Second National Workshop on
"Control Architectures of Robots: from models to execution on distributed control architectures"
May 31 and June 1st, 2007 - Paris - FRANCE

Towards an Adaptive Robot Control Architecture

Noury Bouraqadi
bouraqadi@ensm-douai.fr
Ecole des Mines de Douai
France

Serge Stinckwich
Serge.Stinckwich@info.unicaen.fr
GREYC - Université de Caen
France

Abstract

Robotic Urban Search And Rescue (Robotic USAR) involves the location, extrication, and initial medical stabilization of victims trapped in confined spaces using mobile robots. Such rescue operations raise several issues. Part of them are studied in the AROUND project. The AROUND (Autonomous Robots for Observation of Urban Networks after Disaster) project aims at designing an automated observation system for disaster zone in developing countries like Vietnam. The idea is to deploy a large number of autonomous mobile robots able to self-organize in order to collect the information in impacted urban sites and to dynamically maintain the communication links between rescuers.

Robots involved in rescue operations have to be reactive while smart enough to deal with complex situations. Hybrid agents seem to be valuable architectures for controlling such robots. Such architectures combine a fast reactive layer with a more deliberative one dealing with long term planning. However, most existing models of hybrid agents commit in early design stages to some particular software agent architecture. The resulting robots fit then only a restricted application context. They quickly become inappropriate when the execution context changes.

One possible change in the execution context is the use of robots with different capabilities and resources. The same missions can be performed differently (reactively or in a more deliberative way) according to robots resources. Therefore, it is interesting to be able at deployment-time to tune agents "hybridity", i.e. switch some tasks from the reactive layer to the deliberative one or vice versa. That is adapting the agent architecture statically between two rescue operations.

Robot's control architecture should not only bear static adaptation, but it should also allow dynamic adaptation. Robots controlled by such an architecture has to react to the evolution of their environment (evolution of resources, robot failures, ...) in order to be as efficient as possible.

In this paper we present our on-going work on component-based adaptive hybrid agent architectures. Our approach relies on the InteRRaP hybrid agent model which is extended with reflective capabilities. The resulting adaptive architecture will be experimented to control robots within the AROUND project.

Keywords: Adaptation, Agent Architectures, Software Components, Mobile Robots, Resource Awareness

1 Introduction

1.1 Context: Robotic Urban Search and Rescue

Robotic Urban Search and Rescue involves the location, extrication, and initial medical stabilization of victims trapped in confined spaces using mobile robots. The idea of using robots to aid human rescue after a major natural or human induced disaster is not new: the first examples were about wildfire rescue [KN83] and after the Oklahoma city bombing in 1995 [Mur04],

but it was only after the New York WTC attack in 2001 that robots have been used in real situation. Such rescue operations raise several interesting issues. Part of them are studied in the AROUND project. The AROUND (Autonomous Robots for Observation of Urban Networks after Disaster) project aims at designing an automated observation system for disaster zone in developing countries like Vietnam. The idea is to deploy a large number of autonomous mobile robots equipped with sensors that enable them to make raw observations. They will also serve as a support for a dynamically deployed communication network between the rescue teams (see figure 1). Their tasks are then twofold:

- **Reconnaissance**: (a) to collect information about the number and location of casualties, dangerous situations (gas leaks, live wires, overhanging walls, unsafe structures) or anything which might endanger rescuers and survivors; (b) to determine the possibility of access to the casualties in an unstructured environment about which little sure information exists.

- **Covering**: (a) to explore, in a "smart" way, as much terrain as they can once they have been deployed; (b) respect the communication constraints and stay in touch with others in order to transmit the perceptions (if we assume that some are "connected" to servers) and to act as relays in a communication network between possibly distant rescue teams.

