

# Goal-driven behavior in S-CLAIM

*Gerard Simons & Alex Garella*



# Table of Contents

|                                                                |    |
|----------------------------------------------------------------|----|
| Goal-driven behavior in S-CLAIM .....                          | 1  |
| Table of Contents .....                                        | 2  |
| 1. Introduction .....                                          | 3  |
| 2. State of the art in Multi-Agent System programming .....    | 4  |
| 3. Concept for goal-driven behavior in S-CLAIM.....            | 7  |
| 3.1 Goal life-cycles of different goal types.....              | 7  |
| 3.2 Goal delegation, cooperative and competitive behavior..... | 10 |
| 3.3 Goal Priorities .....                                      | 10 |
| 3.4 Representation of the mental state.....                    | 11 |
| 3.5 Inference of sub-goals and plans .....                     | 12 |
| 3.6 Finite State Machine Generic Goal.....                     | 13 |
| 3.7 Illustrative Example .....                                 | 15 |
| 3.8 Challenges to Plan Generation.....                         | 17 |
| 3.9 Plan Generation.....                                       | 20 |
| 3.10 Illustrative GraphPlan example.....                       | 23 |
| 3.11 Goal Representation in S-CLAIM .....                      | 25 |
| 4 Software design & Implementation .....                       | 27 |
| 4.1 Software Design .....                                      | 27 |
| 4.2 Implementation .....                                       | 28 |
| 4.3 Testing .....                                              | 34 |
| 5. Conclusion & Future Work .....                              | 35 |
| 6. Bibliography .....                                          | 36 |

# 1. Introduction

## Multi Agent Systems

Multi-agent systems (MAS) are used to describe several agents that interact with each other (positively, but also negatively) within an environment. These agents are able to solve problems difficult for centralized systems to solve by interacting with each other. These agents have cognitive and communication capabilities and the capability to achieve goals of various forms.

## CLAIM

The CLAIM[1] (Computational Language for Autonomous, Intelligent and Mobile Agents) project constitutes a framework and higher programming language in which agents can be modeled. CLAIM aims to reduce the gap in the concept and design stages by improving the tools to design agents, freeing designers from low-level details. Goals, capabilities, processes and messages can be created within this language. The framework also contains functionality for mobility. Agents are able to migrate from one computational site (e.g. a computer or mobile device) to another. CLAIM agents can be modeled as agents in a tree structure hierarchy. The CLAIM language contains primitives to modify the hierarchy by means of migration of the agents. The use of goals combined with the mobility of these agents sets CLAIM apart from other agent programming languages.

## S-CLAIM

S-CLAIM is an extension of CLAIM. S-CLAIM is now built on top of the JADE platform[2] instead of the SyMPA platform[3], the reasons for the switch to JADE are its compatibility with android devices, extensive documentation and it has been used worldwide for more than 10 years. Additionally, S-CLAIM should be even more agent-oriented to the point that it should be possible to be used by people not specialized in programming. The structure of S-CLAIM should be clearer and more understandable thanks to the more high-level approach compared to CLAIM. Goals should be defined in a declarative manner like in languages such as the GOAL programming language while also retaining and expanding on its mobile functionality. The use of declarative goals should make the agents more proficient in cooperative behavior. The idea of the reactive and proactive paradigm, where proactive behavior is an agent's desire to fulfill its goals and reactive behavior is its response upon messages from other agents or percepts from the environment should be explicitly built into S-CLAIM

## Introducing goal-driven behavior in S-Claim

Our aim will be mainly to create a proof of concept for goal-driven behavior by implementing goal declaration and functionality in the S-CLAIM language. Things to consider are the way goal-driven behavior is currently used in BDI (believe desire intentions) systems and how it could be implemented within S-CLAIM framework. How do the difference in concepts like belief, desires and intentions in BDI systems and knowledge, goals and capabilities affect implementation of goal-oriented behavior? How can an agent's goals be reached using its finite set of capabilities? As goals are high level structures, how should these structures be translated to the agent's basic capabilities? When and how should agents work together to achieve common goals? What kind of mechanism (e.g. Automata and Petri nets) can we design to control the status of the goals?

## 2. State of the art in Multi-Agent System programming

### Declarative goal types

A declarative goal is a state in the environment which the agent wants to reach. The use of declarative goals has been introduced to simplify the design and implementation of Multi-Agent Systems. The use of declarative goals has the following advantages:

1. Well suited for a natural description of Multi-Agent Systems
2. Preservation of a high level of abstraction throughout the development process
3. The concept of goals is similar to the way humans think and act.

These points simplify the transition from the design to the implementation phase. This simplification leads to clearer and thus less error-prone code.

To be able to use goals in a declarative way we must define a way to represent them. When representing goals we can distinguish between three types of goals: Achievement goals, Perform goals and Maintain goals. This particular set of goal types has been deduced from literature and implemented systems. [4][5]

### Achievement Goals

Achievement goals are used to specify a state in the world that an agent wants to achieve. When the agent adopts an achievement goal it adopts a plan to achieve the desired target. The desired target is specified in the *target condition* of the achievement goal. When the target has been reached, the achievement goal has succeeded and it will be dropped as a consequence. An achievement goal will only fail when there is no way for the goal to be achieved, in this case the achievement goal will be aborted and dropped. The *failure condition* indicates when the achievement goal will fail and thus abort.

An example of an achieve goal is *AchieveCleanup*, specifying that the agent has to cleanup at a certain spot which is stored in its knowledge, with target condition: *not( wasteAt( 20,16 ) )* and failure condition: *not( reachable( 20,16 ) )*, where *( 20,16 )* is an *( X, Y )* coordinate. In this case the agent wants to clean up at position 20,16. The *AchieveCleanup* goal will be achieved once the condition *not( wasteAt( 20,16 ) )* will become true. Meaning that there is no more waste at position 20,16. The failure condition states that the achieve goal will fail if the position of the waste is not reachable by the agent.

### Perform Goals

A perform goal specifies activities that have to be performed by the agent. The perform goal will succeed when a corresponding plan has been generated and is performed. If no plan is found or generated the perform goal will fail. Thus the perform goal's outcome is solely dependent on the fact if an activity has been performed. The perform goal contains a redo option which allows the perform goal to execute a plan iteratively.

An example of a perform goal is when an agent has the goal to patrol a certain area. The goal could be named *PatrolArea* and it would consist of moving through a plane in a certain manner under certain conditions. The agent could move up en down the plane between 22.00 in the evening and 6.00 in the morning.

## Maintain Goals

A maintain goal has the purpose of maintaining a certain state in the environment. The agent will monitor this desired state and will re-establish it if it is violated. The maintain goal has a *maintain condition* and an optional *target condition*. Once the maintain condition has been violated the agent will select a plan to re-establish the desired state. In some cases the plan has to be executed until a certain target has been reached which is specified in the target condition. This is the case when an agent has to maintain a charged battery. In this case we can specify a *maintain condition*: "charge state > 20%", and a target condition: 'charge state = 100%'. In this example the agent will start charging the battery when the charge state is lower than 21% and will keep charging until the charge state reaches the *target condition* of 100%.

## Plan Generation

Goals are defined to be able to create, or infer, a series of actions, called a plan. Generating a feasible plan in order to achieve the goals is paramount to creating an effective goal-driven agent.

Plan generation in BDI systems is the act of creating a sequence of actions (a plan) from an agent's given goals and beliefs. Plans are necessary for agents to reach their goals. Therefore a suitable plan generation in any MAS systems is of the greatest importance. Because of the real-time and variable characteristics of the environment the plans have to be kept up to date by means of a mechanism that ensures that the preconditions of the generated plans are consistent with the environment. If the preconditions do not hold any more part of the plan has to be recalculated.

## Backward chaining

Backward chaining will be used in S-CLAIM to achieve proactive behavior. it allows the agent to work from a goal in its goal queue back to a capability it can perform. This 'chain' back to a simple capability constitutes a sequence of actions (or capabilities) which becomes the agent's plan to achieve it's goal.

## Forward chaining

In addition to being proactive, an agent may exhibit reactive behavior by reacting to messages it receives from other agents. The agent can respond to a message it receives by ways of forward chaining, which creates a set of actions; a plan. From the actions described in the message the agent will chain forward to see if by doing so it can (partially) achieve its own goals, which may lead to its cooperation in performing the actions asked for by the other agent.

A combination of backward and forward chaining can be used to achieve proactive behavior.

## Task delegation

An important aspect is task delegation. First of all we must consider the attitudes of an agent towards other agents. We can distinguish between two types of attitudes: cooperative and competitive attitudes.

When agents have a cooperative attitude towards each other and have overlapping goals, the agent may decide to receive tasks from another agent, so that the agents work together to reach a common goal. Conversely, if agents have competitive attitudes towards each other, the agent has to decide for himself if it is in his best interest to cooperate with other agents to reach a goal.

The use of declarative goals should make task delegation easier by means of goal delegation. For example agent A has a certain goal: TurnOffLight. If however it is more economical for agent B to achieve this particular goal because he is closer to the light switch, then the goal should be delegated to agent B.

## **Agent Mobility**

The agents in CLAIM are structured in a hierarchy with a tree structure. Agents are ordered as such, an agent can have several agents as children. When an agent moves to another agent or device, the sub-agents of this agent move with it. The root of each tree is the computational device platform. Removing, adding and other primitives are used to change the structure of these hierarchies are based on the ambient calculus.

Using mobility effectively poses another important possibility for increasing the MAS' overall efficacy.

## **BDI vs. CLAIM**

BDI agents have mental states composed of Beliefs (the state of the environment as far as the agent may know), Desires (the states the agent wants to reach) and Intentions (a collection of actions the agent intends to do to fulfill its desires).

CLAIM agents consist of knowledge, goals and (reactive and proactive) capabilities. Knowledge in CLAIM is similar the beliefs in BDI agents and the goals similar to the desires. BDI does not usually introduce explicit goals. Introducing goals may overcome some "traditional" limitations of the pure-BDI approach. Examples of limitations of the pure BDI approach are its lack of explicit communication goals and explicit goals.[4]

BDI systems are also not explicitly mobile, introducing mobility can provide additional advantages to an multi-agent system by adding mobile computing capabilities which increase the efficiency of a multi-agent system.

The BDI approach is currently the most widely used and established approach to multi-agent system programming. Claim has been envisioned to tackle the limitations of the pure BDI approach by adding functionalities as mobility and explicit goals.

# 3. Concept for goal-driven behavior in S-CLAIM

Our aim is to implement goals and the goal-driven behavior in CLAIM by using the various goal types. By looking at related articles such as *Goal Types by van Riemsdijk[5]* and *Goal representation for BDI Agents by Braubach[4]* and comparing these to the functionality of CLAIM[3] we can create a proof of concept which describes how the goal-driven behavior should be implemented.

***The goal-life cycles and goal types are based on the papers by Braubach[4] and van Riemsdijk[5]. These notions have been extended by us by introducing goal priorities and a hierarchical graph structure to represent the goal base.***

## 3.1 Goal life-cycles of different goal types

### Generic goal type

A goal can be seen as being in three distinct states. These states are “New”, “Adopted” and “Finished”.

When a goal is created - goals can be either created by the agent programmer, or come from requestGoal messages - the goal is in the “New” stage in which it stays until an agent chooses to adopt the goal upon which the goal’s state changes to “Adopted”. When an agent adopts a goal it can always choose to drop the goal. When adopted, the goal is added to goal structure, where it is stored - possibly with other goals.

Within “Adopted” several sub-states exist, these can be defined as “Option”, “Active” and “Suspended”. When the goal enters the “Adopted” state, it enters the “Option” state. This state indicates that the goal is an option for the Agent to pursue when the circumstances allow it.

It is only when an agent does this that the goal changes it’s state to “Active”. When in this state, the goal will be actively pursued by the agent who adopted this goal.

When a goal’s context becomes invalid (e.g. the goal is not achievable) the state changes from either “Active” or “Option”, to “Suspended”.

A goal may return from the option state after begin suspended, to return it to the “Option” state, after which it may once again be deliberated for further processing.

Whenever a goal meets its drop condition (e.g. the goal has been achieved) the state changes to finished and the life-cycle ends.

Figure 1 from *GOAL representation for BDI agents by Braubach[4]* illustrates the life cycle of a goal as we envisioned it for CLAIM2.

