the next waypoint.

Figure 3 is the specification of the cockpit agent. Since it is not a human resource it has unlimited workload capacity and maintains the attributes which describe the speed *etc.* of the aircraft. We assume that there is some notion of global time whereby any changes in the controls are implemented causing the speed *etc.* of the aircraft to be updated. The cockpit offers a task to update the position of the joystick, which is how the pilot flies the aircraft.

Figure 4 is the specification of the pilot agent. During the flight the pilot will monitor the instrument displays in order to check whether the joystick needs moving, whether a new waypoint is required and whether the aircraft position needs to be broadcast. The pilot is a client to both the cockpit and the navigator. Each of the tasks performed by the pilot has a predetermined effort.

The specification of the navigator agent (not shown) follows the same lines as the pilot and cockpit and provides tasks to get the current position and the next waypoint. The navigator is a client of the cockpit.

Figure 5 shows the workload graph for the cockpit scenario. There are three agent nodes labelled *Pilot*, *Navigator* and *Cockpit*, each of which is connected to task nodes for each of the named tasks which the corresponding agent can perform. Each task node is connected to a collection of nodes which in turn

A *cockpit* is an agent with unlimited workload
with attributes for altitude, speed, direction and joystick position; and
with tasks to perform the following
get the current altitude; and
get the current speed; and
get the current direction; and
update the position of the joystick; and
fly the aircraft which involves a condition
if it is time to update the displays
then perform the following tasks in sequence
update the altitude, speed and direction from the joystick position; then
request the cockpit to continue to fly the aircraft
else request the cockpit to continue to fly the aircraft

Figure 3: A specification of the cockpit agent

are connected to task nodes which can be invoked in parallel. As shown, there is a possibility of overload because each of the tasks *Next Waypoint*, *Position* and *Check Position* can be performed by the pilot in parallel through the *Start Flight* task. So when $\omega_p < \omega_9$ where $\omega_9 = \omega_6 + \omega_7 + \omega_8$ there is a possibility that the pilot will be overloaded.

In order to prevent overloading there must be some mechanism for preventing certain tasks from being performed in parallel. It is beyond the scope of this paper to discuss the various mechanisms for achieving this, but we intend to investigate simple extensions to agent descriptions which specify the legal combinations of tasks, such as provided by predicate path expressions [**?**].

6 Task representation

The entire state of an executing task is a directed cyclic graph and execution proceeds by graph transformations which add new nodes and edges, remove existing nodes and edges, replace nodes and reattach edges from one node to another. A node of type X has a collection of constructors, *e.g. consX*, which are applied to a collection of graph nodes $n_1, n_2, \ldots, n_k$ and produce new nodes x of type X with edges leading from x to each of the nodes $n_1, n_2, \ldots, n_k$. For each sub-node $n_1, n_2, \ldots, n_k$, the type X defines an operator f which has two purposes: when used in a *left hand expression*, $f(x)$, represents the path from the root of the graph, through x to the sub-node associated with f; when

A *pilot* is an agent with a maximum workload of ω_p
with attributes for altitude, speed, direction and the current waypoint
with tasks to perform the following
read the cockpit displays for altitude, speed and direction; and
check the current position which involves the following sequence of tasks
concurrently, check all instrument dials; then
depending on whether the aircraft is off course
move the joystick accordingly; then
continue to monitor the position; otherwise
just continue to monitor the position
depending upon whether the aircraft is approaching the waypoint
request the next waypoint from the navigator; then
update the joystick accordingly; then
continue to monitor the waypoint; otherwise
just continue to monitor the waypoint
depending on whether the current waypoint is prearranged
request the current position from the navigator; then
broadcast aircraft position information; then
continue to test whether the position should be broadcast; otherwise
just continue to test whether the position should be broadcast
start the flight by
initializing the attributes; then
throughout the flight check the position, the waypoint and whether to broadcast the position