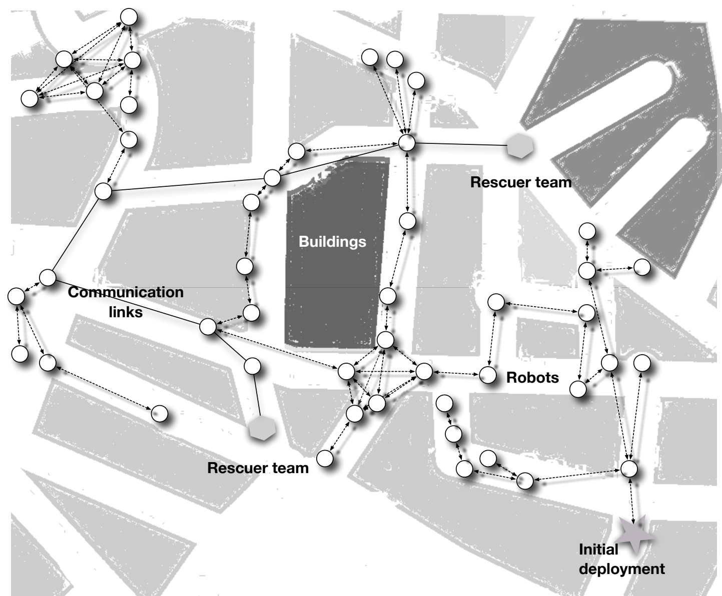

Figure 1: AROUND project

AROUND robots should be able not only to coordinate their behaviors in order to solve theirs tasks, but also adapt to the continously changing situation. One way to take into account coordination and adaptation in the design of the robot controller is to use a software agent architecture. Multiagent Systems (MAS) research is the study of the collective behavior of a group of possibly heterogeneous agents with potentially conflicting goals. Agents are now well known effective abstractions in order to model and build complex distributed applications,

like robotic ones. Agents have their own thread of control (autonomy) and their own internal goals (intentions), thus localizing not only code and data, but also invocation. In the multi-agent vision, the application developer simply identifies the agents suitable to solve a specific problem, and they organize themselves to perform the required functionality. In our context of Multi-Robot Systems (MRS), we would like to associate an agent with each autonomous robot. Although work in both MAS and MRS deals with multiple interacting entities, it is still an open question as to whether techniques developed in the MAS community are directly applicable to the embodied MRS community. In MRS, the resources of the robot and the environment heavily influence the performance of the system and, therefore, cannot be completely ignored.

1.2 Motivation for Adaptation

Adaptation is generally one of the most desirable properties prescribed to agent in the MAS community. The aim of adaptation is to approximate an optimal behaviour with respect to available resources. This is correlated to the bounded rationality principle drawn initially by Herbert Simon [Sim55] that differ from the classic perfect rationality approach by taking into account the available limited resources used by the agent. In [Zil95], three levels of adaptation are distinguish : (1) *resource-adapted systems*, that are pre-configure to a specific domain, (2) *resource-adaptive systems*, that are able to react to resource modifications and (3) *resource-adapting systems*, that explicitly manage and represent resources.

Robots involved in rescue operations have to be reactive while smart enough to deal with complex situations. Hybrid agents seem to be valuable architectures for controlling such robots. A such architecture combines a fast reactive layer with a more deliberative one dealing with long term planning. However, most existing models of hybrid agents commit in early design stages to some particular software agent architecture. The resulting robots fit then only a restricted application context, because they are only *resource-adapted*. They quickly become inappropriate when the execution context changes.

One possible change in the execution context is the use of robots with different capabilities and resources. The same missions can be performed differently (reactively or in a more deliberative way) according to robots resources. Therefore, it is interesting to be able at deployment-time to tune agents "hybridity", i.e. switch some tasks from the reactive layer to the deliberative one or vice versa. That is adapting the agent architecture statically between two rescue operations.

Supporting static adaptation is not enough for mobile robots. Autonomous robots need to dynamically adapt to resource evolutions while performing their tasks. *Resource-adaptive* architectures allow addressing dynamic adaptations. However, such architectures are ad hoc solutions that can not be reused and scaled. Therefore, an ideal robot control architecture should be *resource-adapting*.

2 Adaptation in Questions

Adaptation is the process of conforming a software to new or different conditions [KBC02]. The history of adaptation dates back to the early days of computing, when "self-modifying code" was used for dynamic optimizations in programs. Nonetheless, such programming was generally considered as bad practice due to issues such as program inconsistency and difficulty in debugging. More recently, interest in adaptation increased considerably, due in part to

the needs of ubiquitous computing or autonomous systems, more generally to all the the self-managing systems that deals with a changing environment. The adaptation process can involve different actors that participate to at least one of the following 3 steps:

- Triggering the adaptation: The actor responsible of this step detects that an adaptation is required. This detection implies sensing and analyzing some indicators about the pertinence of adaptations.
- Deciding the adaptation to perform: This step results into the definition of a set of changes representing the adaptation to perform. The output of this step may include also the when to perform the adaptation and how to perform it (e.g. adaptation should be performed atomically or not).
- Performing the adaptation: Decisions made in the previous step are realized at this stage. Mechanisms used here should ensure that changes will be performed smoothly, especially in case of run-time adaptation.

In order to compare adaptation solutions, we introduce the following 4 dimensions base on existing criteria in literature [Sen03, Ket04, Dav05]:

What is adapted? This dimension is about the impact of adaptations, that is the kind of changes performed.

