A Goal-Oriented Execution Module Based on Agents

Javier Palanca, Vicente Julian, Ana García-Fornes
DSIC - Universidad Politécnica de Valencia
Camino de Vera s/n, 46022
Valencia, SPAIN

Abstract—In this work a goal-oriented execution module is presented. Its aim is to explore the definition of new Operating Systems abstractions based on paradigms that have been developed during the recent years, filling the gap between classical Operating Systems designs and current trends in software engineering. This works makes an approach to a system where applications are based in technologies like multi-agent systems and service-oriented computing, defining the basis that an Operating System needs to run the abstractions provided by these paradigms.

I. INTRODUCTION

The amount of developed software and its complexity has currently grown so huge that it has lead to the discovery that the traditional paradigms of software development are not enough to create complex software. That is why there is a constant work on new paradigms, to improve the level of abstraction needed to develop the increasingly complex applications. Among these paradigms, we can highlight the multi-agent systems.

The multi-agent systems paradigm allows us to develop agents which are autonomous, proactive, reactive to their environment and social skills to communicate with others and even reach agreements, share common goals and self-organize[1]. Some of these paradigms use plan-based systems to define and achieve their goals using diverse technologies[2], [3].

But every paradigm needs a framework where it is executed. Old imperative and object-oriented paradigms use the classic abstractions provided by traditional operating systems. Abstractions such as processes, threads or files are largely supported by those systems from their base. With the changes provided by software engineering we can see that the more you increase the level of abstraction used by paradigms, the more complex is to generate final code from them. In this line, this work proposes to support the new higher-level abstractions, such as multi-agent systems, from the operating system execution module. This is going to be done by bringing as much as possible the system abstraction layers to these paradigms in order to make easier building applications.

The multi-agent systems paradigm consists of very high level abstractions (agent, message, ontology,...) and even more if we use the Belief-Desire-Intention (BDI) approach [2] (desires, intentions, knowledge base,...). These abstractions have little correlation with the executing support provided by classical operating systems. This leads us to have a very high stratification in the system, where previously it was necessary to introduce layers that support these higher level abstractions. These abstractions are finally converted, by means of more or less artificial changes, in the low-level abstractions offered by the operating system [4].

We can not see an agent as a process, it has not an entry point and an exit, but it is an entity that lives in the system. It communicates with its environment, has goals and does its best to meet them. However, the support offered to us from the operating system execution model is process-oriented, where the OS (operating system) has no knowledge of what each system application intends to do. That is why agents need to run agent platforms in order to run an agent model in a current operating system. These agent platforms are usually developed exclusively for the support of a particular agent model[5], [6], [7]. However, these platforms provide a limited view of the system. They are not able to manage the execution of each of the agents because they have no control over the execution model of the operating system. Even in cases where such support is provided through a virtual machine [8], which is capable of modeling a specific hardware, we can not completely control the execution model, since the virtual machine which is running agents is ultimately another OS application.

In this work a goal-oriented execution module is presented. This module aims to provide an alternative and powerful paradigm to the common Operating Systems. What is shown with this execution module for system applications (using the paradigm of multi-agent systems) is the possibility of developing applications with the flexibility, dynamism and fault tolerance of an agent system without the need of too many layers between the application and the operating system.

To design this execution module we have been inspired by the BDI agent model. This agent model has been selected for the facilities it can offer to our execution module. The BDI agent model[2] is well structured and provides some advantages to the agent-based execution model. In this model, agents are driven by goals and plans.

The proposed execution module uses goals as information to select the plans that may lead to the execution of applications. The advantages of this module could be the dynamism, thanks to the opportunity of composing on-line plans to meet goals, the fault tolerance, with the ability to repair a plan when it fails, and the optimization, where the system could automatically and transparently look for best quality solutions for their agent goals using trust and efficiency metrics.