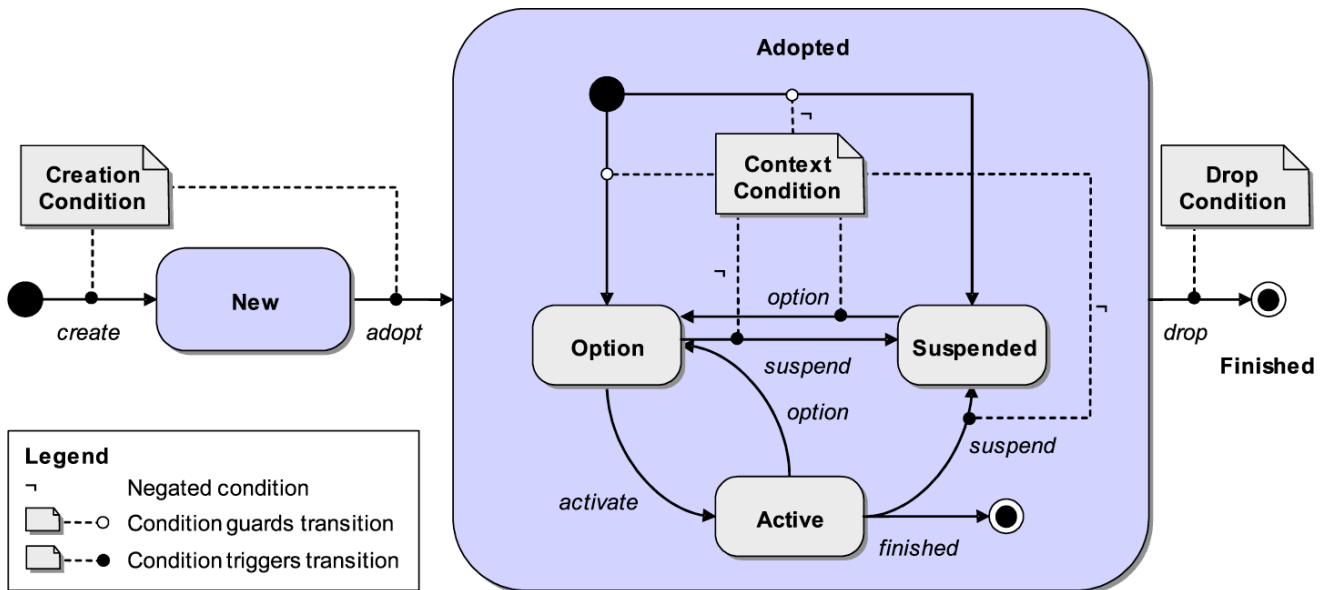


Figure 1: Generic goal life-cycle

## Achieve goal

The notation  $\text{AchieveGoal}(t, P, f)$  is read as “target condition  $t$  using the set of plans  $P$ ; failing if  $f$  becomes true”. An achieve goal succeeds if and only if the target condition  $t$ , is met and fails iff the fail condition  $f$  is met. The conditions  $t$  and  $f$  are defined declaratively as states in the environment. Since the environment is bound to change, either of these states could become true while the agent is executing plans to pursue its goals. An achieve goal is only dropped when it either succeeds or fails, independent from its plans. The goal will be pursued indefinitely until it either fails or succeeds. Figure 2 illustrates the life-cycle of an achieve goal.

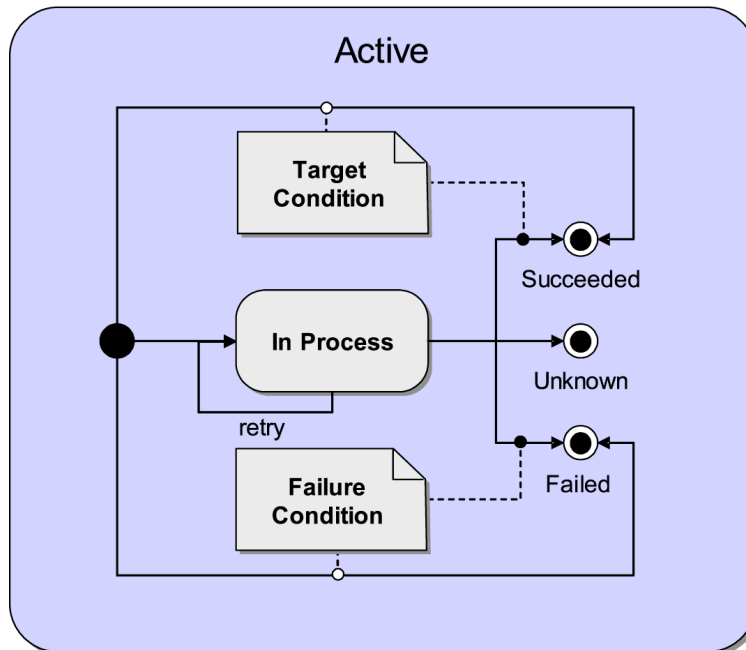


Figure 2: Achieve goal life-cycle

## Perform goal

The outcome of a perform goal, unlike the outcome of an achieve goal depends on the execution of a plan. A perform goal succeeds when a plan corresponding to a goal is executed and fails when no corresponding plan is found to execute. The perform goal has a redo option to perform the goal iteratively. The goal is dropped either if the goal succeeds or fails. The life-cycle of a perform goal is shown in figure 3.

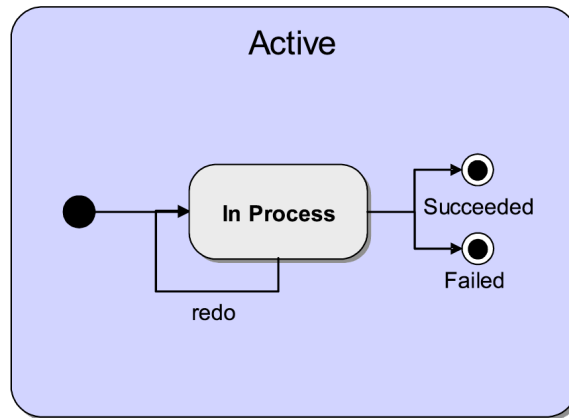


Figure 3: Perform goal life-cycle

## Maintain Goal

The purpose of a maintain goal is to maintain a certain state in the environment. Once the maintain goal is adopted the agent will observe the state in the environment as specified in the *maintain condition*. Once the *maintain condition* is violated the agent will come in action to re-establish the desired state in the environment. In some cases the plan has to be executed until a certain target has been reached which can be specified in the optional *target condition*. This is the case when an agent has to maintain a charged battery. In this case we can specify a *maintain condition*: 'charge state > 20%', and a target condition: 'charge state = 100%'. In this example the agent will start charging the battery when the charge state is lower than 21% and will keep charging until the charge state reaches the *target condition* of 100%. The maintain goal lifecycle is shown in figure 4.

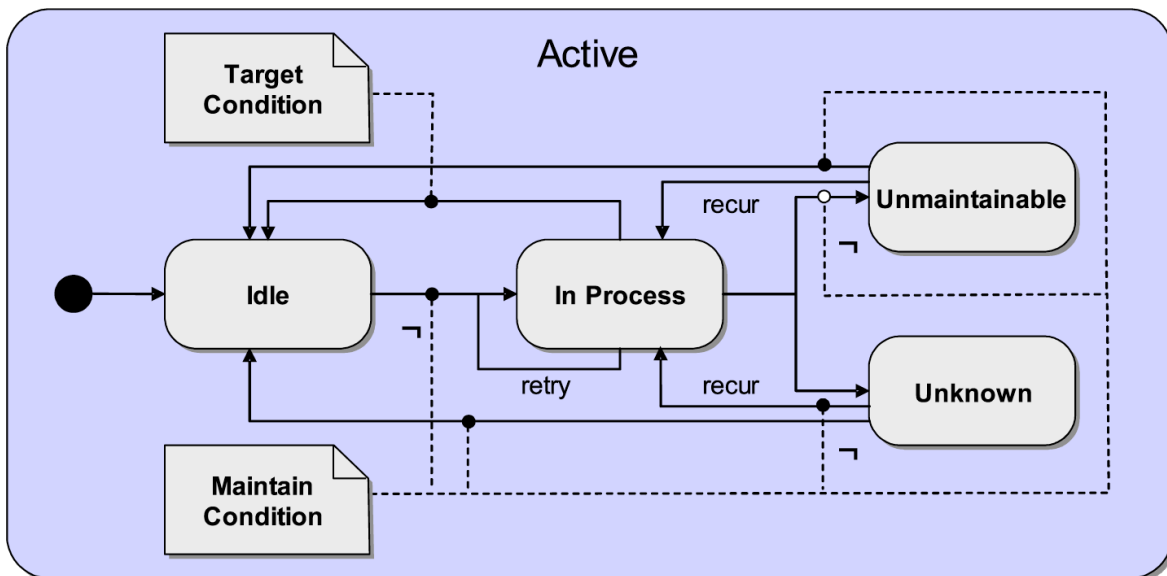


Figure 4: Maintain goal life-cycle

## 3.2 Goal delegation, cooperative and competitive behavior

Depending on the attitude towards other agents, which can be either cooperative or competitive, an agent can decide to delegate tasks to other agents. Task delegation can be simplified with the use of declarative goals because agents can reason about their goals and can simply request other agents to adopt a certain goal or sub-goal whenever beneficial.

Suppose *agent A* and *agent B* have a *cooperative attitude* towards each other and *A* asks *B* to adopt the goal TurnOffLights. Whenever it is more economical for *B* than for *A* to achieve this particular goal then *B* should adopt the goal TurnOffLights. For example, *B* should adopt the goal if it is closer to a light switch than *A* in which case the global performance of the MAS would improve.

When two agents have a *competitive attitude* towards each other, an agent might still try to delegate a goal to another agent. In this case the receiving agent has to decide for himself it is beneficial for it to adopt the goal, not taking in account the global performance of the MAS but uniquely its own performance.

From this analysis of goal delegation we can conclude that when agents are in a *cooperative* setting they should accept goals if it is beneficial for the MAS as a whole and that agents in a *competitive* setting should only accept goals whenever it is beneficial for themselves.

## 3.3 Goal Priorities

The agent should have a mechanism to decide which goals should be performed first. An easy and intuitive way to build such a mechanism is by assigning a priority level to each top-level goal. The agent will simply choose to perform the goal with the highest priority assigned to it. Assigning priorities to the goals is an easy and intuitive way for a programmer to define in what order top-goals should be performed. In more complex applications, there should be a mechanism that assigns priorities to the goals automatically depending on the environment. This is a point which should be treated in future work as it is not directly related to our research.

When an agent can choose between multiple sub-goals to reach a top-level goal the priorities can be regarded as weights. When a sub-goal fulfills multiple top-level goals, the weights of the top-level goals should be taken into consideration. The sub-goal that fulfills the top-level goals with the highest weight should be chosen.

Sub-goals are also assigned a weight by means of a cost-function which can be defined by the programmer. The optimal path is a sequence of goals and sub-goals in the graph that can be computed by subtracting the cost of the sub-goals from the top-level goals. The path that yields the highest weight after subtracting the costs is the optimal one.

### 3.4 Representation of the mental state

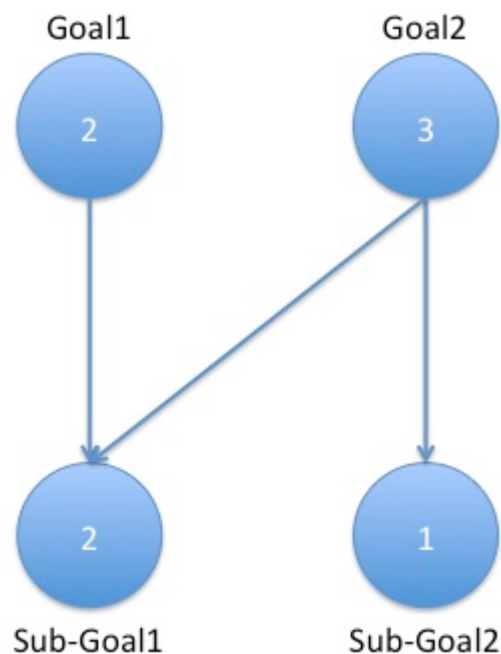
For our purpose we can define the mental state of the agent as composed of three parts:

1. The knowledge base,
2. the goal base,
3. and the intention base.

The knowledge base contains information from the environment, the goal base contains goals that have been adopted by the agent and the intention base contains the goals and plans that are currently being pursued by the agent.

The idea is to match the knowledge and goal base to be able to select which goals and plans should be added to the intention base for further processing.

The goal base is composed of instantiated goals and sub-goals which can be represented in a graph structure composed of interconnected tree structure with the high level goals as roots of each tree structure. Figure 5 is a graphical representation of the goal base.