Figure 4: A specification of the pilot agent

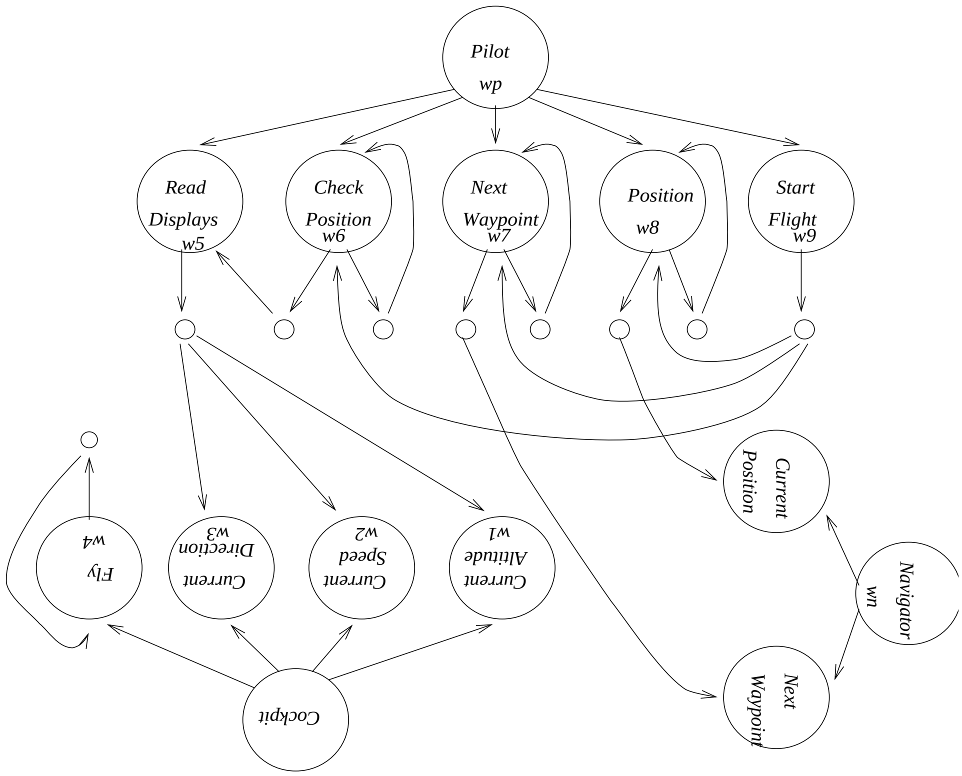

Figure 5: The workload graph for the cockpit example

An agent is a value $a = \mathit{consagent}(s, \omega, e_1, e_2, S)$

where

s is the formal parameter name;

ω is the maximum workload;

e_1 is an environment of inherited attributes;

e_2 is an expression which is the behaviour of the agent;

S is a set of activities

where

An activity is a value $a = \mathit{consactivity}(t_1, t_2)$

where

$t_1 = \mathit{client}(a)$ is a client task;

$t_2 = \mathit{server}(a)$ is a server task

where a task is a value $t = \mathit{constask}(\sigma, l, b)$

where

$\sigma = \mathit{consstate}(s, e, c, d)$ is a machine state

where

s is a sequence of values;

e is a value environment;

c is a sequence of expressions and instructions

where an instruction is either *app* or $\mathit{mktup}(n)$;

d is a dump which is either null, (), or a machine state;

l is a list of tasks which are subtasks of t;

b is a boolean determining the liveness of t

Figure 6: The definition of an agent

used in a *right hand expression*, $f(x)$, represents the sub-node associated with f. Left hand and right hand expressions are used to define side effects to the task graph. The infix operator := has arguments which are left hand and right hand expressions respectively. The expression $e_1 := e_2$ causes the path denoted by e_1 to lead to the node denoted by e_2 without affecting the rest of the graph.

A sequence of values of type X, l, is either *null* or a value $\mathit{cons}(x, l)$ with components $x = \mathit{hd}(l)$ and $l = \mathit{tail}(l)$ where x is of type X and l is sequence of X. Given a sequence l of values which contains x, $\mathit{pos}(x, l)$ is the path from the task graph root, through the root of l to the first occurrence of x in l. An atomic value is a string, an integer or a boolean. An X-environment is a collection of bindings between strings and values of type X. An environment is either empty *nullenv* or a value $\mathit{consenv}(e, s, x)$ where e is an environment, s is a string and x is a value of type X. The expression $\mathit{lookup}(e, s)$ is the path through the task