A. Related work

Operating Systems research is always trying to improve their security, efficiency and reliability. This is one of the
great challenges of the OS that remain to be overcome. Several studies have focused on improving certain aspects of OS as data access or the input/output (I/O) abstractions, leading to propose new abstractions in this field (file, object, socket, ...)[9], [10], [11]. However, no significant progress has been made in implementing OS execution models. Introducing a deliberative component in the implementation of the OS model (as proposed in this work) adds more information than can be used by the OS kernel to decide a better way to schedule a running process. A different approach to improve the system reliability is proposed by the experimental Operating System called Singularity[12]. This OS uses code evaluation at compile time with a specific programming language named Sing# [13] to determine whether a piece of code is right or not. The complication of this proposal is the static component that the evaluation of code introduces, which removes flexibility to the system. This proposal is also tied to the process abstraction. One reason why no significant progress has been made in the OS execution model abstractions, such as the process, is that these abstractions are closely tied to current hardware. Processors are designed to work optimally with processes. Thereby, when adding improvements to the OS execution model, as well as defining new execution abstractions (as the proposed in this paper) would be interesting to start thinking about adapting the hardware to such abstractions.

The following sections will present the new proposed execution module using a goal-oriented computing paradigm based on agents. Then an application example of this model will be shown and finally some conclusions are presented.

II. A GOAL-ORIENTED EXECUTION MODULE

The execution module presented here uses the multi-agent systems paradigm with a model inspired by the BDI model. It incorporates the abstractions of agent, knowledge base, services, goals and plans[14]. Its purpose is to define an execution support based on a different computation paradigm that provides the advantages presented earlier in this document. The execution module operates on an operating system kernel, which provides the other necessary functionality of a common OS, such as memory management, security, etc. This module is made up primarily of the following components (see Figure 2):

- **An Off-line Planner**: it provides precompiled plans for their execution from a set of goals. This component is executed in the design phase of the agents and its motivation is to help agent designers to provide a set of precompiled plans that help the agent to reach their goals.

- **An On-line Planner**: it is able to repair or refine running plans. This planner is executed concurrently inside the Execution module and its task is to help the agents to reach a goal when the agent has no precompiled plans that guide him to the goal completion by composing or repairing plans.

- **A Services cloud**: plans are composed as a set of services offered by agents. We represent the set of services with the cloud abstraction following the paradigm of Cloud Computing[15] or Service-Oriented Computing[16], where services are available without knowledge of their exactly location and services are the key to solve problems or applications. At this level of abstraction we represent the set of services as a services cloud since its location is not determinative.

  - **A Runtime Engine**: it takes the plans provided by their planners and manage their execution by transferring the service execution to the OS kernel. It uses the services cloud if necessary.

  - **A Deliberation Engine**: It is responsible for deciding how and in what order plans are executed. This engine is permanently running in background evaluating the goals that the agents want to achieve and selecting them for its completion. This component interacts concurrently with the Runtime Engine and the On-line Planner.

Under this computing paradigm the goals of each agent are selected by the execution module for their achievement. Then the module chooses the appropriate plan to meet each goal. Note that the agent model preserves its desirable features like autonomy and proactivity since is the agent who activates its own goals when it decides he wants to achieve them. The module just provides the needed resources to help achieving them. Plans may be provided by the agent itself or can be compounded on-line. These plans are a sequence of services offered by the agents both locally and remotely. It is also an agent choice to share its precompiled plans with other agents. The basic running elements are the services that make the plans. Plans are provided to the module in two different ways: the off-line generation of the plan or the on-line generation of the plan by the on-line planning module. Once the plan that meets the active goal is selected, the Runtime Engine activates the services that comprise the selected plan.

In next section we will define the execution entities involved in the execution module.

A. Execution entities

In the proposed execution model there are four elements involved: agent, goal, plan and service.

**Agents** which are present in the system looking for their goals comprise a set of agents called AS. In our execution module an agent $A \in AS$ is defined through the following tuple:

$$ A = \{KB, SS, CP, GS\} $$

Where:

- **KB** represents the agent knowledge base.
- **SS** represents a set of services offered by the agent. This services are used by the agent to perform its goals, but they can also be offered to other agents to help achieving their own goals.
- **CP** represents a set of precompiled plans provided by the agent to meet its goals.
- **GS** represents the set of goals that the agent wants to achieve.