- Changing the value of some parameters within a predefined range (e.g. size of a cache). This is the simplest possible adaptation, but also the one with the least impact. It don't change algorithms and must always be anticipated. Besides, it does not scale up well, since having too much parameters makes the software difficult to develop, and maintain.
- Re-organizing the software either from the logical point of view or from the physical one. A logical re-organization stands for changing the connections between the software's building blocks (e.g. changing the superclass of some class). A physical re-organization stands for changing the location of some part of distributed software.
- Additions, suppressions and replacements of some software entities. These operations allow performing totally unplanned adaptation. The software can even be deeply changed. New building blocks implementing new functionalities or new strategies can be introduced.

Who performs the adaptation? This dimension is about the autonomy of the adapted software. The software can adapt itself, or it can be adapted by another software or by a human (developer, administrator, end user). Actually, there are degrees of autonomy, since each step of the adaptation process can be delegated to a different actor (human or not). There are even more autonomy degrees since the triggering and the decision steps can be more or less *anticipated* by developers.

- The triggering can be either fully done by a human. It can be automated by having a software that matches some sensed data with some patterns specified by a human. An even more advanced triggering autonomy is that the software autonomously (without any data from humans) that an adaptation is needed (i.e. the software's functioning isn't "satisfactory" anymore).

- The decision process can be fully done by a human. It can be automated so the software selects some adaptation among a set provided by a human (e.g. the developer). Last the decision can be fully autonomous if the software builds "from scratch" the set of changes to perform.

When does the adaptation occur? This dimension is about the point in time when adaptation is performed.

- Statically: the software is adapted during development stages (i.e. compile-time). Static adaptation may also be performed later in the software's life-cycle, on deployment or load-time when more data about the actual execution context can be collected.
- Dynamically: the adaptation is performed after the software is started. Delaying adaptations to run-time allows making more relevant decisions as compared to static adaptations. However, run-time adaptation is more challenging compared to the static one. Care must be taken regarding the coherence of data and computations.

How is the adaptation performed? This dimension is about mechanisms and strategies used to perform the adaptation, such as solutions to ensure software coherence (data and computations) while dynamically adapted. The following criteria allow comparing different approaches:

- Ease of use: this criterion refers to the effort that developers need to provide in order to use a mechanism or a tool to perform adaptation. This is tightly related to the declarativeness, abstractness and expressiveness of the solution. Expressiveness includes the ability to define adaptations on only part of the software.
- Transparency: refers to the availability of software's functionality during adaptation. Transparency is maximized if the software "clients" (humans or other software) are as less disturbed as possible (e.g. no freeze of the GUI).
- Efficiency: corresponds to the amount of resources (e.g. memory, cpu, energy) required to perform adaptations. Ideally, adaptations should be as efficient as possible, especially in embedded systems, where resources are scarce.
- Control: refers to the management and coordination of adaptation operations. It can be centralized, if a single actor (human or software) supervises all operations. Conversely, adaptation control can be distributed, which is more suitable for distributed systems, but raises coordination issues.
- Separation of Concerns: refers to the separation between the code that support adaptation and the code that is actually adapted. Separation of concerns is a desirable software engineering quality, as highlighted by approaches that improve modularity such as Aspect-Oriented Software Development [KLM+97, FEAC05] and Component-Based Software Engineering [SGM02].

3 An Abstract Adaptive Robot Control Architecture

3.1 Software Components are Suitable for Adaptation

Software component [SGM02] is a programming paradigm that aims at going beyond Object-Oriented programming from the point of view of modularity, reuse and improvement of soft-

ware quality. Indeed, a software component is a software entity which explicits its dependencies and interactions with other components and resources it relies on. The exact definition of what is a component is still an open issue, even though there exist multiple component models [DYK01, Gro04, BCS02, PBJ98]. However, most of them comply with Szyperski abstract definition stating that "A component is a unit of composition with contractually specified interfaces and explicit context dependecies only. A software component can be deployed independently and is subject to composition by third parties" [SGM02].

Software components are more suitable for supporting adaptation as compared to other programming paradigms. Since dependencies are explicit, a component can be safely disconnected from a software and replaced by another one. The impact of the replacement can be pre-determined because the connections and interactions between components are explicit. No costly, complex and subject to failures analysis is required to identify the consequences of such adaptation. Of course, if performed at run-time, this operation requires some mechanism ensuring coherence of data and computations.

3.2 Requirements for an Adaptive Robot Control Architecture

Because of aforementioned benefits of software components, our first requirement is that the targeted architecture should be component-based. The other requirements are listed below based on adaptation dimensions provided in section 2.