$$
\begin{array}{l}
\mathit{step}(t) = \\
\quad \textbf{if}\ \mathit{subtasks}(t) \neq \mathit{null} \\
\quad \textbf{then if}\ \mathit{every}(\mathit{complete})(\mathit{subtasks}(t)) \\
\quad\quad \textbf{then let}\ \mathit{consstate}(s, e, c, d) = \mathit{state}(t) \\
\quad\quad\quad\quad\quad \mathit{outcomes} = \mathit{map}(\mathit{hd} \circ \mathit{stack} \circ \mathit{state})(\mathit{subtasks}(t)) \\
\quad\quad\quad \textbf{in}\ \mathit{constask}(\mathit{constate}(\mathit{outcomes} +\!\!+\, s, e, c, d), \mathit{null}, \mathit{true}) \\
\quad\quad \textbf{else}\ \mathit{constask}(\mathit{state}(t), \mathit{map}(\mathit{cond}(\mathit{complete}, \mathbf{I}, \mathit{step}))(\mathit{subtasks}(t)), \mathit{true}) \\
\quad \textbf{else if}\ \mathit{active}(t) \\
\quad\quad \textbf{then if}\ (\mathit{hd}(\mathit{control}(\mathit{state}(t))) = \mathit{app})\ \&\ (\mathit{isagent}(\mathit{hd}(\mathit{stack}(\mathit{state}(t))))) \\
\quad\quad\quad \textbf{then let}\ \mathit{constask}(\mathit{constate}(\mathit{cons}(a, \mathit{cons}(v, s)), e, c, d), \_, \_) = t\ \textbf{in} \\
\quad\quad\quad\quad \textbf{let}\ t = \mathit{constask}(\mathit{consstate}(s, e, \mathit{tl}(c), d), \mathit{null}, \mathit{false}) \\
\quad\quad\quad\quad\quad \sigma = \mathit{consstate}(\mathit{null}, \mathit{consenv}(\mathit{env}(a), \mathit{formal}(a), v), \mathit{cons}(\mathit{body}(a), \mathit{null}), ()) \\
\quad\quad\quad\quad \textbf{in}\ \mathit{actions}(a) := \mathit{actions}(a) \cup \{\mathit{consactivity}(t, \mathit{constask}(\sigma, \mathit{null}, \mathit{true}))\}; t \\
\quad\quad\quad \textbf{else if}\ \mathit{isapply}(\mathit{hd}(\mathit{control}(\mathit{state}(t)))) \\
\quad\quad\quad\quad \textbf{then let}\ \mathit{consstate}(s, e, \mathit{cons}(\mathit{consapply}(e_1, e_2), c), d) = \mathit{state}(t))\ \textbf{in} \\
\quad\quad\quad\quad\quad \textbf{let}\ t_1 = \mathit{constask}(\mathit{consstate}(\mathit{null}, e, \mathit{cons}(e_1, \mathit{null}), ()), \mathit{null}, \mathit{true}) \\
\quad\quad\quad\quad\quad\quad t_2 = \mathit{constask}(\mathit{consstate}(\mathit{null}, e, \mathit{cons}(e_2, \mathit{null}), ()), \mathit{null}, \mathit{true}) \\
\quad\quad\quad\quad\quad\quad \sigma = \mathit{consstate}(s, e, \mathit{cons}(\mathit{app}, c), d) \\
\quad\quad\quad\quad\quad \textbf{in}\ \mathit{constask}(\sigma, \mathit{cons}(t_1, \mathit{cons}(t_2 \mathit{null})), \mathit{true}) \\
\quad\quad\quad\quad \textbf{else if}\ \mathit{istuple}(\mathit{hd}(\mathit{control}(\mathit{state}(t)))) \\
\quad\quad\quad\quad\quad \textbf{then let}\ \mathit{consstate}(s, e, \mathit{cons}(\mathit{tuple}(\mathit{exps}), c), d) = \mathit{state}(t)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ \mathit{states} = \mathit{map}(\lambda \mathit{exp}.\mathit{consstate}(\mathit{null}, e, \mathit{cons}(\mathit{exp}, \mathit{null}), d))\mathit{exps}\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ \mathit{tasks} = \mathit{map}(\lambda \sigma.\mathit{consstate}(\sigma, \mathit{null}, \mathit{true}))\mathit{states} \\
\quad\quad\quad\quad\quad\quad\quad \sigma = \mathit{consstate}(s, e, \mathit{cons}(\mathit{mktup}(\#(\mathit{exps})), c), d) \\
\quad\quad\quad\quad\quad\quad \textbf{in}\ \mathit{constask}(\sigma, \mathit{tasks}, \mathit{true}) \\
\quad\quad\quad\quad\quad \textbf{else}\ \mathit{constask}(\mathit{step}(\mathit{state}(t)), \mathit{null}, \mathit{true})
\end{array}
$$

Figure 7: Performing a single task step

graph root to the rightmost value associated with s in e when used as a left hand expression or is the value associated with s in e when used as a right hand expression. An expression e is either an identifier $id(s)$, a lambda expression $lambda(\omega, s, e)$, an application $apply(e_1, e_2)$ or a tuple expression $tuple(l)$ where l is a sequence of expressions. A builtin function is a value $bif(s)$. A value v is either a builtin function, an atomic value, a closure or an agent, where the type agent is defined in figure 6.