**Services** in this Execution model are OWL-S services[17]. A OWL-S service $S_i \in SS$ is defined by the tuple:

$$ S_i = \{SP, GR, PM\} $$
0. WHILE TRUE:
1. . SELECT agent A FROM AS:
2. . SELECT goal G FROM A->GS:
3. . IF ISPOSSIBLE(G) AND ISCONSISTENT(G):
4. . . plan C = SELECT_PLAN(G)
5. . . IF NOT C:
6. . . C = PLANNER(G)
7. . . IF C:
8. . . . WHILE NOT FINISHED(C):
9. . . . . TRY:
10. . . . . RUN(C)
11. . . . . EXCEPT:
12. . . . . C = REPAIR_PLAN(C)
13. . . . ENDWHILE
14. . . IF CHECK_POSTCONDITION(C) IS TRUE:
15. . . . PURSUED(G)
16. . . ELSE: NOT_PURSUED(G)
17. . ELSE: NOT_PURSUED(G)
18. . ENDIF
19. ENDWHILE

Fig. 1. The execution algorithm

Where \( SP \) is the service profile, \( GR \) the grounding and \( PM \) the process model of the service. The service profile defines what the service does. The grounding defines how to interact with the service and the process model defines how the service is used.

OWL-S service process model can be composite processes and atomic processes. A composite process is a set of atomic processes (which have no internal structure and run in a single step) with an internal structure built up by composite and atomic processes and a few control constructs (sequence, if-then-else, choice, etc).

This kind of OWL-S service is a well-defined standard which provides this model enough power to construct all the functionality provided by an agent.

The services that make up a plan are the real executable part of a plan. A service \( S_i \) is also composed of a pre-condition \( P \), a post-condition \( Q \) and a set of inputs and outputs. The post-condition is the Goal entity that the agent wants to achieve. The pre-condition \( P \) is a prerequisite for the execution of a service. The postcondition \( Q \) is the impact that will drive the execution of the service \( S \). Both \( P \) and \( Q \) are defined in the functional aspect of the service profile.

As we presented in the above definition each agent has a set of plans that guide it to achieve its goals. These plans consist of a composition of OWL-S services.

Since a composition of OWL-S services is a composition of services, which include both atomic and composite processes and control structs, we define a Plan as a process model composed by one or more composite process models (again, including composite services, atomic services and control structs). A Plan defines the way to achieve some results or post-conditions by joining different OWL-S services which can be connected. Composite services or even atomic services can be seen as very simple plans, but we also define a plan as the result of joining different composite services in order to achieve a goal.

Next, we present the deliberation and runtime process of the module. Finally an example with an execution trace and some implementation and results are shown.

III. THE EXECUTION PROCESS

The process of running agents is divided into two main steps: selection of plans to execute based on the active goals and execution of the services that make up the selected plans.

The selection of plans to be executed is done by the Deliberation Engine of the execution module. The deliberative process must take into account what goals can be selected for their achievement, without conflicting with other active goals. Furthermore, the deliberative process should decide the most appropriate plan for achieving the intended goal or, alternatively, instructs the on-line planner the composition of an appropriate plan.

The deliberation process and the execution of plans includes the following steps (Figure 1):

- At the initial stage an agent has a set of goals defined in its GS. Which means that the agent has determined that it wants to achieve these goals and requests the operating system to schedule the plan execution.
- The role of the Deliberation Engine is to analyze the goals of the agents and select potential goals to activate for each agent. Before the activation of a goal it is checked that the goal is consistent with other active goals and it is possible its activation at this time. That is, it does not conflict with other goals of the system or other agents. Conflicts between goals can appear both between the agent goals and between goals of different agents. This is usually
the case with access to system resources simultaneously with contradictory orders (e.g. accessing a file at the same time). When a conflict occurs the execution module could apply a heuristic that would prioritize goals, putting the conflicting goal with a lower priority off. This consistency of goals can affect all running agents, since they share the same hardware and access to resources.