What is adapted? We would like to perform arbitrary complex adaptations. Therefore, adaptations should impact both component's attributes, their connections to each other, and their locations. It should also be possible to add, remove or replace components in order to cope with unpredictable adaptations. In case of a hierarchical component model, these operations can be performed at different levels. Adaptations should not only be possible on top-level components, but they also should be applicable inside any composite component. Therefore, it should be possible to alter the set of sub-components of any composite component.

Who performs the adaptation? Mobile robots in our project may operate out of reach of humans. Therefore, they have to be autonomous and make decisions without human supervision. Such decisions may include adaptations. Human driven adaptations should still be possible, however.

When does the adaptation occur? Adaptations have to be performed at run-time. Indeed, stopping controllers may introduce delays that can be risky for robots in a dangerous area. And it's undesirable regarding the emergency of rescue operations. Moreover, since robots may adapt themselves autonomously, their control software should not be stopped in order to actually perform adaptations.

How is the adaptation performed? We can stress that transparency and efficiency are important criteria since adaptations has to be performed dynamically, and robots have limited resources. Moreover, since efficiency may vary according to resources variations, the targeted architecture should allow tuning the adaptation process. So, the adaptation process should be itself adaptive.

3.3 Abstract Architecture

In this section, we present our abstract architecture for robot control. This architecture is abstract because: i) it's incomplete. We highlight main components and those important to adaptation. ii) it does not make any assumption about the component model to use, iii) no specification is given for the actual implementation of used components, and iii) their interfaces are not specified either. Only their roles are provided.

3.3.1 Overview

As shown by figure 2, our abstract architecture is a hybrid layered architecture inspired from InteRRaP [Mül96] extended with *reflective* capabilities. Thanks to these capabilities, the software agent is able to observe and adapt itself [Smi84, Mae87] to best fit its execution context.

The architecture is organized in three layers running concurrently. Each layer may include adaptation behavior. Performed adaptations may involve any part of the agent.

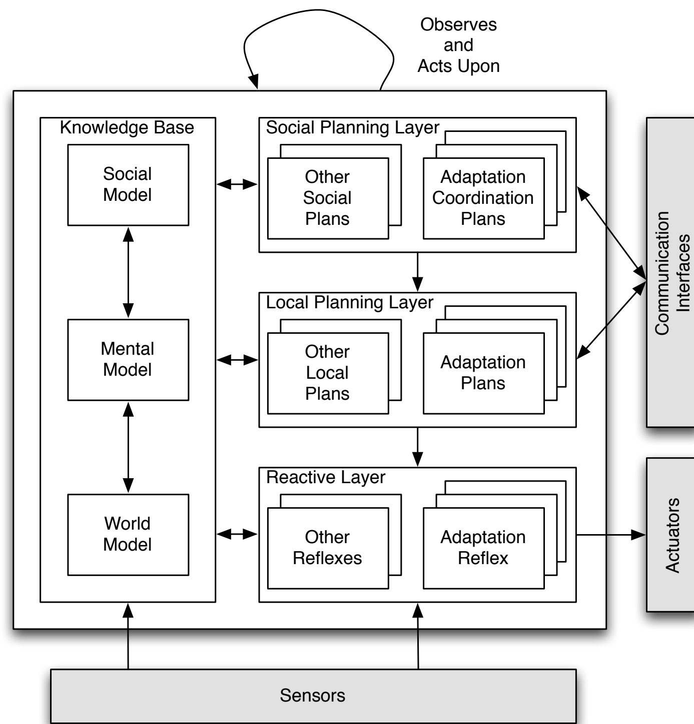

Figure 2: Abstract Adaptive Robot Control Architecture

The top-most layer is the *social planning layer* that performs plans for coordination with other agents. Such plans may include ones related to coordinating adaptation tasks. For example, in the AROUND project, robots disseminated in some area have to decide which ones should collect data and which one will be in charge of maintaining the mobile ad hoc network and route messages. And in case the fleet is split into two or more groups, members

of each group has to re-organize in order to set up again a communication path to rescue central servers and still collect data.

The second layer is the *local planning layer* performs plans for making the agent have a rational functioning. Performed plans can be related to adaptation. They not only decide *which* adaptation to perform, but also *when* and *how* to perform it. An adaptation example is making a robot switch from the exploration task to the ad hoc network maintenance task, without having all skills for this new task. The plan is then to first roam until finding an other robot that can provide the missing skills, then came-back to the original location, and last actually perform the adaptation.