**Figure 5: Graphical representation of the goal base**

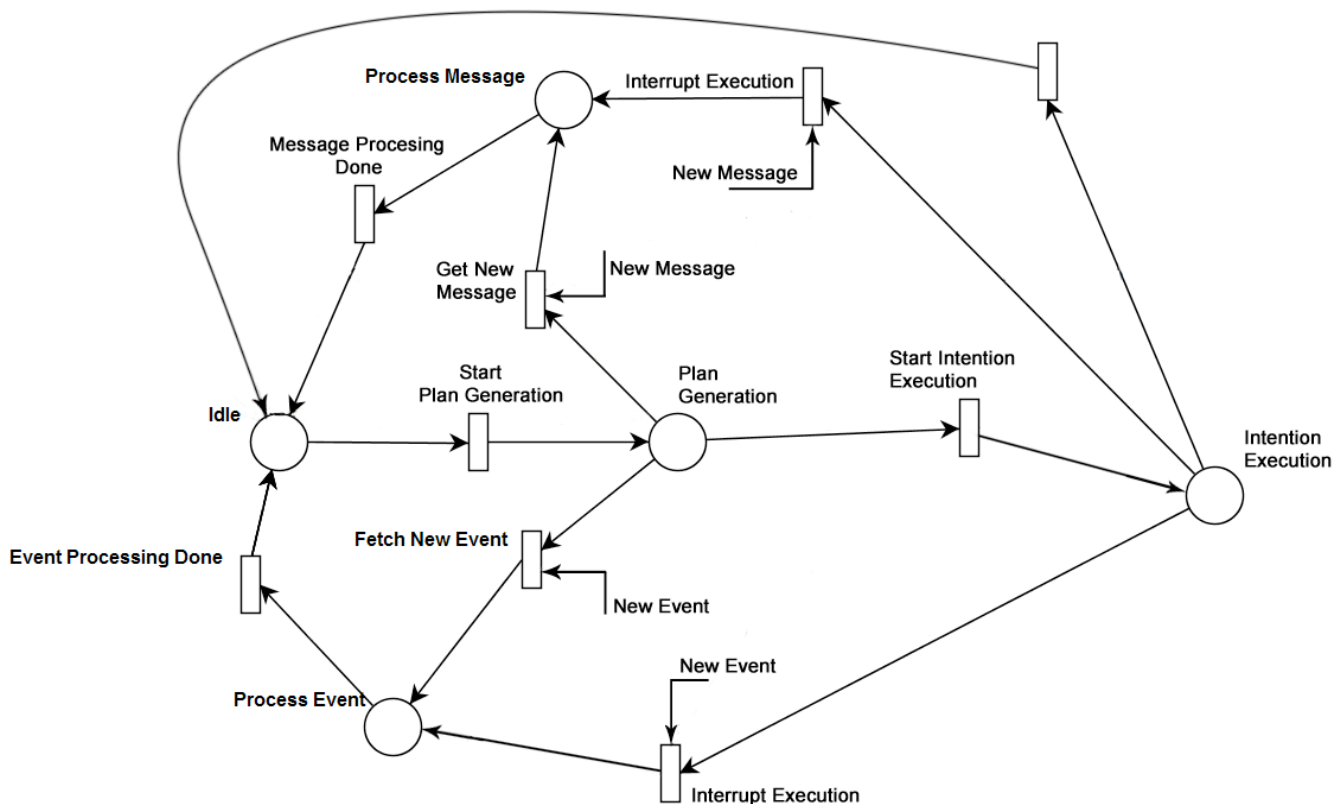
Suppose the goal base consists of two top-level goals: *Goal1* and *Goal2*. The integer in the nodes of the top-level goals represents the priority level and the integer in the nodes of the sub-goals represents the cost to perform the goal, which is defined by an arbitrary cost function. Because of the higher priority level of *Goal2*, the agent will choose to pursue *Goal2* first. Now that the agent has decided to pursue *Goal2* it must choose between *Sub-Goal1* and *Sub-Goal2*. Choosing *Sub-Goal2* would lead to a weight of  $3 - 1 = 2$  and choosing *Sub-Goal1* would lead to  $3 - 2 + 2 = 3$  because *Sub-Goal1* fulfills *Goal1* and *Goal2*. Therefore the agent should choose *Sub-Goal1* over *Sub-Goal2*, because this decision would lead to a higher overall utility.

### 3.5 Inference of sub-goals and plans

An agent can be in different states. Grossly defined these states are **Idle**, **Goal Deliberation**, **Intent Execution**, **Message Processing**, **Event Processing**

Another important part is the **Updating of the goal base**, which creates the tree structure from the goals necessary to assess the **Intent**. This mechanism can be seen more as a subroutine than an actual state, so it is not part of the Petri net, but is executed nonetheless after each event or message.

The global execution of an agent can be viewed from this Petri net is shown in figure 6.



**Figure 6: Global execution mechanism of an agent**

Descriptions of the states can be seen below as also a more detailed description of the functionality inside each state. Whenever an agent is idle, it can start generating a plan, when a new message or event is received, the agent postpones this execution and proceeds with updating (if necessary) the goal base and/or environment. Execution is then possibly re-executed.

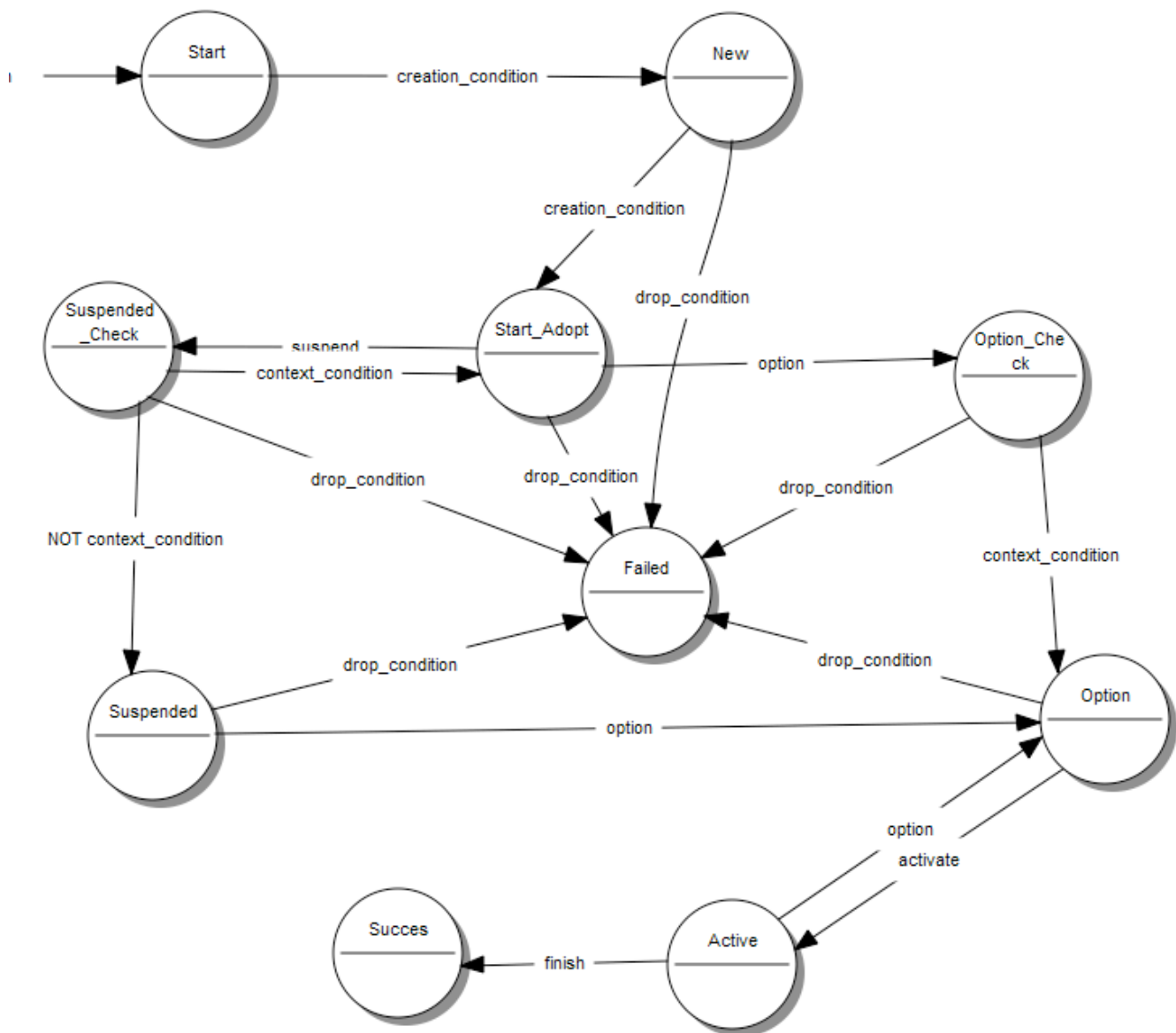
**Idle**

An agent is idle, if it has nothing to do, an idle agent will however always be listening for incoming messages and events.

**3.6 Finite State Machine Generic Goal**

Formalizing Braubach to useful UML-like state diagrams will allow us to achieve a better understanding of the goal life cycle, and gives us the capability to formally check for integrity and correctness.

The diagram of the state machine is given in figure 7 and is further explained underneath.



**Figure 7: Finite state machine describing the generic goal execution**

Each goal is created when it's creation condition is met. It can then be adopted by an agent by going to the state **Start\_Adopt**. It can then also become an option, when this criterion is met. According to the context condition it will continue to become an option and eventually active, where it may be achieved. Furthermore a goal can always be suspended. See the goal life cycles for more information on each state.

### 3.7 Illustrative Example

Our scenario is based on the blocks world, which is a scenario most commonly used to show the effectiveness and validity of an agent system; it is a world which consists of a table and several blocks. The blocks can be moved according to a simple rule: A block can only be moved when there is no block on top of it and the target block, the block it wishes to be put on. It focuses on a single agent and its plan generation based on creating a tree from the goal base and finding the most efficient path to reach the highest value of goals in the most efficient way possible. We provide different granularities for the algorithm, where a rough granularity of goals will be less exact but is more rapidly deployed than more precise ones.

We denote the knowledge of the initial state for this scenario as follows

```
{ on(A,Table) on(B,A) on(D,C), on(C,Table), clear(Table) };
```

The predicate on(X,Y) means block X is placed on top of block Y. And clear(X) means that there is no block on top of block X. The table is always so clear so that blocks can be moved to the table.

The capabilities are denote as:

Move X from Y to Z:

```
move(X,Y,Z) ::=
  pre: {clear(X),clear(Z)}
  post: {on(X,Z), clear(X), Clear(Y), ¬clear(Z)}
```

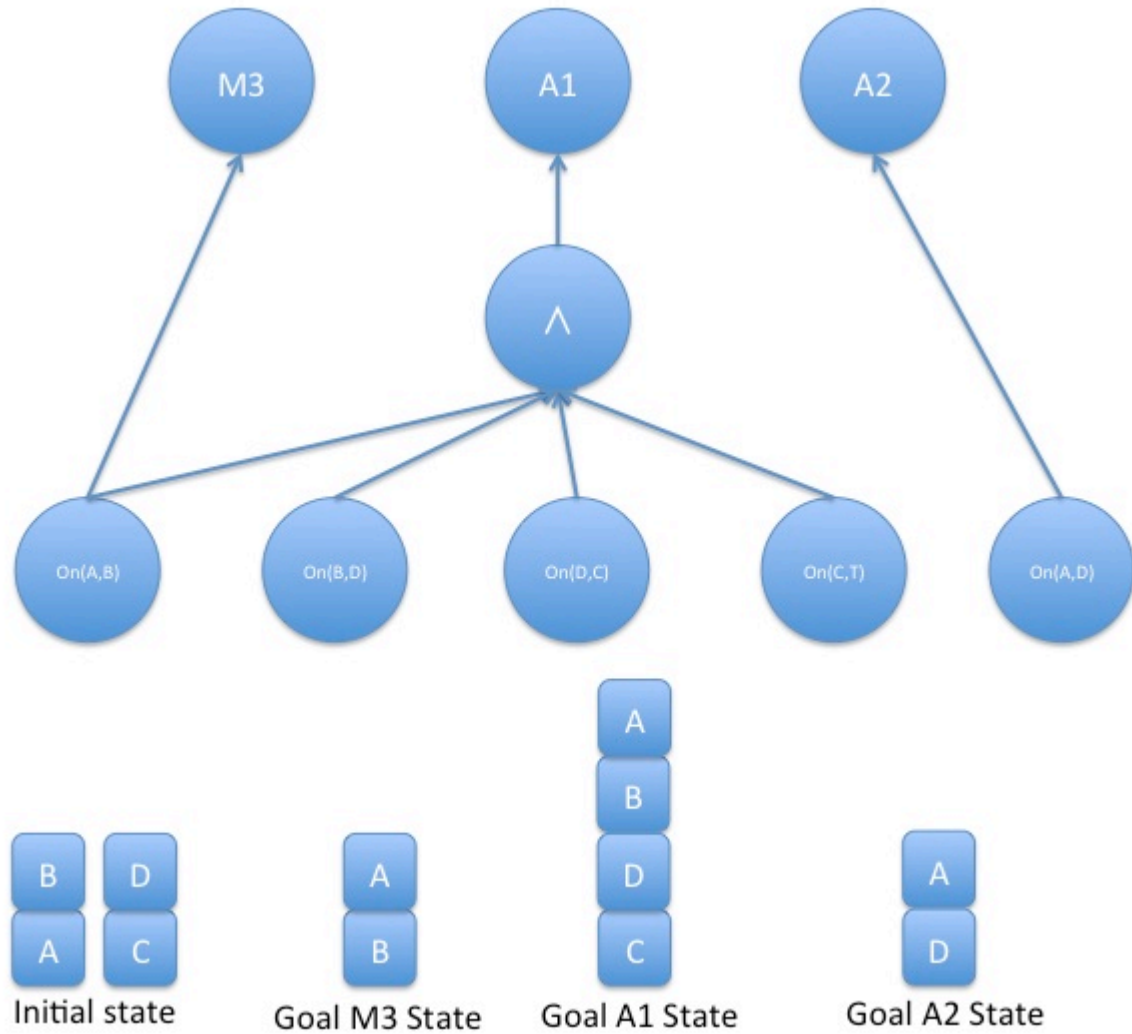
Goals are

```
A1::=
  type: Achievement
  requirements: {on(C,Table), on(D,C), on(B,D), on(A,B)}

A2::=
  type: Achievement
  requirements: {on(A,D)}

M3::=
  type: Maintain
  maintain-condition: {on(A,B)}
  target-condition: {on(A,B)}
```

A graphical representation of our scenario can be seen in figure 8.