- For each active goal a plan that meets that goal is selected from the agent set of precompiled plans. The selected plan is transferred to the Runtime Engine. The Runtime Engine is the module responsible for executing the plans. This module is in charge to select from among all the service plans that are running.

- Each service selected for execution is managed by the OS kernel. The kernel runs a scheduler and a dispatcher, responsible for selecting which service is set to run and sets it for execution. Thus, the OS kernel and the execution module invoke these actions following the order: dispatcher(scheduler(deliberation())).

- The Runtime Engine checks the state of the running plans and invokes the corresponding service if the precondition is valid. When the execution of the service is finished the Runtime Engine checks the postcondition of the service and the agent’s knowledge base and makes the transition to the next node in the service graph according to these values. If the postcondition of the service is not met then the plan is aborted and the goal that caused the execution of the plan remains available for been achieved.

- In case of absence of a precompiled plan that meets the selected goal the execution module launches in background the on-line planner module (Figure 1, line 6). This module will try to compose a new plan that meets that goal, searching among the services offered by other agents. If there is no plan the module marks the goal as not pursued (Figure 1, line 17).

- Agents have also the choice of share their own precompiled plans. If an agent knows how to achieve a defined goal, it can teach other agents that at the present time have expressed that they want to achieve the defined goal.

- It is possible that during the execution of a plan, one of its services fails. This may be because is not an available service or because it is faulty or it went out of service abruptly. If a service fails during the execution of a plan, the on-line planner will attempt to repair the plan from the nearest possible point to continue its execution. The on-line planner uses known algorithms to make a service composition. Some of them are the Service Aggregation Matchmaking (SAM)[18], from Brogi, Corfini et al. and XPlan[19], [20], from Klusch, gerber et al. Both algorithms are able to make a composition from a set of OWL-S services. This process is run in background since this is the slowest process of the execution model.

- There is a concurrent deliberation about the available goals, deciding whether to enable or disable one. If necessary a new plan is activated (a precompiled one or an on-line compounded one) and finally the runtime engine selects the next service to run.

- Finally, when the execution of the plan is finished, the execution module must verify the effects or postconditions of the plan, since they may lead to the instantiation of new targets following the completion of the first. If the postconditions are not met, the goal can not be marked as pursued (Figure 1, lines 14–16).

After reaching a goal, it can be removed from the active goals set, unless it has some specific property such as persistence or periodicity.

In Figure 3 you can see the decision diagram of the module to achieve a goal. The involved components in this process are the Deliberation Engine, the Runtime Engine and the On-line Planner. The execution of a plan has its own life cycle, since a plan is a composition of interrelated services.

In Figure 2 the relationship between the different components of the execution module and the agents is represented. The Deliberation Engine uses the goal set and the plan library to select the most suitable goal and the plan that leads to its achievement. The Runtime Engine needs to interact with all agents’ services to run the scheduled service at each time.
Finally both the service itself and the Deliberation Engine have the ability to access the knowledge base of the agent. Services may change the knowledge base of the agent during its execution while the Deliberation Engine should be able to query the knowledge base to validate preconditions and postconditions in its deliberative process.

In this paper we present the proposed execution process focusing on one component of this module: the Runtime Engine. We introduce briefly how the agents are executed in the Runtime Engine and its relationship with the other components of the execution module, as the Deliberation Engine and the On-line Planner.

A. The runtime engine

The Runtime Engine is the component in charge of controlling the life cycle of the system entities and their running process.

The atomic service entity has a life cycle and running process inspired by the traditional operating system’s process abstraction. They are autonomous processes which run in their own execution context but with some privileges. Some of these privileges are the access to the knowledge base of the provider agent and the access to the knowledge base of the applicant agent, since the service may modify both knowledge bases during its execution. The life cycle of the services is similar to traditional processes. Services can be ready to run, running and sleeping (see the execution model of traditional OS[21]).