The bottom-most layer is the *reactive layer*. It groups reflexes that are triggered either by some sensor's value or by a change in the knowledge base. Reflexes may include adaptations of any part of the agent, including the knowledge base and the social planning and the local planning layers. These reflexes allow efficient adaptation decisions for cases where the robot doesn't have enough resources for planning adaptations.

The knowledge base is organized in a hierarchy of abstractions. Sensed data allows updating the *world model*, i.e. the agent's beliefs about its environment. Of course such beliefs can be of high-level which involves fusion of sensed data and inferences. The knowledge base also includes the agent's beliefs about itself (*mental model*) and about other agents (*social model*). These beliefs may include knowledge that may trigger adaptations (e.g. battery low) or be used for adaptations (e.g. cost of).

3.3.2 Adaptation Dimensions of Our Architecture

We present here our architecture according to adaptation dimensions presented in section 2.

What is adapted? Since our architecture is reflective, every part of the agent can be adapted. The knowledge base and its inference mechanism can be altered. Every layer can be adapted to add or remove behaviors or change the decision process that selects the behavior to run. A notable adaptation is replacing a reflex with a plan or vice versa and hence making the agent become more or less reactive or deliberative. We call such adaptations *hybridity tuning*. Since the agent is reflective, adaptations can also concern adaptation behavior. Indeed, adaptation planning or collaborations can prove expensive regarding available resources. The agent then can switch to a more reactive adaptation or even dismiss all components that drive adaptations.

Who performs the adaptation? Agent designers can intervene statically by for instance removing a layer (e.g. agents without collaboration skills) or deciding which adaptations to implement as reflexes and which others to implement as plans. Administrators can trigger adaptations during the activity of agents. Besides, at run-time each agent can autonomously perform its own adaptations. These adaptations can be driven by any layer, including the social planning layer, since collaborative adaptations can be possible. Two or more robots may agree to perform some adaptations and then perform them in sync.

When does the adaptation occur? Our architecture supports both static and dynamic adaptations. At run-time, the agent decides which adaptations to perform according to faced situations and resource availability. An example of adaptation, a mobile robot navigating within a maze can have a deliberative behavior to build its path. In case

energetic resources are low, a possible adaptation can be to dismiss the navigation plan and adopt only a reactive obstacle avoiding behavior. Static adaptation is performed by the robot designer, often according to the targeted platform and previous knowledge about the execution context. It consists in providing reflexes and plans that perform the same tasks according to different strategies. Providing alternative strategies apply also for adaptation. It is the result of the designer's decision about how much freedom is given to the agent to tune its hybridity.

How is the adaptation performed? In order to keep the adaptation process efficient even in case of resource scarcity, the adaptation support is itself adaptive. This means that a possible adaptation is to change the way to perform future adaptations. When resources are very low, only reflex adaptations should be possible. Adaptation plans that consume more resources, should be dismissed. Symmetrically, when more resources are available, adaptation plans that compute smart cognitive adaptation decisions should be available.

4 Application to Anticipatory Agents

In this section, we instantiate our abstract architecture for building an *anticipatory agent*. An anticipatory agent is a hybrid agent which is able to adapt itself according to predicted changes of itself and its environment [Dav96, Dav03]. Such an agent combines a reactive fast layer with a cognitive layer capable to perform predictions and adaptations to avoid undesired situations before they occur actually.

In order to make our anticipatory agent architecture as much explicit as possible, we used the MALEVA software component model[BMP06] to design and implement it[1].

4.1 MALEVA: A Software Component Model Expliciting Data and Control Flows

In this example, we use the MALEVA component model [BMP06]. A MALEVA component is a run-time software entity providing encapsulation like objects, while expliciting its interactions with other components. MALEVA components interact only through their interfaces. Interfaces can be of two kinds: data interfaces or control interfaces. Data interfaces are dedicated to data exchange, while control interfaces are dedicated to control flow.

A component can be either active or passive. A passive component is a component that does perform some computation only after being triggered through one of its control input interfaces. Once the component computation is over, it stops until being again triggered. As opposite, an active component don't need to be triggered to act. It has a thread and thus can autonomously run.

A component can do arbitrary computations, including reading data available on its data input interfaces, sending data to other components through its data output interfaces, and triggering other components through its control output interfaces. The activity (i.e. its computations) of a component is suspended when triggering other components. The activity is resumed once the triggered components have finished their own computations.