$$
\begin{array}{l}
step(consstate(cons(v, \_), \_, null, consstate(s, e, c, d))) = consstate(cons(v, s), e, c, d) \\
\\
step(consstate(s, e, cons(x, c), d)) = \\
\quad \textbf{if } isid(x) \textbf{ then } consstate(cons(lookup(name(x), e), s), e, c, d) \textbf{ else} \\
\quad \textbf{if } islambda(x) \textbf{ then } consstate(cons(conclosure(formal(x), e, body(x)), s), e, x, d) \textbf{ else} \\
\quad \textbf{if } isagent(x) \textbf{ then } consstate(cons(consagent(formal(x), workload(x), e, body(x), \{\}), s), e, c, d) \textbf{ else} \\
\quad \textbf{if } istup(x) \textbf{ then } consstate(cons(constuple(reverse(take(length(x), s))), drop(length(x), s)), e, c, d) \textbf{ else} \\
\quad \textbf{if } x = app \textbf{ then if } isclosure(hd(s)) \\
\qquad\qquad\qquad \textbf{then let } consclosure(n, e', b) = id(s) \\
\qquad\qquad\qquad\qquad\qquad v = hd(tl(s)) \\
\qquad\qquad\qquad\qquad \textbf{in } consstate(null, consenv(e', n, v), cons(b, null), consstate(tl(tl(s), e, c, d))) \\
\qquad\qquad\qquad \textbf{else if } isbif(hd(s)) \\
\qquad\qquad\qquad\quad \textbf{then if } name(hd(s)) = \text{“}loc\text{”} \\
\qquad\qquad\qquad\qquad \textbf{then } consstate(cons(consloc(hd(tl(s))), tl(s)), e, c, d) \textbf{ else} \\
\qquad\qquad\qquad\qquad \textbf{if } name(hd(s)) = \text{“}!\text{”} \\
\qquad\qquad\qquad\qquad \textbf{then } consstate(cons(contents(hd(tl(s))), tl(s)), e, c, d) \textbf{ else} \\
\qquad\qquad\qquad\qquad \textbf{if } name(hd(s)) = \text{“}:=\text{”} \\
\qquad\qquad\qquad\qquad \textbf{then } contents(hd(elements(hd(tl(s))))) := hd(tl(elements(hd(tl(s))))); \\
\qquad\qquad\qquad\qquad\qquad consstate(cons(hd(tl(s)), tl(tl(s))), e, c, d) \textbf{ else} \\
\qquad\qquad\qquad\qquad \textbf{if } name(hd(s)) = \text{“}task\text{”} \\
\qquad\qquad\qquad\qquad \textbf{then } consstate(tl(s), e, c, d)
\end{array}
$$

Figure 8: Performing a single machine state step

$$
\begin{array}{l}
step(a) = \\
\quad \textbf{let } P = workloadset(workload(a), activities(a)) \textbf{ in} \\
\quad \textbf{let } Q = activities(a) - P \textbf{ in} \\
\quad \textbf{let } R = /(complete)(P) \\
\qquad\quad S = /(complete)(Q) \\
\quad \textbf{in } activities(a) := (map(step)(P - R)) \cup (map(stumble)(Q - S)); \\
\qquad \textit{for each } a \in R, \textit{ return(a)}; \\
\qquad \textit{for each } a \in S, \textit{ lose(a)}
\end{array}
$$

Figure 9: Performing a single agent step

$$loading(a) = \backslash(+)(loading)(0)(activities(a))$$

$$loading(a) = loading(server(a))$$

$$
\begin{array}{l}
loading(t) = \\
\quad \textbf{if } active(t) \\
\quad \textbf{then if } subtasks(t) = null \\
\qquad \textbf{then if } (hd(control(state(t))) = app) \,\&\, (hd(stack(state(t))) = bif(\text{"}task\text{"})) \\
\qquad\quad \textbf{then } hd(tl(hd(stack(state(t))))) \\
\qquad\quad \textbf{else } 1 \\
\qquad \textbf{else } \backslash(+)(loading)(0)(subtasks(t)) \\
\quad \textbf{else } 0
\end{array}
$$

Figure 10: Calculating the loading on an agent

$$
\begin{array}{l}
\textbf{let rec } cockpit = \\
\quad \textbf{let } speed = loc(0) \\
\qquad alt = loc(0) \\
\qquad direction = loc(0) \\
\qquad joystick = loc(null) \\
\quad \textbf{in } \Lambda_\infty(m, v). \\
\qquad \textbf{if } m = \text{"}alt\text{"} \textbf{ then } !(alt) \textbf{ else} \\
\qquad \textbf{if } m = \text{"}updatejoystick\text{"} \textbf{ then } movejoystick(joystick, v) \textbf{ else} \\
\qquad \textbf{if } m = \text{"}fly\text{"} \textbf{ then if } timer(\text{"}nextclick\text{"}, ()) \\
\qquad\qquad\qquad \textbf{then } updatecockpit(speed, alt, direction, joystick); \\
\qquad\qquad\qquad\qquad cockpit(\text{"}fly\text{"}, ()) \\
\qquad\qquad\qquad \textbf{else } cockpit(\text{"}fly\text{"}, ())
\end{array}
$$