The additional work for this Runtime Engine comes from the plan entity running process. To invoke a service it should be part of a selected running plan. This plan also needs to fulfill each step of its process model, following the logical flow that determines its preconditions and postconditions. The task of visiting the process model of each active plan and check the preconditions and postconditions of each node belongs to the Runtime Engine.

As shown in Figure 1, when a plan is selected for its execution by the Deliberation Engine it calls the RUN(C) method, which invokes the main Runtime Engine (see Figure 4).

1) The Runtime Engine first checks that the precondition of the plan is valid and it can be executed.

2) Once the plan is running, the Runtime Engine selects the first node of the plan service graph and invokes it in the Operating System kernel.

3) For every instantiated service the Runtime Engine checks previously its precondition and, after the service execution is finished, it checks the service postcondition. If the postcondition is valid the execution of the plan can continue.

4) At this point the Runtime Engine checks the preconditions of all the service neighbors, which are all the nodes from the process model that are directly accessible from the given node through a control construct. For each service whose precondition is valid the Runtime Engine instantiates it and repeats the process. This process continues until the service process model reaches a final node or their services fail and a plan reparation is needed (using the On-line Planner).

The life cycle of a plan switches from not selected to selected, when it is selected for execution. The Runtime Engine also manages which atomic services of the plan are being executed at each time and the service flow via the control structures.

Finally, the entity that motivates this Execution model is the Agent. The agent entity has four possible states:

- **Applicant**: The agent has goals to pursue and does not offer any service.
- **Provider**: The agent offers services to other agents but has no current goal.
1. FOREACH plan IN SELECTED_PLANS():
2.   IF CHECK_PRECONDITION(plan) == TRUE:
3.     Service_Queue = EMPTY_QUEUE()
4.     n = SELECT_FIRST_NODE(plan)
5.     APPEND(Service_Queue, n)
6.     WHILE( HAS_NODES(Service_Queue) ):
7.       n = GET_NODE(Service_Queue)
8.       IF CHECK_PRECONDITION(n) == TRUE:
9.         INVoke(n)
10.        IF CHECK_POSTCONDITION(n) == TRUE:
11.           FOREACH node IN NEIGHBORS(n):
12.             APPEND( Service_Queue, node )
13.         ENDIF
14.       ENDIF
15.     REMOVE( Service_Queue, n )
16.   ENDWHILE
17.   IF CHECK_POSTCONDITION(plan) == TRUE: RETURN TRUE
18.   ELSE:
19.     RAISE REPLANNING_EXCEPTION
20. ENDIF

Fig. 4. RUN(C): The Runtime Engine algorithm

- **Provider-Applicant**: The agent has goals to pursue and also provides some services for both its own use and for use of other applicant agents.
- **Inert**: The agent has neither current goals nor provided services. It is ready to leave the system.

IV. Execution Trace

This section will expose a sample trace where the different steps that follow this execution module for achieving a goal are shown. For simplicity we have prepared a simple scenario with few elements and a single goal to achieve. To show the flexibility of the system we will simulate an error in the trace, showing the fault tolerance of the module.

In Figure 5 the elements that form part of the example are shown. There are four agents, where some of them are provided by the system (Providers) and some of them are looking for their goals (Applicants). *Printer Agent*, *System Agent* and *CUPS Agent* are all agents providing services and with no immediate goals to achieve. There is also an applicant agent (called *User Agent*) which does not provide any service or plan but is looking for a goal, which is to print a document.

Print and CUPS agents are specialized agents whose task is to provide services to print in the system. The System Agent provides low-level services which may be needed by other services.

In order to make the example easy to read there is only one precompiled plan (provided by the Printer Agent) which tries to print a document using the LPP protocol (a known protocol to communicate with printers). The plan is shown in Figure 6.

As an example, we’ll follow an execution trace using this plan:

1) Initially, the Deliberation Engine would select a goal of an agent. For simplicity there is only one goal, which is *Document_Printed* from *User Agent*. Since there is only one goal, the Deliberation Engine selects it.