Last, MALEVA is a hierarchical component model. It allows building composite components out of existing component. A composite encapsulates its sub-components. Sub-

[1] We used MalevaST the Smalltalk implementation of the MALEVA component model for our implementation. MalevaST is available under the MIT licence at: `http://csl.ensm-douai.fr/MalevaST`

components can not be reached unless some of their interfaces are exported by the composite, i.e. when the composite makes the sub-component interfaces available outside.

From the above quick description, we can see that the MALEVA components are close to building blocks of the Brooks subsumption model in their encapsulation and interaction through data exchange [Bro85]. This is why we chose MALEVA in order to define and implement our anticipatory hybrid agent architecture (see section 4.3). However, we have chosen MALEVA also because it explicits the control flow and hence allows identifying concurrency issues more easily.

4.2 From Prediction to Anticipatory Agents

Prediction is a statement made about the future. A typical example is the act of predicting a trajectory such in the case of a moving object. Anticipation refers to taking prior actions on the basis of prediction about the future. These actions can be directed to avoid or encourage a particular future. Robert Rosen's [Ros85] standard definition of anticipatory systems characterizes an anticipatory system as one "containing a predictive model of itself and/or of its environment, which allows it to change state at an instant in accord with the model's predictions pertaining to a latter instant". Thus, anticipation could be viewed as a form of model-based adaptation.

From the definition of Rosen, Davidsson [Dav03] defines a very simple class of anticipatory agent system: it contains a causal system S and a model M of this system that provide predictions of S. As the model M is not a perfect representation of the reactive system, this is called a quasi-anticipatory system. This architecture is rather coarse-grain, it is only composed of 5 parts:

- Sensors: provide information about the agent environment.
- Effectors: allow the agent to act upon its environment.
- Reactor: drives the effector in reaction to latest information provided by sensors.
- World Model: is an abstract view of the agent's environment built based on data collected using sensors.
- Anticipator: performs reactor modifications based on the world model.

4.3 A Maleva-based Anticipatory Agent Architecture

Figure 3 shows that our agent architecture is an assembly of six Maleva components. We can see that in this example, the abstract model was adapted at design time by removing some parts. The knowledge base is restricted to a single world model component. The social plan layer is dismissed. The "Anticipator" component plays the role of the local plan layer but it does only plan adaptations to perform on the reactor. The reactor provides two generic interfaces for modifications input: one for modification data flow and the second for modification control flow. The former allows the anticipator to provide modifications to be performed on the reactor, while the latter allows the anticipator to trigger the modifications. These two interfaces can be viewed as the so-called "meta-interfaces" in the work on Open Implementation [Kic96], since they allow a disciplined modification of the reactor.

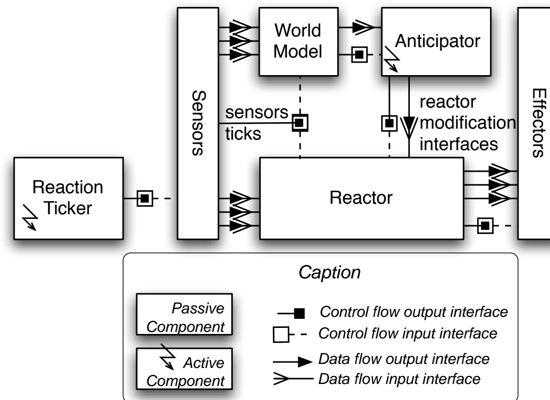

Figure 3: Maleva-based Anticipatory Agent Architecture

The “Reaction Ticker” is a generic active component that drives the agent’s reaction. It defines the frequency at which the agent will sense its environment and react to changes. Indeed, the “Reaction Ticker” triggers the Sensors component every m milliseconds, where m is the duration between two ticks and depends on the application context.

The Sensors component[2] collects data from the agent’s world and propagate it through its data interfaces to both the reactor and the world model. Then, it triggers the activity of both the anticipator and the reactor. When getting triggered, the reactor decides the appropriate reaction to perform and translates this decision into data propagated to the Effectors component[3].

Once updated by sensed data, the world model provides the new world state to the anticipator. The anticipator is an active component that runs concurrently to the reactor. Its ticking frequency is higher than the “Reaction Ticker” one, since the anticipator has to work faster than the reactor, in order to make useful adaptations. This frequency can be easily changed to tune the anticipator consumption of resources (computing, energy, ...), particularly in case of embedded devices with low capabilities. This frequency can even be changed dynamically, according to resource evolutions, such as the battery level in a mobile robot.

5 Related Work

There are numerous works that tackle the adaptation problem from a software perspective. Some of them deals more explicitly with adaptation in the context of mobile robots or autonomous agents.