**Figure 8: Graphical representation of the goal base & environment states**

In the graphical representation we can see the three top-level goals: M3, A1 and A2 as defined previously. A1 is the conjunction of the on(C,Table), on(D,C), on(B,D), on(A,B) predicates, A2 consists of only ON(A,D) and M3 consists of only the ON(A,B) predicate. Underneath the graph consisting of the predicates we can see what these goals look like in the environment. To be able to go from the initial state to one of the goal states we need plan generation which generates the appropriate actions to achieve these goals.

### 3.8 Challenges to Plan Generation

Additional challenges exist in generating an acceptable plan for achieving goals. There are some specific cases which are worth expanding on. We try to illustrate the difficulty of each of these using a petri net. In each petri net the places represent goals, and transitions are the effects of each action. The core of the problem is that actions needed for goals may have conflicting results, completely or partially negating the other goal.

#### Competing Needs

When a goal is to be achieved, but the actions to achieve the sub-goals conflict, an action of one sub-goal makes the other invalid, a way to resolve the conflict must be found. One action preceding another may resolve this conflict. The Petri net in figure 9 shows this issue. As you can see, when trying to achieve the top-level goal P4, both P2 and P3 need to be achieved. To do this actions T0 and T1 need to be executed. Executing T1 before T0 however results in a deadlock, where P2 can no longer be achieved as there is no longer a token at P0 in order to execute T0. Executing T0 first and then executing T3 will allow tokens to be available to T2, making P4 attainable once more.

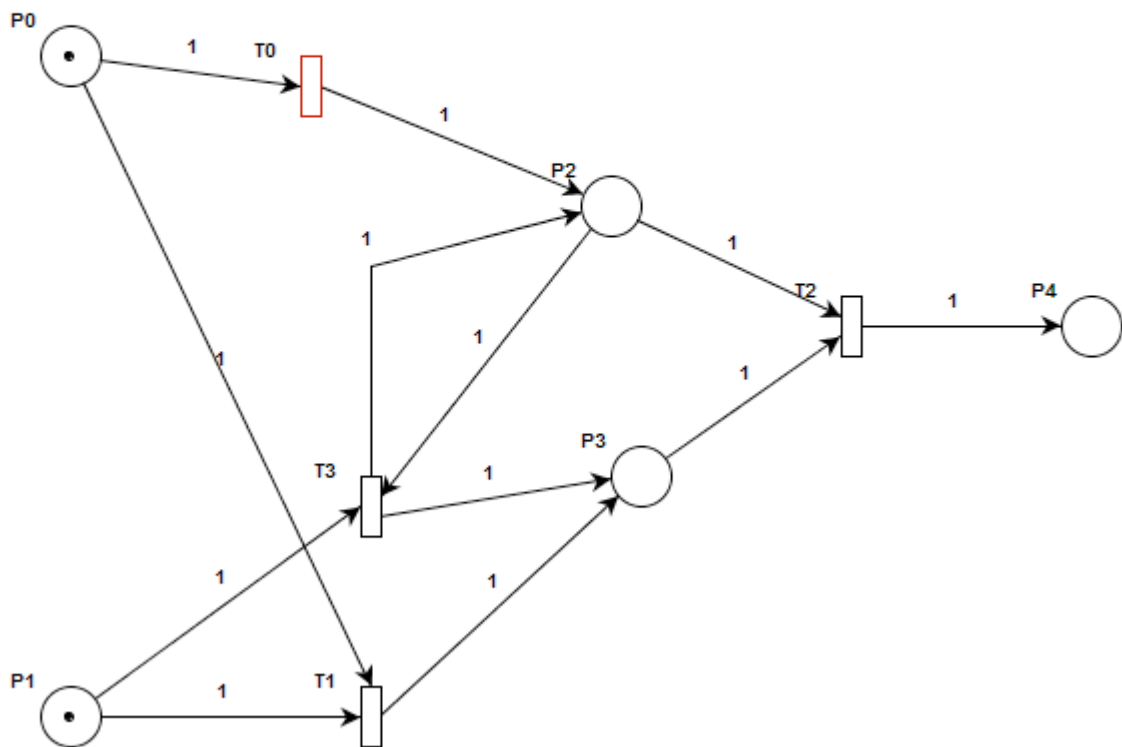
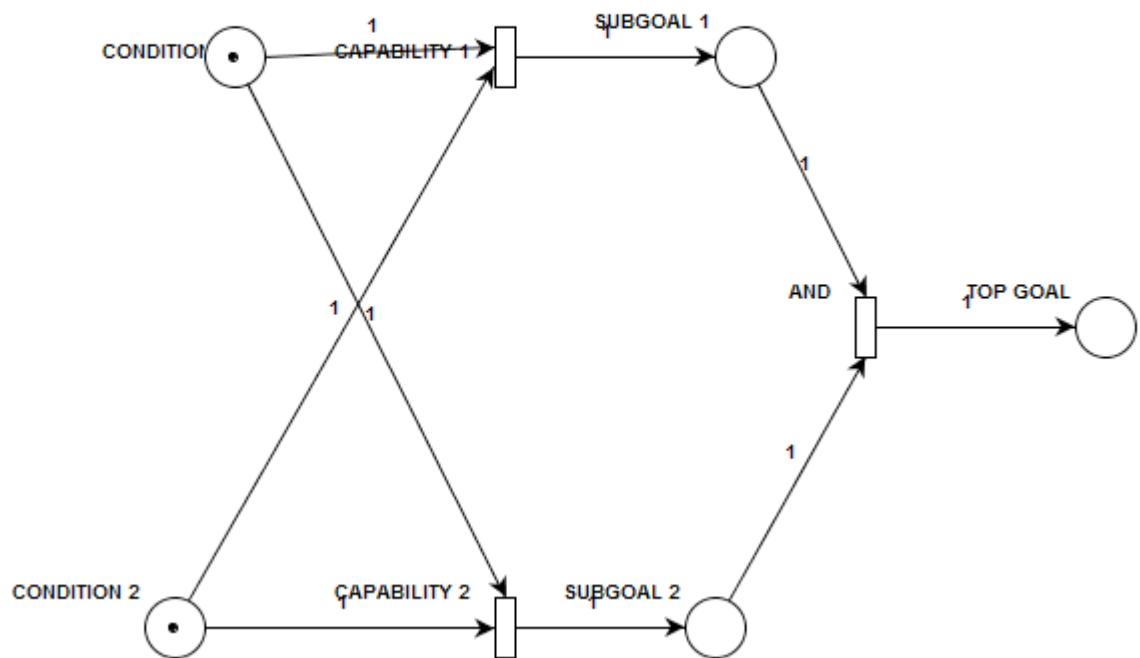


Figure 9: Petri net representation of competing needs

Here the places represent goals, and the transitions means to achieve them (actions) P4 represents the (top-level) goal we wish to achieve. By executing T1 before T0, the sub-goals and thus the top-goal can still be achieved.

## Interference

When two goals have conflicting goals, where either fulfillment excludes the other, there is an unresolvable conflict. This can be seen from the petri net in figure 10.

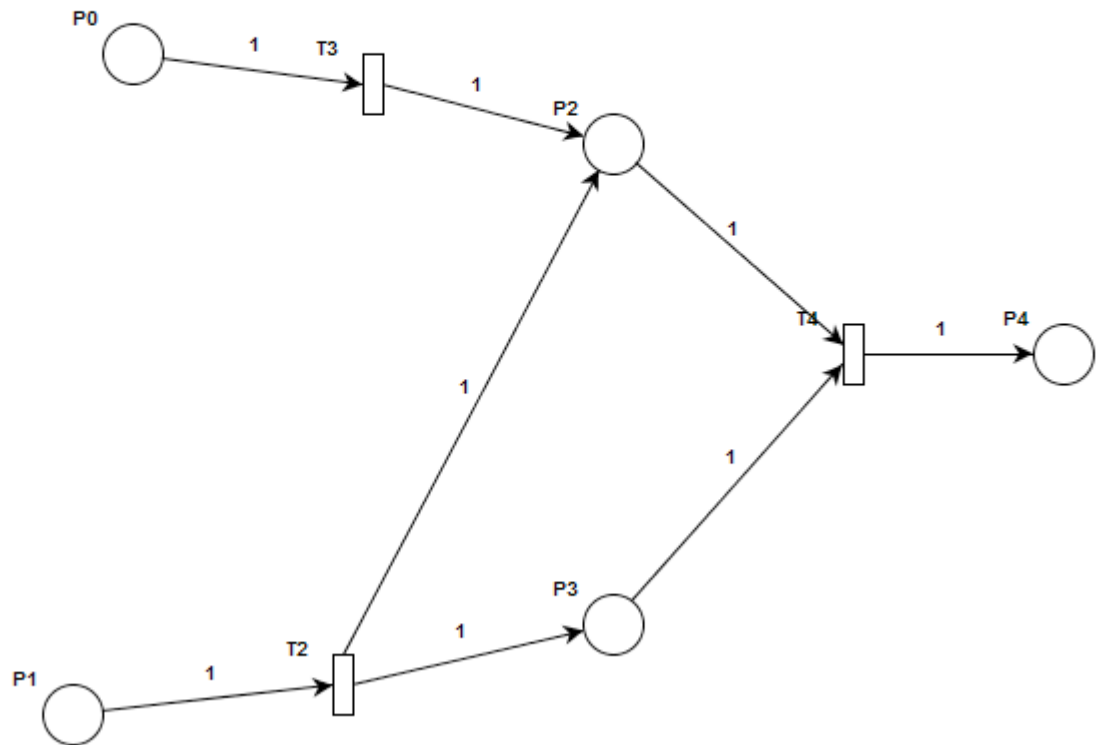


**Figure 10: Petri net representation of interference**

Executing capability 1 will make execution of 2 impossible and vice versa. So only 1 sub-goal can be achieved, and thus the top-goal will never be reached. Recognizing these difficulties is very important for an agent to be effective in its execution. How to deal with these situations will be explained later on.

## Subsumption

We defined the subsumption of one capability's execution realizes a goal, which was first thought to have to be validated through another capability. This example can be seen in figure 11.



**Figure 11: Petri net representation of subsumption**

Here execution of T2 realizes P2 as well, making T3 obsolete.

### 3.9 Plan Generation

Plan Generation is the key aspect to introducing goal-driven behavior in S-CLAIM. It is needed to convert the initial state of the environment to the goal state as defined by the programmer by means of declarative goals. Plan generation should take the initial state and the goal states and generate a plan which consists of actions that have to be executed in the environment to achieve the goal states. We use an approach of defining a *GraphPlan*[8] graph, processing mutual exclusion relations (mutex), converting it to a SAT instance and solving these with an arbitrary SAT solver. A plan can then be extracted from the SAT solution. This method has been described by Kautz et al. in SatPlan: Planning as satisfiability.[6][7]

#### Graph creation

We create our graph according to the GraphPlan[8] methodology. We intend to use the graph creation and mutex computation from it, but not the actual plan creation, as the SAT conversion is faster and more modular (an arbitrary SAT solver can be used).