2) Searching among the available precompiled plans the Deliberation Engine finds the plan *Print_Document* (Figure 5). This plan states having as input prerequisite for its execution the *Document* object in the agent knowledge base. It also says it has the effect *Document_Printed*.

3) As long as the plan meets the precondition (the agent *knows* *Document*), the deliberative engine will select the plan for its execution since its postcondition is compatible with the desired goal (it generates *Document_Printed*).

4) The Runtime Engine executes the service *Convert_Document*, achieving as effect the value *PS_Document*.

5) The Runtime Engine executes the service *Validate_Document*, achieving as effect the value *Valid_Document*.

6) The Runtime Engine executes the service *Locate_Any_Printer*, achieving as effect the value *Printer_URI*.

7) To show the advantages of running this model, we introduce an error at this point. Let us assume that the service *LPP Print* is unavailable (the agent that provides the service is not connected, the service is saturated, or maybe the selected printer does not support the LPP protocol).

8) At this time, the Runtime Engine would ask the online planner a repair of the running plan to continue the execution of this agent.

9) The planner would return the plan shown in Figure 7. This repaired plan continues where the other plan has failed its execution and replaces the failed service with...
other structure thanks to other services offered by the CUPS Agent. The new plan has a very similar structure but replaces the LPP service with a choice for other two standard protocols (CUPS and PS).

10) Finally the execution of the plan is ongoing through the services CUPS_Print, USB_Transport and Print_Validation.

11) When the execution of the service Print_Validation has finished, the User Agent has in its knowledge base the fact Document_Printed, so the goal has been achieved and it can be removed from the agent set of goals.

A remarkable aspect of the User Agent is that despite the selected plan has failed, it has been able to achieve its goal on a completely transparent way to the agent through the ability of replanning of the execution module. With this module the success degree of goal achievement is higher than on classic BDI systems. This module has also the ability of providing system services for the plan composition, allowing the OS to work with this paradigm.

V. IMPLEMENTATION AND RESULTS

As part of this work a simulator has been developed to demonstrate the execution capabilities presented in this paper. This simulator has shown how introducing a replanning component can considerably improve the success degree of goal completion. The simulator has been designed to run a huge sets of agents with different goals to achieve and with diverse services to provide. We are mainly interested in how agents increase their success degree when they have more precompiled plans and when the number of services that compose the service cloud goes up.

Given our previous experience developing a flexible multi-agent platform [22], [23], [24], it became natural to develop a
simulator which implements the aforementioned features and in which the platform layer was substituted by a simulated kernel layer. This kernel layer allows us to simulate agent services which would be provided by the OS itself.

Due to the OWL-S service specification being fairly complex, the initial version of the simulator was implemented using atomic services, leaving composite services for later iterations of this software. Thanks to this simplification, the simulator is able to use simpler and more proven composition algorithms in the On-line Planner component.

The experiment itself is composed of:

- A series of test runs in which the number of agents gets incremented over time and the number of precompiled plans and the size of the agents’ knowledge base remain constant.
- Each test run is executed using three distinct collaboration algorithms: one where there is no collaboration between agents; another one where agents decide to share all their precompiled plans; and finally one where agents share all their services and the On-line Planner composes new plans dynamically.
- A small error factor has been introduced into the service execution. This means that every time a service is executed there is a 1% chance that it fails. This kind of error forces plan repairing to be invoked.

Different algorithms have been developed to compare the approach proposed in this work with the most common BDI approaches. The first algorithm, where there is no collaboration between agents, simulates a standard BDI system where each agent has its own goals and intentions and tries to perform its desires with no help. The second algorithm introduces the social aspect, where agents have the skill of sharing their plans to improve the success degree but with no possibility of generating new plans. Finally the third algorithm represents the proposal shown in this work, where agents are aided by the Operating System in the composition of plans and they share not only their plans, but also their services in order to make possible the plan composition.