ARM ARM [Mal06] is an Asynchronous Reflection Model used to control modular robots. This work reconsiders the traditional "implements" link between the object-level layer and the meta-level layer in a reflective system, that seems to be no more appropriate in an embedded or

[2] Actually, the Sensors component can be a composite with multiple subcomponents corresponding to different sensors.

[3] Actually, the Effectors component can also be a composite with multiple subcomponents corresponding to different effectors.

distributed context, either because the system lacks ressources or there is no more a centralized state. ARM main idea is to found the "right combination of connection and detachment" between the base and the meta-level by using a new reflective model based on an asynchronous publish/subscribe communication model. Adaptations may lead to split an agent by "moving" higher layers to remote hosts. Of course such adaptations need to take into account network latencies.

Touring Machine and InteRRaP In [Fer92], Ferguson describes Touring Machine, a vertical layered architecture for autonomous agents, that consists of three modules: a reactive layer, a planning layer and a modeling layer. The modeling layer offers the possibility of modifying plans based on changes on its environment that cannot be dealt with the replanning mechanims of the planning layer. Similar ad-hoc adaptation mechanisms are found in the InteRRap ('Integration of Reactivity and Rational Planning') architecture [Mül96], a well-known hybrid control architecture. These adaptations are made from top to bottom layers and there is no possibility to modify the adaptation machinery. Moreover, these architectures are not reflective.

Meta-Control Agents Meta-control agents introduced by Russell and Wefald [RW91] are a specific form of horizontal modularization very similar to the layered hybrid architecture. A rational agent has to solve two problems: optimize its external behavior according to the available ressources in the environment and optimize its internal computing with respect to computational ressources. These two problems are instantiated as two distincts levels: an object-level sub-system and a meta-level sub-system monitoring and configuring the first one. Hence, resource-adapting mechanisms are proposed as an architectural principle to balance agents computations between several goals, tactics, and skills according to overall system constraints. On contrary to the layered architectures like InteRRaP, the decoupling between of the two level enhances separation of concerns in adaptation mecanisms. Using the InteRRaP-R architecture[Jun99] Jung made an attempt to mix InteRRaP hybrid agents and the meta-control. Each layer is responsible of adapting the layer below according to resource evolutions. However, nothing is said about how and when the adaption should occur.

MaDcAr Grondin et al. [GBV06] introduced MaDcAr a Model for Automatic and Dynamic Component Assembly Reconfiguration. This model provides an abstract description for an engine that allows assembling and re-assembling a component-based software, either statically or at run-time. This engine has four inputs: a set of components to assemble, an application description providing a specification of all valid assemblies, an assembling policy and a set of sensors observing the execution context. According to sensed data, and based on the assembling policy and the application description, the engine builds a constraint satisfaction problem (CSP). The engine includes a constraint solver that allows solving the CSP and finding out which components to assemble and according to which assembly blueprint. This approach allows both building an application by automatically assembling components and re-assembling the application according to the context. The set of components to assemble and the application description may also vary, which in return causes a re-assembling.

Safran SAFRAN [Dav05] is an extension of the Fractal component model [BCS02] that supports run-time adaptations. As opposite to MaDcAr Safran's adaptation descriptions are

spread over all components. The software's designer assigns to each component an adaptation policy that states when to trigger adaptations and which adaptations to actually perform. This policy is expressed using a collection of reactive rules of the form *Event–Condition–Action*. Events are generated by WildCAT, a part of Safran's platform which observes the software's activity and its context. Each component selects within its adaptation policy, rules that match received events. A second filtering is performed to keep only rules which condition part evaluates to true. Last, adaptation is performed by execution action parts of remaining rules.

6 Conclusion

Mobile robots in large and particularly those involved in Urban Search And Rescue operations require a control architecture that is both fast and smart. Hybrid agent architectures seem appropriate since they mix reactive behaviors with cognitive ones. So, they have reflexes to quickly react to fast events, while they are still able to do long term plans. However, in existing hybrid architecture the decision to implement an agent's behavior as a reflex or as plan is performed during development. The resulting control architecture does not take into account the evolution of the robot's resources and unplanned variations of its surrounding.

In this paper, we introduced a component based abstract adaptive hybrid agent architecture that is suitable for controlling mobile robots in a changing environment. Based on the InteRRaP [Mül96] model, our architecture has three layers for expressing reflexes, local plans and collaboration (i.e. social) plans. In order to support adaptation and tune "hybridity"[4], we introduced reflexive capabilities that can be expressed in every layer. So, adaptations can be expressed either as reflexes or as local plans or even as social plans to allow coordination of adaptations of multiple agents. Reflective capabilities allow not only to trigger and perform adaptations, but they also allow to decide when and how to perform adaptations. Besides, our architecture is fully reflective. So, even adaptation description can be adapted for example to allow only reactive adaptations if deliberative adaptations are too costly regarding available resources.