Figure 11: The cockpit agent

$$
\begin{array}{l}
\textbf{let rec } \ \textit{pilot} = \\
\quad \textbf{let } \ \textit{alt} = \textit{loc}(0) \\
\quad \quad \ \ \textit{wp} = \textit{loc}(\textit{null}) \\
\quad \textbf{in } \ \Lambda_{\omega_1}(m, v). \\
\quad \quad \textbf{if } \ m = \text{``}\textit{checkalt}\text{''} \ \textbf{ then } \ \textit{alt} := (\textit{cockpit}(\text{``}\textit{alt}\text{''}, ())); \textit{task}(\omega_2) \ \textbf{ else} \\
\quad \quad \textbf{if } \ m = \text{``}\textit{monitoralt}\text{''} \ \textbf{ then } \ \textit{pilot}(\text{``}\textit{checkalt}\text{''}, ()); \\
\quad \quad \quad \quad \textbf{if } \ \neg \textit{inrange}(\textit{alt}) \\
\quad \quad \quad \quad \textbf{then } \ \textit{pilot}(\text{``}\textit{modifyalt}\text{''}, ()); \\
\quad \quad \quad \quad \quad \quad \textit{pilot}(\text{``}\textit{monitoralt}\text{''}, ()) \\
\quad \quad \quad \quad \textbf{else } \ \textit{pilot}(\text{``}\textit{monitoralt}\text{''}, ()) \ \textbf{ else} \\
\quad \quad \textbf{if } \ m = \text{``}\textit{modifyalt}\text{''} \ \textbf{ then if } \ \textit{low}(\textit{alt}) \\
\quad \quad \quad \quad \textbf{then } \ \textit{cockpit}(\text{``}\textit{updatejoystick}\text{''}, \text{``}\textit{up}\text{''}); \textit{task}(\omega_3) \\
\quad \quad \quad \quad \textbf{else } \ \textit{cockpit}(\text{``}\textit{updatejoystick}\text{''}, \text{``}\textit{down}\text{''}); \textit{task}(\omega_4) \ \textbf{ else} \\
\quad \quad \textbf{if } \ m = \text{``}\textit{broadcastposition}\text{''} \ \textbf{ then if } \ \textit{timeforiff}(\textit{wp}) \\
\quad \quad \quad \quad \textbf{then } \ \textit{cockpit}(\text{``}\textit{broadcastiff}\text{''}, ()); \textit{task}(\omega_5); \textit{pilot}(\text{``}\textit{iff}\text{''}) \\
\quad \quad \quad \quad \textbf{else } \ \textit{pilot}(\text{``}\textit{iff}\text{''}, ()) \ \textbf{ else} \\
\quad \quad \textbf{if } \ m = \text{``}\textit{checkwaypoint}\text{''} \ \textbf{ then if } \ \textit{at}(\textit{wp}) \\
\quad \quad \quad \quad \textbf{then } \ \textit{wp} := \textit{navigator}(\text{``}\textit{nextwaypoint}\text{''}, ()); \\
\quad \quad \quad \quad \quad \quad \textit{task}(\omega_6); \\
\quad \quad \quad \quad \quad \quad \textit{pilot}(\text{``}\textit{checkwaypoint}\text{''}, ()) \\
\quad \quad \quad \quad \textbf{else } \ \textit{pilot}(\text{``}\textit{checkwaypoint}\text{''}, ())
\end{array}
$$

Figure 12: The pilot agent

7 Example

8 Conclusion and further work

Some text to be included: we have identified the main components of task specifications and identified problems which can occur with the validity of the information which is passed between different agents, namely the problems arising from exceeding workload restrictions. We have shown how such task specifications can be represented using a simple functional notation which has been extended with agent expressions. The resulting task notation has been given a semantics, defining precisely how tasks are performed. This forms the basis for a Human Factors tool which can be used to prototype task specifications where the tasks are performed by interacting agents. Using knowledge from a Human Factors expert, the tool can analyse the task specifications in order to identify potential problems. Some text describing the types of Human Factors expert analysis which we envisage can be performed.