Graphs used in GraphPlan are directed, leveled graphs. Leveled meaning there are several distinct sets, such that edges only connect between two adjacent levels.

The nodes of these graphs consists of two kinds. There are proposition nodes and action nodes. The proposition nodes are presented as states of elements of the environment, are either goals, post- or preconditions. Action nodes are the actions of an agent with directed edges to pre- and post-conditions.

Edges represent relations between actions and propositions of two adjacent levels. actions in level  $i$  are connected by precondition-edges to their preconditions propositions of level  $i - 1$ . The post-conditions in level  $i + 1$  are connected to actions by use of add=edges and delete=edges. A no-op relation defines a “no operation” semantic, where a proposition is already true it is extended along with other propositions to the next action level using this kind of edge.

The levels of nodes alternate between being actions and propositions. Each proposition which is not true in the current agent’s environment has an action added to it with a directed edge between the action and the proposition node. Other post-conditions an action may have are also added to the graph. Each post-condition may be of two kinds, it may be an add-effect or a delete-effect corresponding to a proposition that is added to the environment and one that deletes a proposition from an environment.

### **Creation of the graph:**

first layer = initial state

current\_layer = first\_layer

while level < max\_level

    actions\_layer = actions with preconditions in current\_layer

    current\_layer.next = action\_layer

    action\_layer.next = post conditions of actions in action\_layer

    current\_layer = action\_layer.next

    level++

    break if

        all goals are reachable

        graph levels off

endwhile

plan = actions to achieve these goals.

### **Finding the goals and plans**

The goals and plans are found by searching the graph. The goals are said to be found if they are contained in the last layer of propositions. The layers are added as long for as long as not all goals are in the final layer (or is aborted if none can be found).

Note that any goal that is any other layer than the last layer is also in the last layer, as a NOOP action is simply used in the intermediate layers to propagate the proposition's existence unto the intermediate layers and the last layer.

When the goals are all in the last layer, the plan is found as follows:

Each action level is searched for actions supporting the goals and is added to the plan list. This returns the action sequence or plan after having traversed all the action layers.

### **Limitations**

The current algorithm has some limitations, which constrains its usability for S-CLAIM severely. First and foremost, a plan is only generated when all the goals are met. For the algorithm to be of use to us, we would like it to return a set of sets of goals which can be achieved. These goal sets would be mutually achievable. The plan to reach each of these goal sets would also be needed to be returned.

Furthermore the algorithm does not account for priorities of goals and the costs involved of invoking actions.

To control the amount computations of the GraphPlan instance, a constraint should be placed on the graph creation. This constraint is already available in the current GraphPlan implementation, but it simply fails

when not finding the desired goals. Improvements should be made to at least return the goals found and their respective plan back to the agent.

## **Additions**

To resolve these matters with the current graphplan algorithm we propose to alter it in a fashion that will suit the needs of S-CLAIM.

First of all it is necessary to clearly distinguish the agent from the graphplan instance. We would like the agent to be clearly distinct from the graphplan. They would then only communicate by a clearly defined protocol.

The protocol can be described like this:

The agent sends to the GraphPlan instance it's goals (or a subset of it) to the graphplan.

The graphplan computes a set of sets of compatible goals with their appropriate planning back to the agent.

The agent then considers which of these goals sets are best to achieve given the costs of the capabilities and the priorities of each goal.

Note that the graphplan has no idea of costs and priorities involved with the goals and capabilities. Even though this knowledge could improve overall calculation speed this is done to keep (data)dependencies low.

It also ensures, the graphplan can easily be extended by any sort of SAT converter and solver like we discussed before, as these do not (yet) work with costs and priorities.

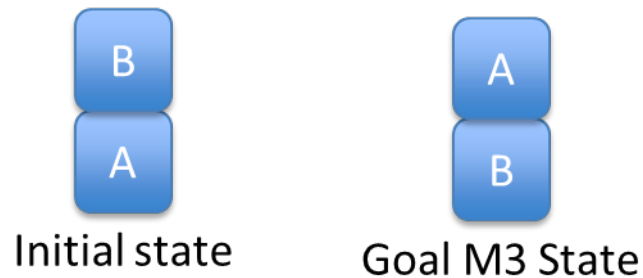
Another important note is that the amount of goals sent to the GraphPlan instance needs to be constrained in order to keep it computationally tractable. The amount of goal sets which can be derived from  $n$  goals is  $n!$ . Which is in EXPSPACE. By adding constraints of a maximum number of goals, the overall efficiency of the plan creation can still be held at an acceptable level. One could consider that the agent only sends the  $m$  highest ranking goals (in terms of priority) are send to the graphplan instance.

Other mechanisms can be applied as well. Memoization of certain goals and their costs/benefits can be used as a heuristics for determining whether it should be included in the set of goals sent to the graphplan for plan creation.

### 3.10 Illustrative GraphPlan example

In this example we will illustrate how graphplan works by means of an example by elaborating on the illustrative example used in 3.7.

Suppose we have defined the goals as seen in figure 12.



**Figure 12: An initial state and a maintain goal state**

Suppose we want to generate a plan from the initial state to achieve the A2 goal state. The initial proposition layer will consist of the initial state of the environment:

`ON(A, Table), ON(B, A), Clear(B).`

The following proposition always holds:

`Clear(Table).`

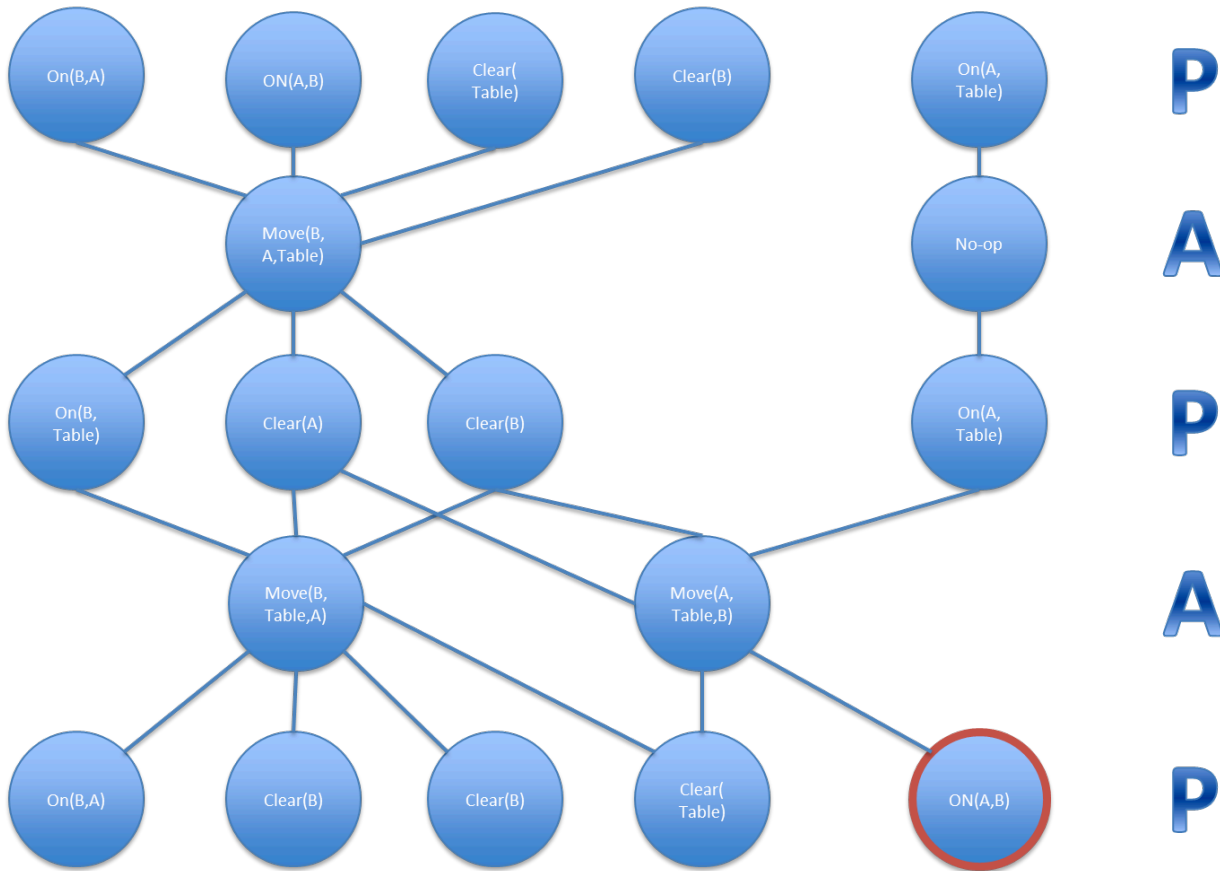
The goal state we want to achieve consists of the following propositions:

`ON(A,B).`

The possible action is:

`MOVE(X,Y,Z).`

Which stands for move block X from block Y to block Z where X, Y and Z are different blocks and either Y or Z may be the table.



**Figure 13: Resulting graph from GraphPlan execution**

Figure 13 shows how the graph would be created by a GraphPlan algorithm, with some adjustments for readability; The principal however, still remains valid, only superfluous nodes and edges were removed. The P's and A's at the side indicate a proposition layer and action layer respectively. The node with the red contour is indicated as the goal.

Creating the graph above happens as follows. The top layer, a proposition layer, is initiated as the given initial environment. Actions are generated for the given objects (in this case the blocks). Whenever an action's preconditions are met by the proposition layer, it is added to the next layer. The proposition layer is created from the action layers. Post-conditions of the action layer become the next proposition layer. This cycle is continued until the goal condition is found in one of the proposition layer (or alternatively none is found or the maximum amount of created levels is reached). Here ON(A,B) is reached in the fifth layer so graph creation halts. When graph creation is successful, collecting the appropriate actions is necessary for the graphplan to be able to return a viable plan. This is done by backtracking back from the goal to the initial layer. The action required to fulfill ON(A,B) is added to the collection (here MOVE(A,Table,B). The actions that make the preconditions CLEAR(A), CLEAR(B), On(A,Table) true are then added to the collection until the initial environment is reached. The collection is then returned as the action sequence, or plan, which will lead to the desired environment state.

### 3.11 Goal Representation in S-CLAIM

In order to introduce goal-driven behavior in the S-CLAIM agent programming language an appropriate goal representation and syntax have to be defined. The aims are to choose a goal representation so that the process of implementation is simplified for the programmer and to make the goal-driven behavior compatible with other aspects of a multi-agent system and the planning algorithm.

#### Overview

The goal representation must account for the different goal types. We can distinguish between the three following types of goals: achievement-, maintain- and perform goals. Because of the declarative nature of goals they can be represented as a k-tuple of k propositions describing a desired state in the environment which has to be reached or maintained by the goal. In case of the performance goal, the goal would consist of a plan that should be triggered when adopting the perform goal.

The goal representation has to be defined from two different perspectives:

1. the programmer's perspective and
2. the agent perspective.

To define the goal representation from the programmer's point of view we have to think about a suitable syntax which is compatible with the existing S-Claim syntax and its semantics. The agent's capabilities have to be defined so that it can pursue its goal driven behavior and perform actions in the environment. The goals have to be defined as a conjunction of valid states in the environment in the form of a conjunction so that the GraphPlan algorithm can generate a plan to pursue the goals.

When the goals and the actions of the agent have been defined and programmed by the environment, they have to be parsed so that the agent can store the goals in its knowledge base so that it can start reasoning about them and start pursuing them proactively. The goals such be stored in the knowledge base in such a way that the agent can efficiently use the plan generation algorithm to determine its course of actions to act proactively.

In both cases we need to keep in mind that a goal has a priority assigned to it.

#### Syntax

We have argued that we need a goal representation from the programmer's perspective for the programmer to be able to introduce goal-driven behavior in an agent. We therefore have to define a syntax which is compatible with the existing S-Claim syntax so that it can be used efficiently by the programmer.

Since we have three different goal types. We must introduce a specific notation for each goal type.

The maintain goal consists of two states: the maintain condition and the target condition.

This means that the **maintain goal** takes a name parameter, two state parameters and a priority parameter, which can be defined as follows *m-goal(String goal\_name, Integer priority, Conjunction maintain\_state, Conjunction target\_state)*, where a conjunction is a sequence of states in the environment joined by conjunctions. Where the *goal\_name* parameter defines the name of the goal, the *maintain\_state* parameter describes the state that has to be maintained by the agent, the *target\_state* which describes the state that has to be reached while pursuing plans to reinstate the *maintain\_state* and the *priority* which describes the priority of the goal.