In Figure 8 the results of the experiment are shown. The \(x\) coordinate shows the number of agents running while they are trying to reach their goals. The \(y\) coordinate show the success degree of goal completion, from 1 to 100%. The chart shows how the success degree is very low (near to 0%) when the agents have as their only way to reach their goals a small set of precompiled plans. This degree grows conspicuously when the agents share their precompiled plan sets between them and finally has a very high and convergent success degree when they use the system capability of composing on-line plans using all the available services in the OS.

Analyzing the obtained results, the most important part of this execution model is the selection of the plans to execute. The module in the simulation is able to deliberate about running agents, what are their goals, how all agent goals interact between them and what services are available to meet these goals. Agents maintain their autonomy because they are able to deliberate about their own goals: which to
activate at some point, which are in conflict, and so on. But in order to bring this deliberative process to an environment where there are multiple agents running on the same resources (hardware, services, CPU ...) it is more desirable to have a higher level where they can reason intelligently about the running resources. That’s why the highest level of deliberation is encapsulated inside the execution module, which ultimately will run the agents plans.

Developing the deliberative engine as part of the execution module itself also allows us to handle events (software and hardware) that occur in the system and that may force changes, as a currently active goal override which is running with a plan or a change in the knowledge base of agents.

The fact of using dynamic service composition might seem a priori disadvantage: what happens if a service becomes unavailable? However, what this model offers is the ability to recover from system failures and develop an entirely new plan with updated services available at each time. It is possible to not only recover from failure but also look for a better quality composition of services than the original that solves the same problem. In this way system applications are always updated. The execution of applications in this new concept of execution is not necessarily always the same, it produces the same result but different paths each time. It is always seeking the highest quality performance, as we have seen in the example above. In our example, after repairing the plan not only an unavailable service has been replaced, but the planner has discovered a new service offering a higher quality plan (minimum execution time, greater precision, etc), so it was introduced at the same time of the plan repair.

An interesting added advantage of this architecture is that the invocation of services may be remote. As long as it is a service-oriented architecture, each service used by a plan can be taken from the agent that executes the plan or not. This distributed architecture allows for a high reuse of components automatically, as plans can be built using a conditional planner.

VI. Conclusions

This work has presented a goal-oriented execution module for an operating system that enables running entities to be agents. This proposal is designed to introduce a model of goal-directed execution to improve the flexibility and dynamism of the system, allowing the selection of different plans that meet the selected goals. The environment in which an agent runs can vary, so it is an interesting advantage the ability to select each time the plan that best suits the goals of the agent. Also, the system fault tolerance can be improved: given a system where the execution of a plan failed (because of the fail of a service or the unavailability of a particular plan), the execution module itself would be responsible for repairing the plan in a completely transparent way to the agent.

Another advantage of the goal-directed execution model is the possibility of refine plans for solutions of more quality. We define a higher quality plan when it improves any feature like execution time, plan trust, precision, etc. This would be done looking for a more efficient service composition, with higher confidence values or a lower cost, given some classification for the services within the plan.

In this model, the execution module is not only limited to run the applications requested by users, but carries some intelligence to the execution of agents. The goal-oriented and distributed computing offers two advantages: first, it allows us to compose applications with a high code reusability and with a high speed. This is done giving the agent only the set of goals that we want to reach, without necessarily saying how it should do. Second, we introduce a high introspection of the code, having the execution module the knowledge of what system applications (agents) intend to carry through their goals. Thanks to this informed knowledge about the runtime environment, the operating system has an omniscience on their applications that can greatly help the process of scheduling and conflict resolution.

As future work is highlighted the possibility of redesign certain components of the operating system with this execution model once introduced the agent support within an operating system. For example, some system drivers or services offered by the operating system itself. Thus, the operating system components and drivers can benefit from the advantages of having an execution model based on goals and could handle new and unknown devices with no prior knowledge through the dynamic composition of the plans.

ACKNOWLEDGEMENTS

This work is partially supported by the CSD2007-00022, TIN2008-04446, TIN2009-13839-C03-01 spanish projects and BES-2006-13320 grant.

REFERENCES