The presented abstract architecture is a work in progress. We are currently in the process of validating it through projections to concrete implementations. We described in this article a first projection to Davidsson's quasi-anticipatory agents [Dav03] using the Maleva component model [BMP06]. Further investigations will be performed within the context of a robotic USAR project. Experiments will be conducted with a fleet of wheeled mobile robots that have to autonomously cooperate in order to explore a damaged area.

References

[BCS02] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *WCOP'02–Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, Malaga, Spain, Jun 2002.

[BMP06] J. P. Briot, T. Meurisse, and F. Peschanski. Une expérience de conception et de composition de comportements d'agents à l'aide de composants. *L'Objet*, 11:1–30, 2006.

[4] i.e. switch some tasks from the reactive layer to the deliberative one or vice versa

[Bro85] R. Brooks. A robust layered control system for a mobile robot. Technical report, 1985.

[Dav96] P. Davidsson. A linearly quasi-anticipatory autonomous agent architecture : Some preliminary experiments. In *Distributed Artificial Intelligence Architecture and Modelling*, number 1087 in Lecture Notes in Artificial Intelligence, pages 189–203. Springer Verlag, 1996.

[Dav03] P. Davidsson. A framework for preventive state anticipation. In Springer-Verlag, editor, *Anticipatory Behavior in Adaptive learning Systems: Foundations, Theories and Systems*, volume 2684 of *Lecture Notes in Artificial Intelligence*, pages 151–166, 2003.

[Dav05] P.-C. David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Université de Nantes, July 2005.

[DYK01] L.G. DeMichiel, L.Ü. Yalçinalp, and S. Krishnan. *Entreprise Java Beans Specification Version 2.0*. 2001.

[FEAC05] R. Filman, T. Elrad, M. Akşit, and S. Clarke, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[Fer92] I. A. Ferguson. *TouringMachines: An architecture for dynamic, rational, mobile agents*. PhD thesis, University of Cambridge, 1992.

[GBV06] G. Grondin, N. Bouraqadi, and L. Vercouter. Madcar: an abstract model for dynamic and automatic (re-)assembling of component-based applications. In *The 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, June 29th – 1st July 2006.

[Gro04] Object Management Group. *Common Object Request Broker Architecture: Core Specification*. March 2004.

[Jun99] C. G. Jung. *Theory and Practice of Hybrid Agents*. PhD thesis, Universität des Saarlandes, 1999.

[KBC02] A. Ketfi, N. Belkhatir, and P.-Y. Cunin. Adapting applications on the fly. In *ASE '02: Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 313, Washington, DC, USA, 2002. IEEE Computer Society.

[Ket04] A. Ketfi. *Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels*. PhD thesis, Université Joseph Fourier, December 2004.

[Kic96] G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of ECOOP'97*, number 1241 in LNCS, pages 220–242. Springer-Verlag, June 1997.

[KN83] A. Kobayashi and K. Nakamura. Rescue robots for fire hazards. In *Proceedings of the first International Conference on Advances Robotics (ICAR'83)*, pages 91–98, 1983.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[Mal06] J. Malenfant. An asynchronous reflection model for object-oriented distributed reactive systems. In *First National Workshop on Control Architectures of Robots*, April 2006.

[Mül96] J. P. Müller. *The Design of Intelligent Agents: A Layered Approach*. Number 1177 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1996.

[Mur04] R. R. Murphy. Rescue robotics for homeland security. *Communications of the ACM*, 47(3):66–68, 2004.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, pages 43–52. IEEE Press, 1998.

[Ros85] R. Rosen. *Anticipatory Systems - Philosophical, Mathematical and Methodological Foundations*. Pergamon Press, 1985.

[RW91] S. Russell and E. Wefald. *Do the right Thing*. MIT Press, 1991.

[Sen03] A. Senart. *Canevas logiciel pour la construction d'infrastructures logicielles dynamiquement adaptables*. PhD thesis, Institut National Polytechnique de Grenoble, November 2003.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 2002.

[Sim55] H. A. Simon. A behavior model of rational choice. *Quarterly Journal of Economics*, 69:99–118, 1955.

[Smi84] B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, Salt Lake City, Utah, USA, January 1984.

[Zil95] S. Zilberstein. Models of bounded rationality. In *AAAI Fall Symposium on Rational Agency*, Cambridge, MA, 1995.