The **achieve goal** takes a name parameter, two state parameters and a priority parameter which can be defined as follows: *a-goal(String goal\_name, Integer Priority, Conjunction target\_condition, Conjunction fail\_condition)*. This representation is similar to that of the maintain goal, the only difference is the *fail\_condition* which describes the state of the environment in which the goal should be dropped.

The **perform goal** is a particular kind of goal because it does not describe a state in the environment. The only thing a perform goal needs to be able to do is to trigger a plan. We can define a perform goal as follows: *p-goal(String goal\_name, Action action)*. Where 'Action action' stands for an action to be performed in the environment.

## Goal base

The other aspect of the goal-representation that has to be defined is the aspect from within the agent. Once the goals have been parsed they will be added to the knowledge base of the agent. To make a distinction between goals and other knowledge we will define a *goal base* which is equivalent to the knowledge base in essence. The only difference is that the goal base consists of goals exclusively. The separation of goal and knowledge base makes it easier for the agent to reason about its goals and to choose what kind of computations it may do on its goals. The goal base will consist of a list of adopted goals which may be in the option, active or suspended state.

It should be possible for the agent and the programmer to query the goal base by using the goal name and / or goal type to request certain information about the goal, for example its state. The agent will make use of the goal base to request plans from the plan generator.

# 4 Software design & Implementation

## 4.1 Software Design

In this part of the report we will explain the design of the implementation part of the project by giving an overview of the packages and classes.

The five following packages have been used for the implementation as shown in figure 14:

1. The Claim package,
2. the JADE package,
3. the Parser package,
4. the Planning package and
5. the GraphPlan package.

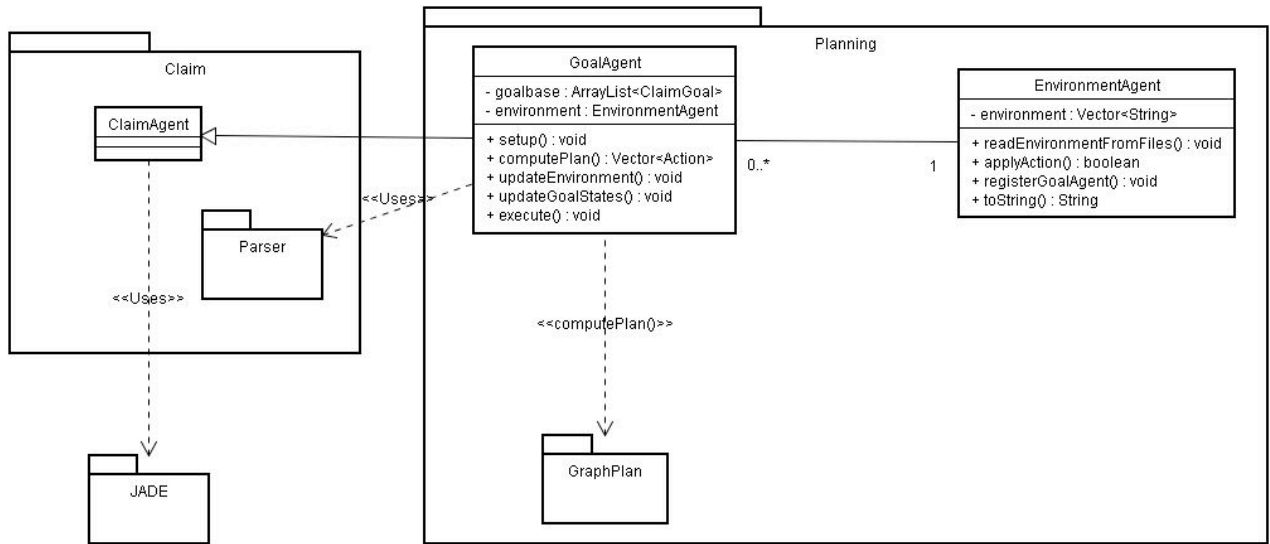
The **Claim** package contains all the core classes used by CLAIM such as the CLAIMAgent and the Parser package.

The **JADE** package contains all classes involved in the JADE platform which is used by all CLAIMAgents in order for it to have all functionality (such as migration, communication etc.)

The **Parser** package contains all classes necessary for parsing, lexical analysis and processing of files. It is used to create agents from user-defined text. We have modified this part so that it can now parse goals.

The **Planning** package contains the implementation that has been added by us from scratch. This means that it contains the goal-driven behavior functionality of the agent, including the environment and GraphPlan.

The **GraphPlan** package contains the computation part of the planning. This package contains all the functionality to convert a goal to an action sequence of applicable actions.



**Figure 14: Overview of the relevant packages and classes**

The previous image shows an overview of the implementation by depicting the most important packages and classes with their relevant attributes and methods. The planning package contains the functionality that we have implemented to obtain goal-driven behavior. The parser package has been modified to be able to parse the user-defined goals.

## 4.2 Implementation

By implementing our concept, we have constructed a proof of concept which demonstrates goal-driven behavior in the Claim platform. The details of the implementation will be explained in this section. We have extended the existing parser and lexical analyzer of the existing S-Claim language to be able to support declarative goals. We have created java classes to implement the goals, the goal agent and the blocks environment. The planning algorithm is a modified version of the GraphPlan algorithm.

### Parser

The parser and lexical analyzer that has been used in the CLAIM project is BYACC/JJ[9]. The reason that this parser has been chosen is because it is a well-known parser with a lot of documentation and compatible with Java.

The parser was edited so it could parse the keywords necessary for creating a GoalAgent. Next to the usual reactive behavior, another behavior was added to the behavior types, called proactive. Proactive behavior is used to define the goal-driven behavior of the Claim agent.

Then the different goals were added along with their possible arguments and goal-specific keywords such as maintain, achieve etc.

The arguments of goals were added to the parser as a new kind of argument\_list, called proposition\_list, this was made to make it compatible with arguments (or propositions) which contain parentheses and commas, which weren't allowed in the arguments\_list. Also the arguments\_list consisted of arguments, which could also be functions and other elements we did not want to have in the goals.

## Lexer

New tokens had to be added in order for the parser to understand these tokens. As such the keywords mGoal pGoal and aGoal were added as tokens. This way the parser is able to understand where and what kind of goal it has to parse.

## GoalAgent

The GoalAgent class contains all the mechanisms that implement the goal-driven behavior of the agent. This class extends the basic ClaimAgent class to add goal-driven behavior. Just like a claim agent, a goal agent is created by using the Jade platform. When a .adf2 file is parsed containing proactive-behavior, a goal agent is created. This happens in boot.java with the following line of code:

```
"allAgents.put(agentName, new AgentCreationData(agentName, GoalAgent.class.getCanonicalName(), new Object[] { cad, agentType, parameters, knowledgeList, null, environmentAgent}, containerName, !doCreateContainer));", where "environmentAgent" is a reference to the environment agent which simulates the blocks world environment.
```

In the setup phase, which is defined by the *setup()* method inside the goal agent, the agent is registered in the environment and it receives the initial state of the environment which is saved as a list called *latestEnvironmentState*, it also constructs its goal base which is the set of goal states as defined in the .adf2 file. At the end of the setup, the *initGoalStates()* method is called. This method goes through all the goals in the goal base, searching for the perform or achieve-goal with the highest priority and sets its goal state to "active". It also sets all the maintain goals to the "active" goal state. This way the agent knows which goals are active and could be used to calculate a plan so that they can be performed in a later stadium.

At the end of the setup phase we define the cyclical behavior of the agent. The cyclical behavior consists of two methods which the agent is going to run indefinitely: the *updateGoalStates()* and *execute()* methods.

The *updateGoalStates()* method goes through all the active goals in the goal base and checks how the goal state defined by the goal corresponds to the environment. In the case of a maintain-goal the *MaintainGoalState* of the maintain-goal is set to IDLE if the target state of the maintain-goal corresponds to the states in the environment, it is set to IN\_PROCESS if the target state of the maintain-goal does not correspond to the environment. In the case of an achieve-goal, we check if the goal-state corresponds to the environment state, if it does the goal has been achieved and its goal state will be set to SUCCE. We then remove all the goals with goal state SUCCE and FAIL, we now have to call the *initGoalStates()* method again to make sure that we have at least one active achieve-goal again with the currently highest priority.

The *execute()* method is used to select the active goal with the highest priority, to generate a plan for the selected goal and to execute it in the environment. In the case of an achieve-goal, a plan is generated by invoking the *computePlan(goal)* method, where goal stands for a goal for which the plan is generated and the actions of the plan are executed sequentially in the environment by invoking *environmentAgent.applyAction(action)*; In the case of a maintain-goal, a plan is only computed and executed if the goal is in the IN\_PROCESS state, otherwise no plan is computed and generated. Figure 15 shows a cycle in the execution of the GoalAgent as described above.

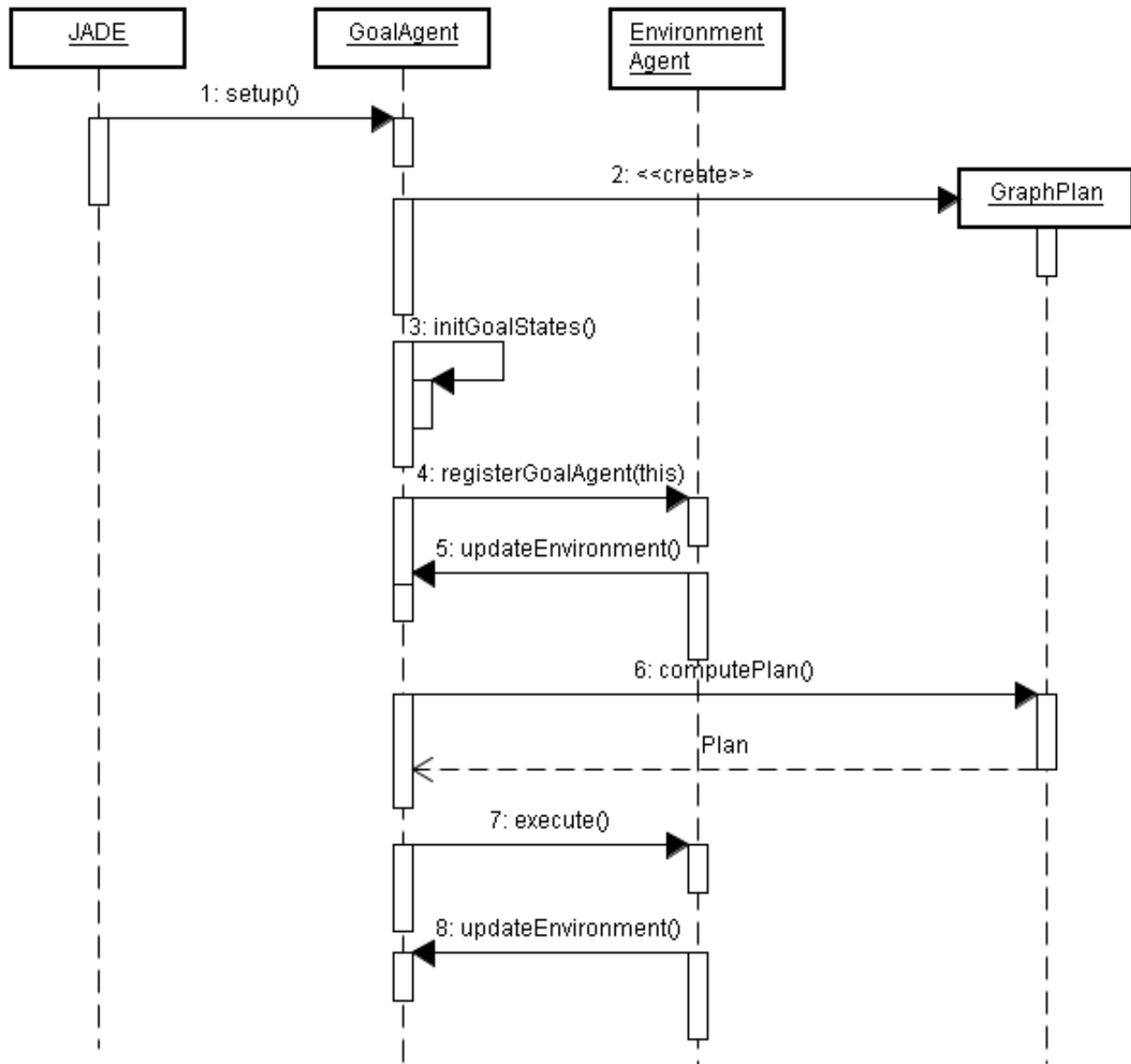


Figure 15: Sequence diagram showing a cycle in the GoalAgent execution

## EnvironmentAgent

The EnvironmentAgent class simulates a blocks world environment. The blocks world environment is a commonly used example in the Artificial Intelligence field. The environment consists of a table, and a number of blocks which can be placed on each other. The goal is to reach a certain configuration of blocks in the environment. The environment contains a list of strings which represents the state of the environment. For example when we have three blocks: b1, b2 and b3. A state could be ON(b1, Table), ON(b2, Table), ON(b3, Table), Clear(b1), Clear(b2), Clear(b3), Clear(Table), in which all these predicates are true. We follow a closed world assumption, so when a certain predicate does not exist in the environment we consider it to be false.

When initialized, the environment reads the environment, its possible actions and the initial state from two files. To do all this we have used the implementation of JPlan which defines a blocks world environment. Whenever an agent wants to 'enter' the environment, the agent has to register itself by calling the *registerGoalAgent(GoalAgent agent)* method. To apply an action in the environment the agent has to call the *applyAction(Action action)* method. This method checks if the precondition of the action are true, if so it adds all the add effects of the action and deletes all the delete effects from the environment states. Now that the action has been applied the environment agent will send the updated environment to all the agents that have been registered with this environment.

## Goals

We have defined three kinds of goals: achieve-goals, maintain-goals and perform-goals. The achieve-goal is implemented in the ClaimaGoal class, the maintain-goal in the ClaimmGoal class and the perform goal in the ClaimpGoal class, all these classes extend the ClaimGoal class. After the parsing of the .adf2 file, the corresponding objects will be created and added to the goal base of the agent. The behavior mechanisms of the agent are controlled by the GoalStates, as defined in the enum GoalState of the goals in the goal base. The superclass ClaimGoal has the following Goal states: FAIL, SUCCES, ACTIVE, OPTION or SUSPENDED. The maintain-goal has more specific states as defined in the enum MaintainGoalState, which are: IDLE and IN\_PROCESS.

## Illustrative Example

Suppose we want to create an agent with goal-driven behavior in the blocks world environment. In this illustrative example we will show how this can be done using the S-Claim platform.

First we must define a scenario. This scenario is an XML file which defines the agents and their parameters. To define an agent as a goal-driven agent, we have added the possibility of adding a new parameter called "goaldriven". When this is added to an agents parameters, it will be initialized as a goal-driven agent. Below you see these parameters as defined by <scen:parameters> tag. The goal-driven parameter is defined as <pr:param name="goaldriven" value="true"/>. Figure 16 shows what the XML scenario looks like.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <scen:scenario xmlns:pr="http://www.example.org/parameterSchema" xmlns:kb="http://www.example.org/kbSchema" xmlns:scen="http://www.example.org/scenarioSchema" />
3 <scen:jadeConfig mainContainerName="Administration" IPaddress="127.0.0.1" port="1099" platformID="1" />
4 <scen:adfPath>scenario/goal</scen:adfPath>
5 <!--<scen:environmentPaths environment="scenario/goal/environment.txt" operators="scenario/goal/operators.txt" /> -->
6 <scen:initial>
7   <scen:container name="Administration">
8     <scen:CLAIMAgent name="BlocksAgent" type="BlocksAgent">
9       <scen:parameter>
10        <pr:param name="goalDriven" value="true"/>
11        <pr:param name="courseName" value="CSCourse" />
12      </scen:parameter>
13    <scen:knowledge>
14      </scen:knowledge>
15    </scen:CLAIMAgent>
16  </scen:container>
17 </scen:initial>
18 <scen:timeline>
19   <scen:event time="2000">
20     <scen:CLAIMMessage>
21       <scen:to>SchedulerUPMCAgent</scen:to>
22       <scen:protocol>newSchedule</scen:protocol>
23       <scen:content>(<struct message newSchedule (<struct knowledge scheduledTo CSCourse Room04 >) </struct>)</scen:content>
24     </scen:CLAIMMessage>
25   </scen:event>
26 </scen:timeline>
27 </scen:scenario>

```

Figure 16: XML scenario

Now we must define the behavior of the agent in the corresponding .adf2 file. First we define the agent called BlocksAgent and then define its proactive behavior. Figure 17 shows what an .adf2 file looks like.

```

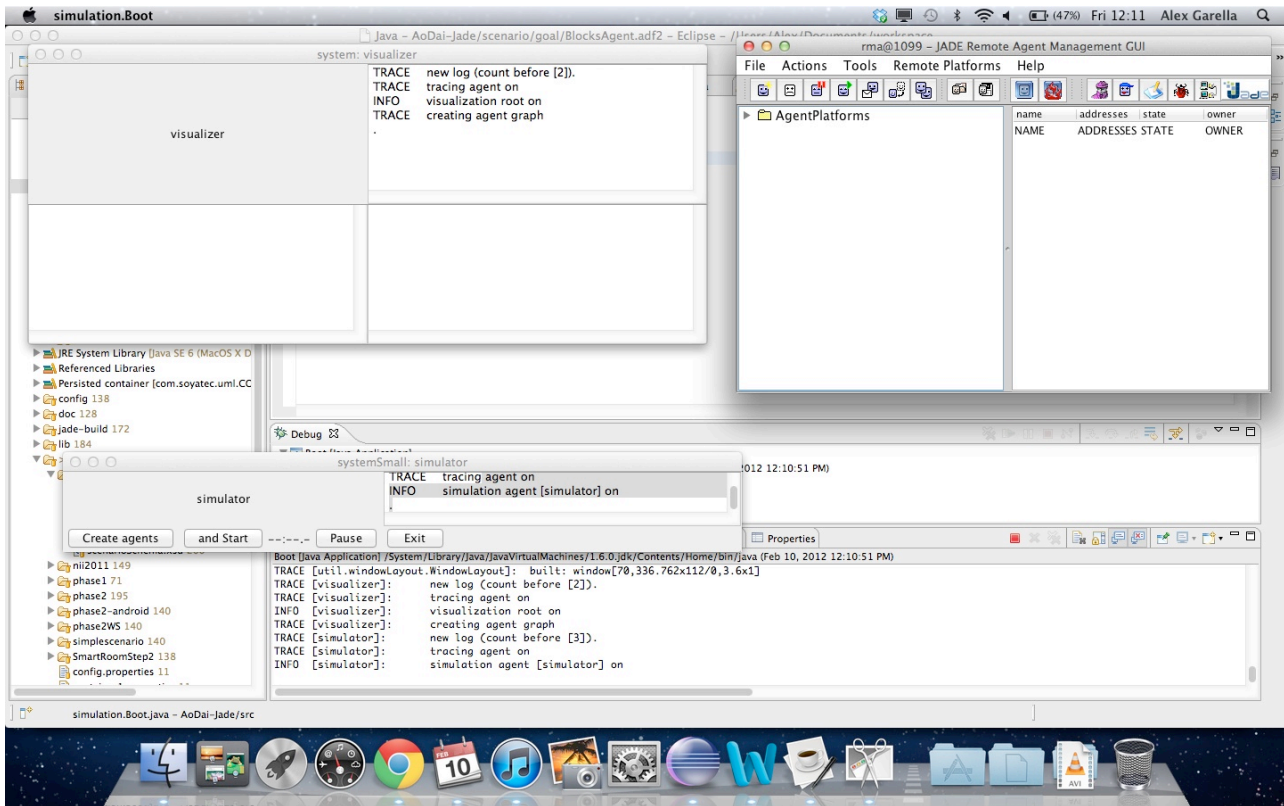
1 (agent BlocksAgent ?courseName ?parent
2   (behavior
3     (proactive goalBase
4       (aGoal name 3 achieve "ON(b2,b1)" "ON(b3,b2)")
5       (mGoal name 1 maintain "ON(b3,Table)")
6     )
7   )
8 )

```

Figure 17: an .adf2 file

In this example we have an agent with two goals. One is an achievement goal with achievement state ON(b2, b1) and ON(b3,b2) with priority 3 and the second is a maintain goal with maintain state ON(b3,Table) with priority 1. In the initial state all three blocks are on the table.

When we run this code the screens as shown in figure 18 appear.



**Figure 18: Appearing windows when running an agent**

Figure 18 shows the agent simulator and other inspection tools from jade. We now have to press the 'and Start' button to run our agent to see what happens.

```
GoalAgent computing plan.
ON(b1,Table)& ON(b2,Table)& ON(b3,Table)& Clear(b1)& Clear(b2)& Clear(b3)
ON(b1,b2)& Clear(Table)& ON(b1,Table)& Clear(b2)& ON(b1,b3)& Clear(b3)& ON(b2,b1)& ON(b2,Table)& Clear(b1)& ON(b2,b3)& ON(b3,b1)& ON(b3,Table)& ON(b3,b2)

Graph Created Successfully
Generated plan: [Move(b2,Table,b1), Move(b3,Table,b2)]
Performing action: Move(b2,Table,b1)
Environment: [ON(b1,Table), ON(b3,Table), Clear(b2), Clear(b3), ON(b2,b1), Clear(Table)]
Performing action: Move(b3,Table,b2)
Environment: [ON(b1,Table), Clear(b3), ON(b2,b1), Clear(Table), ON(b3,b2)]
GoalAgent computing plan.
ON(b1,Table)& Clear(b3)& ON(b2,b1)& Clear(Table)& ON(b3,b2)

Graph Created Successfully
Generated plan: [MoveToTable(b3,b2)]
Performing action: MoveToTable(b3,b2)
Environment: [ON(b1,Table), Clear(b3), ON(b2,b1), Clear(Table), ON(b3,Table), Clear(b2)]
```

**Figure 19: Output when running an agent**

Figure 19 shows the output that is generated by our code. Because of the higher priority we first execute the achieve goal. The generated plan is Move(b2, Table, b1) and Move(b3, Table, b2). After perform these actions sequentially in the environment the goal has been achieved. After an achievement goal has been achieved it is removed from the agent's goal base because it has been tagged as successful. The only remaining goal is the maintain goal. The agent generates the plan MoveToTable(b3,b2) and executes it. The maintain goal will never be removed from its goal base and a new plan will be calculated each time its maintain condition is violated.

## 4.3 Testing

Testing is a very important part of software development. Considering the nature of our project however, testing had an only limited priority. Implementation was, as was earlier stated, only used as a proof of concept. Therefore, the implementation did not need to be completely tested through. It should however show that our proof-of-concept is valid and thus that our research and idea are workable. We used some manual testing, where we tried to find a combination of types of goals with different priorities. Pretty much all combinations of goal types were used. We also combined different priorities, looking for the critical areas for the priorities (all the same, one slightly higher or lower, much higher or lower, etc.). By looking at these different combinations and matching the results with expected results (which was easy to check) we confirmed that our implementation worked as expected and thus that our research and idea were indeed valid.

Further testing by LIP6 should certainly be done, especially when the goal agents are extended to fulfill more complicated functionality, it would be interesting to see what would happen when, for example, an agent would migrate to another computational device. How would this affect the efficacy of the agent?

## 5. Conclusion & Future Work

In this report we have tried to describe the most important aspects of the theory underlying multi-agent systems and specifically the goal-driven behavior involved with these systems. By creating a theoretical framework of current work and proposed solutions we have attempted to propose a well-researched viable solution to the question of how to introduce the goal-driven behavior in the S-CLAIM language, namely by using both declarative and procedural aspects of goals. Where the declarative part is defined in the goal and the procedural part is computed by a planning algorithm.

We have proposed using the different goal types of Braubach, because of its intuitiveness, compactness and allowing for a great amount of different functionality. We have used the STRIPS language as a way to easily declare the goals, and by using a well-understood and seasoned algorithm such as GraphPlan to accomplish the plan generation. We have proposed that the GraphPlan can be extended even further to allow for faster computation, by converting the GraphPlan instance to an SAT problem. If modularity is properly maintained, it is possible to use arbitrary SAT solvers to compute a plan efficiently for any agent.

In the future the implementation should be finished in more detail, our aim was to make a concept of proof. The planning is currently based on the GraphPlan algorithm and works, but it was not possible to implement the SAT conversion due to time constraints. Proper research should be done into how to do this most efficiently. There are algorithms describing the conversion, but as far as we know, none of these take into account the costs and priorities of actions and goals. Modifying the SAT to include these parameters would increase its usefulness.

The environment agent we have used, was in fact not a true JADE powered agent, it was simply a mock-up, or placeholder for a future agent. Designing a true EnvironmentAgent driven by JADE would increase its usefulness and give greater insight in its efficacy in a more complex environment.

In the future more testing should be done using goal-driven agents with more complex goals and goal combinations. Our testing was limited to simple goals. By using more complex goals and goal combinations a better simulation of real world circumstances can be recreated. That way it would be possible to better assess the performance of the platform.

## 6. Bibliography

- [1] CLAIM: A Computational Language for Autonomous, Intelligent and Mobile Agents, Amal El Fallah-Seghrouchni and Alexandru Suna
- [2] Developing Multi-agent Systems with JADE, Fabio Bellifemine
- [3] Claim & SyMPA. Alexandru Suna
- [4] Jadex: A BDI Reasoning Engine, Alexander Pokahr, Lars Braubach
- [5] Goal Types in Agent Programming, Birna van Riemsdijk
- [6] SatPlan: Planning as Satisfiability, Henry Kautz
- [7] Planning as Satisfiability, Henry Kautz
- [8] GraphPlan, Avrim Blum
- [9] <http://byaccj.sourceforge.net/>

# Appendix A: Project Plan

## 1. Introduction

The project plan gives an overview of the activities that have been formulated to address and solve the problems associated with our Bachelor project, which consists of introducing goal-driven behavior in the S-CLAIM programming language. This chapter will be an introduction to the project plan, describing the origins of the project. Chapter 2 describes the project assignment in detail. Chapter 3 describes our approach to the project. Chapter 4 describes the organization of the project. Chapter 5 describes how the project has been planned and chapter 6 describes how we are planning to ensure the project quality.

### Origin

S-CLAIM (SMART- Computational Language for Autonomous, Intelligent and Mobile Agents) is a programming language that has been designed to program mobile multi-agent systems by the computer science laboratory, LIP6 of the UPMC University of Paris. One of the main goals of the language is to enable people without any programming experience to implement the behavior of the multi-agent system. To achieve this goal the language must contain high level constructs which are easy to understand for non-programmers. For this reason it was decided to introduce goal-driven behavior in the S-CLAIM programming language. The formal project assignment can be found in the next chapter.

## 2. Project assignment

This chapter will describe the assignment that we have been given by LIP6.

### Project environment

The project environment consists of the research department called LIP6 at the UPMC university at Paris. This is the sole organization immediately involved in our project, which was deemed S-CLAIM. Further explanation of S-CLAIM and its constituents can be found in our report. A professor was involved in our research as were a few PhD students, with one PhD student giving us direct supervision on a day-to-day basis.

### Project goal

The goal of the project was to research how the S-CLAIM project could be further enhanced, more specifically; research was needed towards goal driven behavior. It is important to note that effective research was much more important than actual implementation. The research should be documented for further use.

### Assignment Description

The assignment is to research and implement some goal-driven aspects into the existing S-CLAIM project. Implementation should allow users to implement goal driven behaviors

using a given set of language constructs. Research should give a state-of-the-art description of goal-driven behavior and an approach to implement it. Some implementation should be done to confirm the validity of the research done (see appendix A).

### **Product- and service deliverables**

Research was the main part of this project. A report should be delivered containing the findings of our research as a basis for further advancement of the S-CLAIM system. Implementation had to encompass a simple yet viable solution to the given problem of producing goal-driven behavior. Functionality should include the ability to define the goal driven behaviors of agents, plan generation and execution of action to modify the environment.

### **Requirements and limitations**

Most importantly there should be some substantial research done into the field. Understanding the current state of affairs is paramount. Furthermore, possible solutions should be researched and clearly defined. As the report created from the research would serve as a guide for future work, the report had to be self-sufficient, that is, it should be more or less sufficient for a future researcher to continue our work. Demands for implementation were solely that they would proof the efficacy of our research. It is important to note that the research itself would still be useful without implementation, it would still gain insight in the domain. Therefore the research had a much higher priority than the actual implementation. Limitations were not very obvious, except for the limited amount of time we would be able to counsel with others in the field, there were no further financial or material resources required for us to do our job.

### **Crucial success criteria**

The project could be deemed a success if we would be able to produce a satisfactory report on the research done. As long as the project supervisors were pleased with that which was produced, the project could be deemed a success. Implementation would also have to meet certain quality criteria, but these were never explicitly stated by the supervisors involved, but were nevertheless met.

## **3. Project Approach**

This chapter will answer the questions on how we will address and solve the problems which have been posed in the preceding chapter of this project plan. The aim of this chapter is to bridge the gap between the project assignment and the desired results which have been defined in chapter 2.

## Approach

The project consists of two main activities.

1. Firstly doing research on how to define the goal structures in S-CLAIM, how to use these structures to define the internal mechanisms of the agent, and how to generate plans considering the agent mechanisms and the goal structures.
2. Secondly implementing a proof of concept of the ideas that have been developed during the research on top of the existing S-CLAIM platform.

The focus of this project will lie with the research, there are two reasons for this approach.

1. The project assignment was envisioned as a research project by the LIP6 laboratory. The focus of the LIP6 laboratory lies in research not in software engineering, therefore it is only natural that a project at this institute will mostly consist of research.
2. LIP6 has granted us a high degree of freedom to come up with a solution to introduce goal-driven behavior in S-CLAIM. This degree of freedom comes with the responsibility of doing more research than usual to make sure the ideas are well envisioned and have a greater possibility of success.

The approach consists of doing roughly 2 months of research to assess the current state of the art in field of multi-agent systems, goal-driven behavior and plan generation. The focus will lie with how the latest developments in these fields can be used to successfully introduce goal-driven behavior in S-CLAIM.

After the research 1 month will be dedicated to introducing the researched notions into the existing S-CLAIM platform. To be able to introduce the appropriate constructs the lexical analysis and parser of S-CLAIM have to be modified to be able to handle goal constructs that are defined in the source-code of the agent. Consequently the existing S-CLAIM agent has to be extended to implement the envisioned internal mechanisms to handle the goal constructs that have been defined in the agent's source code. Finally when the goal handling mechanisms of the agent have been implemented a plan generation algorithm has to be implemented so that the agent can generate plans to achieve the specified goals.

## Risk Factors

The freedom that we have been granted to us in solving the problems described above obviously come at a risk. As we are relatively new to multi-agent systems there is a risk that our approach will fail as we do not have enough experience in this field. To minimize this risk of failure we will work closely with our supervisor Cedric Herpson who has a great deal of experience in these fields. We will also have to gain more knowledge on multi-agent systems by reading more about the basics. An introduction to Multi-Agent Systems by Michael Woolridge is a good example of a book to freshen up our multi-agent systems knowledge.

## Quality Requirements

The quality of our research will continuously be assessed by our supervisor as it is very difficult to define the quality of the research beforehand. The requirement of the quality of

the implementation part is that the ideas that have been developed during the research will be able to be demonstrated in a simple blocks world scenario.

## **4. Project Organization**

The goal of the project organization is to create a more transparent overview of the project organization.

### **Organization**

There were no official roles or responsibilities assigned specifically to us, but were more generally assigned to the both of us.

### **Staff**

Even though many people are involved and actively working for the S-CLAIM project. Only a few people were actively involved in our work. This was prof. Seghrouchni, the head of the department and the lead researcher. She was the one who took us in for the project and was also the person who held final responsibility for the project. Cedric Herpson was a PhD working under Prof. Seghrouchni and was our direct supervisor. Due to the many responsibilities of Prof. Seghrouchni, she left the day-to-day supervision of our project up to him.

### **Administrative procedures**

Administration consisted of filling in forms and supplying documents to get internet and network access at the university compound, a key to access our office at the LIP6 department and verification of our origin at TU Delft and as a student in general along with agreement of Prof. Seghrouchni in order to receive our monthly financial compensation.

### **Financial organization**

Except for our monthly financial compensation no additional costs were made.

### **Requirements for contractor**

It was agreed that the provider would make sure that we were financially compensated for our stay in Paris.

Opsomming van voorwaarden, die gerealiseerd dienen te worden door de opdrachtnemer om het project volgens plan te kunnen uitvoeren. Deze voorwaarden zijn gerelateerd aan en aanvullend op de inrichtingsaspecten.

### **Requirements for principal**

We would do research at LIP6. No specific requirements about office time were made, but of course the project had to be finished satisfactory.

## 5. Planning

This chapter will describe the planning of the project.

### Assumptions

The assumptions that are made regarding the planning are that background studying on multi-agent systems is done in spare time. The research paired with goal-driven behavior and planning are assumed to be included in the available hours in the normal 40 hour workweek. The duration of the project is assumed to be 3 months of 40 hour workweeks.

### Activities

As mentioned before the project consists of two main parts. A research part and an implementation part. The research part will consist of 2 months of research on the following points.

1. Defining appropriate goal structures in S-CLAIM to be able to introduce goal-driven behavior.
2. Design mechanisms internal to the agent to handle the goal structures.
3. Define the plan generation so that the agent can generate a plan to achieve its goals.

During the research of these points we will also be finding out what the current state of the art of the fields encompassing these points is. Two months will be needed to carry out the research part.

After the research part one month will be used for the implementation. The implementation will consist of the following points:

1. Modify the lexical analysis and parser of the currently implemented in S-CLAIM so that it can handle declarative goals.
2. Extend the current S-CLAIM agent by implementing an internal mechanism that handles the goal structures.
3. Implement plan generation so that a plan can be generated to achieve the specified goals.

These activities will need 3 months of time which is the specified time of a bachelor project.

## **6. Project Quality**

The delivered product should consist of a research report which shows the understanding of the domain and provides a guide for future research. This has the highest priority and is acceptable when the supervisors qualify it as acceptable when it is extensive enough, knowledgeable and understandable.

### **Process Quality**

Requirements to be met:

- expertise in domain
- communication (the results found need to be communicated accurately)

### **Proposed measures**

Measures used to maintain quality were based mostly on verbal and non-verbal feedback. Presentations were given to the supervisors on a monthly basis as well as to colleagues in the field. Afterwards a review was presented to us after deliberation.

# Appendix B: Research Proposal

Amal El Fallah Seghrouchni  
Professor at the University Pierre & Marie Curie  
Head of SMA team / Delegation at the CNRS  
4, Place Jussieu - 75252 Paris Cedex 05  
[Amal.Elfallah@lip6.fr](mailto:Amal.Elfallah@lip6.fr)

Paris, 8th of July 2011

**Subject : Research Proposal for Gerard Simons & Alex Garella, 1<sup>st</sup> September 2011 – 1<sup>st</sup> December 2011**

**Title : Goal-Driven Behavior for CLAIM agents**

**Advisor : Prof. Amal EL FALLAH SEGHRUCHNI**

**Team Project : Cédric Herpson, Andrei Olaru, Nga Thi Thuy and Marius Tudor Benea**

## **Description**

The CLAIM programming language (Computational Language for Autonomous, Intelligent and Mobile agents) has been developed at the LIP6 laboratory [Suna and El Fallah Seghrouchni, 2004], with the purpose of offering an easy way of programming cognitive mobile agents, using a simple language, without the need for the programmer to know any more advanced programming languages, like Java or C. Once the code is written in CLAIM, it was executed by the Sympa platform, written in Java. CLAIM allows for easy implementation of mobile agents that have reactive or proactive behavior. The functioning of CLAIM agents is also inspired from the mobile ambients of Luca Cardelli [Cardelli and Gordon, 2000].

After successfully demonstrating the usefulness of CLAIM in Ambient Intelligence applications [El Fallah Seghrouchni et al 2010], a team at LIP6 (Andrei Olaru, Thi Thuy Nga Nguyen and Marius-Tudor Benea) started in 2011 the development of a new platform for the execution of CLAIM2 – a more simple variant of CLAIM. This platform will have improved organization and capabilities, and will be based on the Jade agent development framework. It will also feature a component for CLAIM agents executing on mobile devices (Android smartphones), which is currently being developed by Marius-Tudor Benea.

In this context, this stage will focus on the introduction of goal-driven behavior in CLAIM2. Goal-driven agents are able to act in an autonomous manner, by reasoning on solving the goals and taking appropriate action [Braubach et al, 2005]. More precisely, the stage requires the implementation of the language constructs that allow the programmer to specify goals for the agents, and of new types of agent behavior

by means of which the agent will try to fulfill its goals. These elements will be integrated in the CLAIM2 platform developed at LIP6.

## **References**

Braubach, L. and Pokahr, A. and Moldt, D. and Lamersdorf, W. (2005), Goal representation for BDI agent systems, *Programming Multi-Agent Systems*, 44-65.

Cardelli, L. and Gordon, A. D. (2000). Mobile ambients. *Theor. Comput. Sci.*, 240(1):177-213.

El Fallah Seghrouchni, A., Olaru, A., Nguyen, T. T. N., and Salomone, D. (2010). Ao Dai: Agent oriented design for ambient intelligence. In *Proceedings of PRIMA 2010, the 13th International Conference on Principles and Practice of Multi-Agent Systems*.

Suna, A. and El Fallah Seghrouchni, A. (2004). Programming mobile intelligent agents: An operational semantics. *Web Intelligence and Agent Systems*, 5(1):47-67.